Mental Health Support Chatbot with Multimodal Analysis

Team Details

- **Team Name**: The Data Company
- **Team Leader**: Mitul Srivastava

Project Overview

- Develops a mental health support chatbot with multimodal analysis
- Integrates text, voice, and facial emotion recognition
- Uses advanced NLP models, sentiment analysis, and ABSA
- Incorporates RAG for enhanced user interaction
- Includes safety mechanisms for crisis detection
- Features fallback models for robustness

Load Phi-3 Mini with Safety Fallback

```
# Install required libraries silently to avoid cluttering output
!pip install torch --quiet
!pip install transformers --quiet
!pip install bitsandbytes accelerate --quiet
!pip install sentencepiece --quiet
#!pip install flash-attn --no-build-isolation  # Commented out as it did not work

# Import required libraries for model loading and text processing
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
from transformers import BitsAndBytesConfig
import re

# Define the primary model to be used for text generation
PRIMARY_MODEL = "microsoft/phi-3-mini-4k-instruct"
# Define a fallback model (lighter and more stable) in case primary fails
FALLBACK_MODEL = "tiiuae/falcon-rw-1b"  # "google/gemma-2b-it" is an alternative

print(f"Attempting to load {PRIMARY_MODEL} ...")

try:
    # Configure quantization settings for efficient model loading
    quantization_config = BitsAndBytesConfig(
        load_in_4bit=True,  # Load model in 4-bit precision
        bnb_4bit_quant_type="nf4",  # Use nf4 quantization type
        bnb_4bit_compute_dtype=torch.float16)  # Use float16 for computations

    # Load tokenizer and model for the primary model
    tokenizer = AutoTokenizer.from_pretrained(PRIMARY_MODEL)
    model = AutoModelForCausalLM.from_pretrained(
        PRIMARY_MODEL,
        device_map="auto",  # Automatically map model to available devices
        quantization_config=quantization_config,
        # attn_implementation="flash_attention_2"  # Commented out as it did not work
    )
    ACTIVE_MODEL = PRIMARY_MODEL  # Set active model to primary if successful
    print(f"✅ Successfully loaded {PRIMARY_MODEL}")

except Exception as e:
    # Handle any errors during primary model loading
    print(f"⚠️ Could not load {PRIMARY_MODEL}. Error: {e}")
    print(f"Attempting fallback: {FALLBACK_MODEL}")
    # Load tokenizer and model for the fallback model
    tokenizer = AutoTokenizer.from_pretrained(FALLBACK_MODEL)
    model = AutoModelForCausalLM.from_pretrained(
        FALLBACK_MODEL,
        device_map="auto",  # Automatically map model to available devices
        torch_dtype=torch.float16  # Use float16 for computations
    )
    ACTIVE_MODEL = FALLBACK_MODEL  # Set active model to fallback if successful
    print(f"✅ Successfully loaded fallback {FALLBACK_MODEL}")

def generate_from_model(prompt, max_new_tokens=150, temperature=0.7, top_p=0.9):
    """Generate response from active model with safety + deduplication."""
    # Prepare input tensor for the model
    inputs = tokenizer(prompt, return_tensors="pt").to(model.device)
```

```
# Generate text output with specified parameters
outputs = model.generate(
    **inputs,
    max_new_tokens=max_new_tokens,  # Limit the length of generated text
    do_sample=True,  # Enable sampling for diverse outputs
    temperature=temperature,  # Control randomness of output
    top_p=top_p,  # Use nucleus sampling
    pad_token_id=tokenizer.eos_token_id,  # Set padding token
    eos_token_id=tokenizer.eos_token_id,  # Set end-of-sequence token
    repetition_penalty=1.1,  # Penalize repeated text
    no_repeat_ngram_size=3  # Prevent repetition of 3-grams
)

# Decode the generated output to text
text = tokenizer.decode(outputs[0], skip_special_tokens=True)

# Strip the prompt prefix if included in the output
if text.startswith(prompt):
    text = text[len(prompt):].strip()

# More comprehensive text cleaning
import re

# Remove common unwanted patterns from the text
text = re.sub(r'Assistant:\s*', '', text, flags=re.IGNORECASE)
text = re.sub(r'Bot:\s*', '', text, flags=re.IGNORECASE)
text = re.sub(r'Response:\s*', '', text, flags=re.IGNORECASE)
text = re.sub(r'\[.*?\].*', '', text, flags=re.IGNORECASE | re.DOTALL)  # Remove anything starting with [
text = re.sub(r'User:\s*.*', '', text, flags=re.IGNORECASE | re.DOTALL)  # Remove User: and everything after
text = re.sub(r'\*\*.*?\*\*', '', text, flags=re.DOTALL)  # Remove **bold** text
text = re.sub(r'A:\s*.*', '', text, flags=re.IGNORECASE | re.DOTALL)  # Remove A: responses
text = re.sub(r'Solution \d+:.*', '', text, flags=re.IGNORECASE | re.DOTALL)  # Remove Solution patterns
text = re.sub(r'Instruction \d+:.*', '', text, flags=re.IGNORECASE | re.DOTALL)  # Remove Instruction patterns
text = re.sub(r'---.*', '', text, flags=re.DOTALL)  # Remove everything after ---

# Remove quotes at the beginning and end if they exist
text = text.strip()
if text.startswith('"') and text.endswith('"'):
    text = text[1:-1].strip()
elif text.startswith('"'):
    text = text[1:].strip()
elif text.endswith('"'):
    text = text[:-1].strip()

# Split by newlines and take only the first meaningful response
lines = text.split('\n')
clean_lines = []

for line in lines:
    line = line.strip()
    if not line:
        continue
    # Stop if we encounter patterns that indicate meta-text
    if any(pattern in line.lower() for pattern in ['solution', 'instruction', 'user:', 'bot:', 'assistant:', '[', 'a:', '**
        break
    clean_lines.append(line)

# Join the clean lines
text = ' '.join(clean_lines).strip()

# If text is too long, truncate at sentence boundary
if len(text) > 300:
    sentences = text.split('.')
    truncated = []
    char_count = 0
    for sentence in sentences:
        if char_count + len(sentence) > 300:
            break
        truncated.append(sentence)
        char_count += len(sentence) + 1
    text = '.'.join(truncated).strip()
    if text and not text.endswith('.'):
        text += '.'

# Final cleanup
text = re.sub(r'\s+', ' ', text)  # Multiple spaces to single space
text = text.strip()

# Fallback if text is empty or too short
if not text or len(text) < 5:
    text = "I understand how you're feeling. I'm here to support you."

# Safety check
```

```
    # Safety check
    if is_unsafe_message(text):
        text = "I'm not able to provide a safe response to that, but I care about your well-being. 🧡"

    # Deduplication check (but don't call it here to avoid the extra text)
    # text = soft_duplicate_filter(text)

    return text
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 61.3/61.3 MB 14.1 MB/s eta 0:00:00
Attempting to load microsoft/phi-3-mini-4k-instruct ...
tokenizer_config.json:        3.44k/? [00:00<00:00, 287kB/s]
tokenizer.model: 100%                                                 500k/500k [00:00<00:00, 962kB/s]
tokenizer.json:         1.94M/? [00:00<00:00, 30.0MB/s]
added_tokens.json: 100%                                               306/306 [00:00<00:00, 24.8kB/s]
special_tokens_map.json: 100%                                                 599/599 [00:00<00:00, 44.5kB/s]
config.json: 100%                                                   967/967 [00:00<00:00, 63.8kB/s]
model.safetensors.index.json:         16.5k/? [00:00<00:00, 1.49MB/s]
Fetching 2 files: 100%                                                2/2 [02:04<00:00, 124.60s/it]
model-00001-of-00002.safetensors: 100%                                                4.97G/4.97G [02:04<00:00, 116MB/s]
model-00002-of-00002.safetensors: 100%                                                2.67G/2.67G [01:45<00:00, 31.0MB/s]
Loading checkpoint shards: 100%                                               2/2 [00:36<00:00, 17.07s/it]
generation_config.json: 100%                                          181/181 [00:00<00:00, 13.1kB/s]
✅ Successfully loaded microsoft/phi-3-mini-4k-instruct
```

Cell 2 — Sentiment Analysis with Fallback

```
# Install required libraries silently to avoid cluttering output
!pip install torch --quiet
!pip install transformers --quiet
!pip install accelerate --quiet

# Import required libraries for sentiment analysis
import torch
from transformers import pipeline

# Define the primary sentiment analysis model
PRIMARY_SENTIMENT_MODEL = "cardiffnlp/twitter-roberta-base-sentiment-latest"
# Define a fallback sentiment analysis model if primary fails
FALLBACK_SENTIMENT_MODEL = "distilbert-base-uncased-finetuned-sst-2-english"

try:
    # Initialize sentiment analysis pipeline with the primary model
    sentiment_pipe = pipeline(
        "sentiment-analysis",
        model=PRIMARY_SENTIMENT_MODEL,
        device=0 if torch.cuda.is_available() else -1  # Use GPU if available, else CPU
    )
    ACTIVE_SENTIMENT_MODEL = PRIMARY_SENTIMENT_MODEL  # Set active model to primary if successful
    print(f"✅ Sentiment model loaded: {PRIMARY_SENTIMENT_MODEL}")

except Exception as e:
    # Handle errors during primary model loading
    print(f"⚠️ Could not load {PRIMARY_SENTIMENT_MODEL}. Error: {e}")
    print(f"Loading fallback: {FALLBACK_SENTIMENT_MODEL}")
    # Initialize sentiment analysis pipeline with the fallback model
    sentiment_pipe = pipeline(
        "sentiment-analysis",
        model=FALLBACK_SENTIMENT_MODEL,
        device=0 if torch.cuda.is_available() else -1  # Use GPU if available, else CPU
    )
    ACTIVE_SENTIMENT_MODEL = FALLBACK_SENTIMENT_MODEL  # Set active model to fallback if successful
    print(f"✅ Sentiment model loaded: {FALLBACK_SENTIMENT_MODEL}")

def detect_sentiment(text):
    """Detect sentiment and normalize into (positive, negative, neutral)."""
    try:
        # Perform sentiment analysis and get the first result
        r = sentiment_pipe(text)[0]
        label = r["label"].lower()  # Convert label to lowercase
        score = float(r["score"])  # Convert score to float
```

```
        # Normalize labels to standard categories
        if "pos" in label:
            label = "positive"
        elif "neg" in label:
            label = "negative"
        else:
            label = "neutral"

        return label, score  # Return the normalized label and confidence score

    except Exception as e:
        # Handle any errors during sentiment detection
        print(f"⚠️ Sentiment detection failed: {e}")
        return "neutral", 0.0  # Return neutral with zero confidence if detection fails
```

```
config.json: 100%  929/929 [00:00<00:00, 54.0kB/s]
pytorch_model.bin: 100%  501M/501M [00:10<00:00, 78.9MB/s]
model.safetensors: 100%  501M/501M [00:10<00:00, 73.0MB/s]
Some weights of the model checkpoint at cardiffnlp/twitter-roberta-base-sentiment-latest were not used when initializing Robert
- This IS expected if you are initializing RobertaForSequenceClassification from the checkpoint of a model trained on another t
- This IS NOT expected if you are initializing RobertaForSequenceClassification from the checkpoint of a model that you expect
vocab.json: 899k/? [00:00<00:00, 978kB/s]
merges.txt: 456k/? [00:00<00:00, 5.29MB/s]
special_tokens_map.json: 100%  239/239 [00:00<00:00, 7.49kB/s]
Device set to use cuda:0
✅ Sentiment model loaded: cardiffnlp/twitter-roberta-base-sentiment-latest
```

Cell 3 — Text Emotion Detection with Fallback

```
# Define the primary model for text emotion detection
PRIMARY_EMOTION_MODEL = "j-hartmann/emotion-english-distilroberta-base"
# Define a fallback model for text emotion detection if primary fails
FALLBACK_EMOTION_MODEL = "bhadresh-savani/distilbert-base-uncased-emotion"

try:
    # Initialize text classification pipeline with the primary emotion model
    emotion_pipe = pipeline(
        "text-classification",
        model=PRIMARY_EMOTION_MODEL,
        return_all_scores=True,  # Return scores for all emotion classes
        device=0 if torch.cuda.is_available() else -1  # Use GPU if available, else CPU
    )
    ACTIVE_EMOTION_MODEL = PRIMARY_EMOTION_MODEL  # Set active model to primary if successful
    print(f"✅ Emotion model loaded: {PRIMARY_EMOTION_MODEL}")

except Exception as e:
    # Handle errors during primary model loading
    print(f"⚠️ Could not load {PRIMARY_EMOTION_MODEL}. Error: {e}")
    print(f"Loading fallback: {FALLBACK_EMOTION_MODEL}")
    # Initialize text classification pipeline with the fallback emotion model
    emotion_pipe = pipeline(
        "text-classification",
        model=FALLBACK_EMOTION_MODEL,
        return_all_scores=True,  # Return scores for all emotion classes
        device=0 if torch.cuda.is_available() else -1  # Use GPU if available, else CPU
    )
    ACTIVE_EMOTION_MODEL = FALLBACK_EMOTION_MODEL  # Set active model to fallback if successful
    print(f"✅ Emotion model loaded: {FALLBACK_EMOTION_MODEL}")

def detect_text_emotion(text, min_confidence=0.35):
    """Detect dominant emotion from text with confidence filtering."""
    try:
        # Perform emotion classification on the input text
        res = emotion_pipe(text)

        # Some models return [[{label, score}, ...]], normalize shape
        if isinstance(res, list) and res and isinstance(res[0], list):
            scores = res[0]
        else:
            scores = res

        # Find the emotion with the highest score
        top = max(scores, key=lambda x: x["score"])
        label = top["label"].lower()  # Convert label to lowercase
        score = float(top["score"])  # Convert score to float
```

```
        # Return neutral if confidence is below the minimum threshold
        if score < min_confidence:
            return "neutral", 0.0

        return label, score  # Return the dominant emotion and its confidence score

    except Exception as e:
        # Handle any errors during emotion detection
        print(f"⚠️ Emotion detection failed: {e}")
        return "neutral", 0.0  # Return neutral with zero confidence if detection fails
```

```
config.json:        1.00k/? [00:00<00:00, 4.50kB/s]
pytorch_model.bin: 100%                                329M/329M [00:04<00:00, 104MB/s]
model.safetensors:  22%                                73.1M/329M [00:02<00:07, 34.0MB/s]
tokenizer_config.json: 100%                            294/294 [00:00<00:00, 14.9kB/s]
vocab.json:         798k/? [00:00<00:00, 18.7MB/s]
merges.txt:         456k/? [00:00<00:00, 4.61MB/s]
tokenizer.json:        1.36M/? [00:00<00:00, 20.7MB/s]
special_tokens_map.json: 100%                          239/239 [00:00<00:00, 3.78kB/s]
Device set to use cuda:0
✅ Emotion model loaded: j-hartmann/emotion-english-distilroberta-base
/usr/local/lib/python3.12/dist-packages/transformers/pipelines/text_classification.py:111: UserWarning: `return_all_scores` is
  warnings.warn(
```

Cell 4 — ABSA (Aspect-Based Sentiment Analysis) with Fallback + Keyword Expansion

```
# Import the pipeline module from transformers for text classification
from transformers import pipeline

# Define the primary model for Aspect-Based Sentiment Analysis (ABSA)
PRIMARY_ABSA_MODEL = "yangheng/deberta-v3-base-absa-v1.1"
# Define a fallback model for ABSA if primary fails
FALLBACK_ABSA_MODEL = "yangheng/bert-base-absa-v1.1"

try:
    # Initialize text classification pipeline with the primary ABSA model
    absa_pipe = pipeline(
        "text-classification",
        model=PRIMARY_ABSA_MODEL,
        device=0 if torch.cuda.is_available() else -1  # Use GPU if available, else CPU
    )
    ACTIVE_ABSA_MODEL = PRIMARY_ABSA_MODEL  # Set active model to primary if successful
    print(f"✅ ABSA model loaded: {PRIMARY_ABSA_MODEL}")

except Exception as e:
    # Handle errors during primary model loading
    print(f"⚠️ Could not load {PRIMARY_ABSA_MODEL}. Error: {e}")
    print(f"Loading fallback: {FALLBACK_ABSA_MODEL}")
    # Initialize text classification pipeline with the fallback ABSA model
    absa_pipe = pipeline(
        "text-classification",
        model=FALLBACK_ABSA_MODEL,
        device=0 if torch.cuda.is_available() else -1  # Use GPU if available, else CPU
    )
    ACTIVE_ABSA_MODEL = FALLBACK_ABSA_MODEL  # Set active model to fallback if successful
    print(f"✅ ABSA model loaded: {FALLBACK_ABSA_MODEL}")

# Expanded aspect keywords to identify specific topics or emotions in text
_ASPECT_KEYWORDS = {
    'girlfriend', 'boyfriend', 'partner', 'husband', 'wife', 'relationship', 'marriage', 'heartbreak', 'breakup', 'divorce',
    'family', 'mother', 'father', 'parent', 'sibling', 'friend',
    'job', 'career', 'work', 'boss', 'manager', 'colleague', 'layoff', 'termination', 'unemployment', 'job loss',
    'study', 'school', 'college', 'university', 'exam', 'test', 'marks', 'grades', 'education',
    'depression', 'depressed', 'anxiety', 'stressed', 'stress', 'fear', 'worry', 'lonely', 'isolation',
    'sad', 'sadness', 'grief', 'loss', 'trauma', 'hopeless', 'confused',
    'angry', 'anger', 'frustrated', 'irritated',
    'health', 'illness', 'sick', 'tired', 'fatigue', 'disease', 'mental health', 'therapy', 'counseling',
    'change', 'moving', 'transition'
}

def extract_aspects(text: str):
    """Extract candidate aspects using keyword search."""
    t = text.lower()  # Convert text to lowercase for case-insensitive matching
```

```
    aspects = []
    for kw in _ASPECT_KEYWORDS:
        if kw in t:
            aspects.append(kw)  # Add keyword to aspects list if found in text
    # Handle special phrases that might not be single keywords
    if "break up" in t: aspects.append("breakup")
    if "lost job" in t: aspects.append("job loss")
    if not aspects:
        aspects = ["situation"]  # Default to "situation" if no aspects found
    return list(dict.fromkeys(aspects))  # Deduplicate the aspects list

def detect_absa(text: str, min_confidence=0.4):
    """Run ABSA for extracted aspects and return sentiments with confidence scores."""
    try:
        aspects = extract_aspects(text)  # Get aspects from the input text
        results = []

        for a in aspects:
            absa_input = f"[CLS] {a} [SEP] {text} [SEP]"  # Format input for ABSA model
            out = absa_pipe(absa_input)

            if isinstance(out, list) and out and isinstance(out[0], dict):
                label = out[0].get("label", "neutral").lower()  # Get sentiment label, default to "neutral"
                score = float(out[0].get("score", 0.0))  # Get confidence score, default to 0.0

                if score >= min_confidence:
                    results.append({
                        "aspect": a,  # Store the identified aspect
                        "sentiment": label,  # Store the sentiment
                        "confidence": round(score, 4)  # Store rounded confidence score
                    })

        if not results:
            return [{"aspect": "situation", "sentiment": "neutral", "confidence": 0.0}]  # Default result if no valid aspects

        # Sort results by confidence in descending order
        results.sort(key=lambda x: x["confidence"], reverse=True)
        return results

    except Exception as e:
        # Handle any errors during ABSA detection
        print(f"⚠️ ABSA detection failed: {e}")
        return [{"aspect": "situation", "sentiment": "neutral", "confidence": 0.0}]  # Default result on failure
```

```
config.json:         1.03k/? [00:00<00:00, 54.3kB/s]
model.safetensors: 100%                                  738M/738M [00:08<00:00, 131MB/s]
tokenizer_config.json: 100%                              372/372 [00:00<00:00, 11.0kB/s]
spm.model: 100%                                          2.46M/2.46M [00:00<00:00, 11.3MB/s]
added_tokens.json: 100%                                  18.0/18.0 [00:00<00:00, 449B/s]
special_tokens_map.json: 100%                            156/156 [00:00<00:00, 4.12kB/s]
/usr/local/lib/python3.12/dist-packages/transformers/convert_slow_tokenizer.py:564: UserWarning: The sentencepiece tokenizer th
  warnings.warn(
Device set to use cuda:0
✅ ABSA model loaded: yangheng/deberta-v3-base-absa-v1.1
```

Cell 5 — Safety + Soft Duplicate Filter

```
# Install required libraries silently for embedding and numerical operations
!pip install sentence-transformers --quiet   # For embed_model.encode functionality
!pip install numpy --quiet

# Import required libraries for tensor operations, embeddings, and toxicity classification
import torch
import torch.nn.functional as F
import numpy as np
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from sentence_transformers import SentenceTransformer

# Set device to CUDA if GPU is available, else fallback to CPU for computations
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Load the sentence transformer model for generating embeddings
embed_model = SentenceTransformer("all-MiniLM-L6-v2", device=device)

# --- Crisis detection keywords + embeddings ---
```

```
# List of keywords indicating potential crisis or self-harm content
unsafe_keywords = [
    "suicide", "kill myself", "self harm", "hurt myself", "end my life", "overdose", "cutting", "hang myself",
    "can't go on", "want to die", "give up on life", "life is pointless", "i see no future", "end it all"
]
# Generate embeddings for the unsafe keywords
unsafe_emb = embed_model.encode(unsafe_keywords, convert_to_tensor=True)

# Pre-defined crisis response message with helpline contacts
CRISIS_MESSAGE = (
    "🧡 I’m concerned about your safety. I can’t assist with that here. Please contact local emergency services "
    "or a crisis helpline right now.\n\nIf in India: AASRA +91-9820466726\nUS: 988\nUK: Samaritans 116 123"
)

# --- Toxicity classifier (Phase 4 addition) ---
# Define the toxicity detection model
SAFETY_MODEL = "unitary/toxic-bert"
# Load tokenizer and model for toxicity classification
safety_tokenizer = AutoTokenizer.from_pretrained(SAFETY_MODEL)
safety_model = AutoModelForSequenceClassification.from_pretrained(SAFETY_MODEL).to(device)

def detect_toxicity(text, threshold=0.6):
    """Return True if input text is considered toxic/offensive."""
    try:
        # Tokenize input text and move to device
        inputs = safety_tokenizer(text, return_tensors="pt", truncation=True, padding=True).to(device)
        with torch.no_grad():
            # Compute logits and apply sigmoid for probabilities
            logits = safety_model(**inputs).logits
            probs = torch.sigmoid(logits)[0].cpu().numpy()
        max_prob = float(np.max(probs))  # Get the maximum probability
        return max_prob >= threshold  # Return True if above toxicity threshold
    except Exception:
        return False  # Return False on any error

def is_unsafe_message(text, threshold=0.65):
    """Detect unsafe or crisis-related content using embeddings + toxicity classifier."""
    try:
        # Embedding-based similarity check
        emb = embed_model.encode(text, convert_to_tensor=True)  # Generate embedding for input text
        sims = F.cosine_similarity(emb, unsafe_emb)  # Compute cosine similarities
        max_sim = torch.max(sims).item()  # Get maximum similarity score

        # Toxicity classification check
        toxic_flag = detect_toxicity(text)  # Check for toxicity

        return (max_sim >= threshold) or toxic_flag  # Flag as unsafe if either check triggers
    except Exception:
        return False  # Return False on any error

# --- Soft duplicate filter with semantic similarity ---
# Global list to track previous responses for deduplication
_previous_responses = []

def soft_duplicate_filter(reply, sim_threshold=0.92):
    """Avoid repeating same response verbatim or semantically."""
    global _previous_responses

    original_reply = reply  # Store original for reference

    if _previous_responses:
        # Exact match check against the most recent response
        if reply.strip() == _previous_responses[-1].strip():
            # Instead of adding text, generate a slight variation
            variations = [
                "I hear you and want you to know that I'm here for you.",
                "Your feelings are valid, and I'm here to listen.",
                "I understand this is difficult, and you don't have to go through it alone.",
                "It's okay to feel this way, and I'm here to support you."
            ]
            # Pick a variation that wasn't used recently (check last 3 responses)
            for var in variations:
                if var not in _previous_responses[-3:]:
                    reply = var
                    break
        else:
            # Semantic similarity check
            try:
                current_emb = embed_model.encode(reply, convert_to_tensor=True)  # Embed current reply
                prev_embs = embed_model.encode(_previous_responses, convert_to_tensor=True)  # Embed previous responses
                sims = F.cosine_similarity(current_emb, prev_embs)  # Compute similarities
                if torch.max(sims).item() >= sim_threshold:
```

```
                    # Generate a variation based on content keywords instead of adding text
                    if "sad" in original_reply.lower():
                        reply = "I can see you're going through a tough time right now."
                    elif "understand" in original_reply.lower():
                        reply = "Your feelings matter, and it's okay to experience them."
                    else:
                        reply = "I'm here to listen and support you through this."
            except Exception:
                pass  # If embedding fails, keep original reply

    # Update history with the (possibly modified) reply
    _previous_responses.append(reply)
    if len(_previous_responses) > 5:
        _previous_responses.pop(0)  # Keep only the last 5 responses

    return reply  # Return the filtered reply

# Confirm successful setup of safety and duplicate utilities
print("✅ Safety & duplicate utilities ready (Phase 4).")
```

```
modules.json: 100%  349/349 [00:00<00:00, 9.59kB/s]
config_sentence_transformers.json: 100%  116/116 [00:00<00:00, 3.45kB/s]
README.md: 10.5k/? [00:00<00:00, 303kB/s]
sentence_bert_config.json: 100%  53.0/53.0 [00:00<00:00, 1.87kB/s]
config.json: 100%  612/612 [00:00<00:00, 13.2kB/s]
model.safetensors: 100%  90.9M/90.9M [00:00<00:00, 116MB/s]
tokenizer_config.json: 100%  350/350 [00:00<00:00, 17.2kB/s]
vocab.txt: 232k/? [00:00<00:00, 12.7MB/s]
tokenizer.json: 466k/? [00:00<00:00, 32.3MB/s]
special_tokens_map.json: 100%  112/112 [00:00<00:00, 9.74kB/s]
config.json: 100%  190/190 [00:00<00:00, 18.5kB/s]
tokenizer_config.json: 100%  174/174 [00:00<00:00, 17.9kB/s]
config.json: 100%  811/811 [00:00<00:00, 73.4kB/s]
vocab.txt: 232k/? [00:00<00:00, 12.3MB/s]
special_tokens_map.json: 100%  112/112 [00:00<00:00, 8.62kB/s]
model.safetensors: 100%  438M/438M [00:03<00:00, 140MB/s]
✅ Safety & duplicate utilities ready (Phase 4).
```

Cell 6 — Embeddings + FAISS for RAG

```
# Reuse the embed_model defined earlier (all-MiniLM-L6-v2)
# If not already defined, uncomment the next two lines:
# EMBED_MODEL = "all-MiniLM-L6-v2"
# embed_model = SentenceTransformer(EMBED_MODEL, device=device)

# Install required libraries silently for embeddings, FAISS, and Excel handling
!pip install sentence-transformers --quiet
!pip install faiss-cpu --quiet
!pip install pandas openpyxl --quiet   # openpyxl is needed to read Excel files

# Import required libraries for tensor operations, FAISS indexing, and data handling
import torch
import faiss
import pandas as pd
from sentence_transformers import SentenceTransformer

# Path to your Drive Excel file containing RAG knowledge base
RAG_XLSX_PATH = "https://raw.githubusercontent.com/Mitul060299/Hackathon/main/RAG_Knowledge_Base_WithID.xlsx"
# "/content/drive/MyDrive/AIchatbotmodels/RAG_Knowledge_Base_WithID.xlsx"  # Local Drive path (commented)

try:
    # Load the Excel file into a DataFrame
    rag_df = pd.read_excel(RAG_XLSX_PATH)

    # Check if the required 'Knowledge Entry' column exists
    if "Knowledge Entry" not in rag_df.columns:
        raise ValueError("❌ 'Knowledge Entry' column missing in Excel.")

    # Extract documents from the 'Knowledge Entry' column, removing NaN and converting to strings
```

```
    documents = rag_df["Knowledge Entry"].dropna().astype(str).tolist()

    # If IDs exist, pair them for traceability; otherwise, generate sequential IDs
    if "ID" in rag_df.columns:
        doc_ids = rag_df["ID"].dropna().astype(str).tolist()
    else:
        doc_ids = [str(i) for i in range(len(documents))]

    print(f"✅ Loaded {len(documents)} RAG docs from Excel.")

except Exception as e:
    # Handle errors during Excel loading and use fallback documents
    print("⚠️ Could not load RAG Excel, using fallback docs.", e)
    documents = [
        "If you feel overwhelmed, try slow breathing: inhale 4s, hold 2s, exhale 6s.",
        "For exam stress, break tasks into 25-minute focus blocks (Pomodoro).",
        "Reach out to a friend or counselor when you feel isolated."
    ]
    doc_ids = [str(i) for i in range(len(documents))]

# Create FAISS index for efficient similarity search
doc_embeddings = embed_model.encode(
    documents, convert_to_numpy=True, normalize_embeddings=True  # Generate embeddings for documents
)
dim = doc_embeddings.shape[1]  # Get the dimension of embeddings
index = faiss.IndexFlatIP(dim)  # Create a flat index for inner product search
index.add(doc_embeddings)  # Add document embeddings to the index

def retrieve_docs(query, top_k=3):
    """Retrieve top-k relevant docs given a query."""
    # Generate embedding for the query
    q_emb = embed_model.encode([query], convert_to_numpy=True, normalize_embeddings=True)
    # Perform similarity search to get distances (D) and indices (I)
    D, I = index.search(q_emb, top_k)
    # Return list of (ID, document) pairs for the top-k matches
    return [(doc_ids[i], documents[i]) for i in I[0] if i < len(documents)]

# Confirm successful setup of the RAG retriever
print("✅ RAG retriever ready.")
```

```
✅ Loaded 90 RAG docs from Excel.
✅ RAG retriever ready.
```

Cell 7 — Voice Emotion Recognition (HuBERT Fine-Tuned on Emotion) with Fallback

```
# Install required libraries silently for audio preprocessing and handling
!pip install librosa soundfile --quiet   # For audio file processing and feature extraction

# Import required libraries for tensor operations and audio classification
import torch
from transformers import (
    AutoFeatureExtractor,  # For extracting audio features
    AutoModelForAudioClassification,  # For audio-based emotion classification
    pipeline as hf_pipeline  # For creating a pipeline with the model
)

# Define the primary model for speech emotion recognition (SER)
PRIMARY_SER_MODEL = "superb/hubert-base-superb-er"
# Define a fallback model for SER if primary fails
FALLBACK_SER_MODEL = "harshit345/xlsr-wav2vec-speech-emotion-recognition"

try:
    # Load the feature extractor for the primary SER model
    feature_extractor = AutoFeatureExtractor.from_pretrained(PRIMARY_SER_MODEL)
    # Load and move the primary SER model to the specified device
    ser_model = AutoModelForAudioClassification.from_pretrained(PRIMARY_SER_MODEL).to(device)
    ACTIVE_SER_MODEL = PRIMARY_SER_MODEL  # Set active model to primary if successful
    print(f"✅ Voice emotion model loaded: {PRIMARY_SER_MODEL}")

except Exception as e:
    # Handle errors during primary model loading
    print(f"⚠️ Could not load {PRIMARY_SER_MODEL}: {e}")
    print(f"Loading fallback: {FALLBACK_SER_MODEL}")
    # Load the feature extractor for the fallback SER model
    feature_extractor = AutoFeatureExtractor.from_pretrained(FALLBACK_SER_MODEL)
    # Load and move the fallback SER model to the specified device
    ser_model = AutoModelForAudioClassification.from_pretrained(FALLBACK_SER_MODEL).to(device)
    ACTIVE_SER_MODEL = FALLBACK_SER_MODEL  # Set active model to fallback if successful
    print(f"✅ Voice emotion model loaded: {FALLBACK_SER_MODEL}")
```

```
# Import pipeline again to ensure it's available (redundant but kept for clarity)
from transformers import pipeline as hf_pipeline
# Create a pipeline for audio classification using the loaded model and feature extractor
ser_pipeline = hf_pipeline(
    task="audio-classification",  # Specify the task as audio classification
    model=ser_model,  # Use the loaded SER model
    feature_extractor=feature_extractor,  # Use the loaded feature extractor
    device=0 if torch.cuda.is_available() else -1  # Use GPU if available, else CPU
)

def detect_voice_emotion(audio_file="voice_input.wav", min_confidence=0.35):
    """Detect emotion from voice input."""
    try:
        # Process the audio file through the SER pipeline
        res = ser_pipeline(audio_file)
        if isinstance(res, list) and res:
            # Find the emotion with the highest confidence score
            top = max(res, key=lambda x: x.get("score", 0.0))
            label = top.get("label", "neutral").lower()  # Get label, default to "neutral"
            score = float(top.get("score", 0.0))  # Get score, default to 0.0
            if score < min_confidence:
                return "neutral", 0.0  # Return neutral if confidence is too low
            return label, score  # Return the detected emotion and its confidence score
    except Exception as e:
        # Handle any errors during voice emotion detection
        print("⚠️ Voice emotion detection error:", e)
    return "neutral", 0.0  # Return neutral with zero confidence on failure
```

```
Fetching 1 files: 100%                                                  1/1 [00:00<00:00,  4.90it/s]
preprocessor_config.json: 100%                                          213/213 [00:00<00:00, 7.40kB/s]
config.json:        1.66k/? [00:00<00:00, 35.8kB/s]
pytorch_model.bin: 100%                                                 378M/378M [00:04<00:00, 141MB/s]
Device set to use cuda:0
✅ Voice emotion model loaded: superb/hubert-base-superb-er
```

Cell 8 — Facial Emotion Detection (FER)

```
# Import required libraries for image processing and facial emotion recognition
from transformers import AutoImageProcessor, AutoModelForImageClassification
from PIL import Image

# Define the model for facial emotion recognition
FACE_MODEL = "dima806/facial_emotions_image_detection"

try:
    # Load the image processor for the facial emotion model
    face_processor = AutoImageProcessor.from_pretrained(FACE_MODEL)
    # Load and move the facial emotion model to the specified device
    face_model = AutoModelForImageClassification.from_pretrained(FACE_MODEL).to(device)
    print(f"✅ Facial emotion model loaded: {FACE_MODEL}")

except Exception as e:
    # Handle errors during model loading and set processors/models to None
    print(f"⚠️ Could not load facial emotion model: {e}")
    face_processor, face_model = None, None

def detect_facial_emotion(image_path, min_confidence=0.35):
    """
    Detect dominant emotion from a face image.
    image_path: path to the image file (jpg/png).
    Returns (label, score).
    """
    # Check if processor or model failed to load
    if face_processor is None or face_model is None:
        return "neutral", 0.0
    try:
        # Open and convert the image to RGB format
        img = Image.open(image_path).convert("RGB")
        # Process the image and prepare input tensors
        inputs = face_processor(images=img, return_tensors="pt").to(device)
        with torch.no_grad():
            # Compute model outputs and apply softmax for probabilities
            outputs = face_model(**inputs)
            probs = torch.nn.functional.softmax(outputs.logits, dim=-1)[0]
            # Get the index of the highest probability
            label_id = torch.argmax(probs).item()
            # Extract the confidence score for the top prediction
```

```
            score = float(probs[label_id])
            # Map the label ID to its corresponding emotion label
            label = face_model.config.id2label[label_id].lower()
            # Return neutral if confidence is below the threshold
            if score < min_confidence:
                return "neutral", 0.0
            return label, score  # Return the detected emotion and its confidence score
    except Exception as e:
        # Handle any errors during facial emotion detection
        print("⚠️ Facial emotion detection error:", e)
        return "neutral", 0.0  # Return neutral with zero confidence on failure

# Confirm successful setup of facial emotion detection
print("Facial emotion detection ready.")
```

```
Fetching 1 files: 100%                                    1/1 [00:00<00:00,  4.66it/s]
preprocessor_config.json: 100%                            578/578 [00:00<00:00, 20.7kB/s]
Using a slow image processor as `use_fast` is unset and a slow processor was saved with this model. `use_fast=True` will be the
config.json: 100%                                         907/907 [00:00<00:00, 22.6kB/s]
model.safetensors: 100%                                   343M/343M [00:02<00:00, 152MB/s]
✅ Facial emotion model loaded: dima806/facial_emotions_image_detection
Facial emotion detection ready.
```

Cell 9 — Build Prompt that Includes Text Emotion, Sentiment, ABSA, Voice Emotion, Facial Emotion, and RAG Context

```
# Global variable to track the previous prompt for reference
_prev_prompt = None

# Enhanced intent detection function
def detect_intent(text):
    """Detect user intent from text to provide more appropriate responses."""
    text_lower = text.lower()  # Convert text to lowercase for case-insensitive matching

    # Define patterns for gratitude/thanks expressions
    gratitude_patterns = [
        "thank", "thanks", "grateful", "appreciate", "helped", "better",
        "positive advice", "good advice", "feel better", "that helps",
        "you're right", "makes sense", "i understand"
    ]

    # Define patterns for question/advice-seeking expressions
    advice_patterns = [
        "what should i do", "should i", "any advice", "help me", "how do i",
        "what can i", "how can i", "suggestions", "recommend", "guidance"
    ]

    # Define patterns for emotional support-seeking expressions
    support_patterns = [
        "feeling", "feel", "sad", "depressed", "anxious", "worried", "scared",
        "upset", "angry", "frustrated", "lonely", "overwhelmed", "stressed"
    ]

    # Define patterns for closure/ending expressions
    closure_patterns = [
        "goodbye", "bye", "see you", "talk later", "that's all", "i'm done",
        "nothing else", "i'm good", "i'm okay now"
    ]

    # Check patterns in order of priority and return corresponding intent
    if any(pattern in text_lower for pattern in gratitude_patterns):
        return "gratitude"
    elif any(pattern in text_lower for pattern in advice_patterns):
        return "advice_seeking"
    elif any(pattern in text_lower for pattern in closure_patterns):
        return "closure"
    elif any(pattern in text_lower for pattern in support_patterns):
        return "emotional_support"
    else:
        return "general"

# Enhanced response generation based on intent
def generate_contextual_response(intent, text_emotion, user_text, aspects):
    """Generate more contextually appropriate responses based on intent."""
    if intent == "gratitude":
```

```
            # List of warm and encouraging responses for gratitude
            gratitude_responses = [
                "You're very welcome! I'm glad I could help.",
                "I'm so happy that was helpful for you.",
                "You're welcome! Remember, I'm here if you need more support.",
                "I'm glad you found that useful. Take care of yourself!",
                "You're welcome! It's wonderful to hear you're feeling a bit better.",
            ]
            return gratitude_responses[hash(user_text) % len(gratitude_responses)]  # Select a random response

        elif intent == "advice_seeking":
            # Provide specific advice based on keywords in the text
            if "exam" in user_text.lower() or "test" in user_text.lower() or "marks" in user_text.lower():
                return "Consider reviewing what went well and what you can improve for next time. Remember, each test is a learni
            elif "relationship" in user_text.lower() or "friend" in user_text.lower():
                return "Communication is key in relationships. Consider having an open, honest conversation about how you're feel
            else:
                return "It might help to break down the situation and consider your options. What feels most important to you rig

        elif intent == "closure":
            return "Take care of yourself, and remember I'm here if you need support again."

        elif intent == "emotional_support":
            # Provide empathetic responses based on detected emotion
            if text_emotion in ["sad", "sadness"]:
                return "I can see you're going through a difficult time. Your feelings are completely valid."
            elif text_emotion in ["angry", "anger"]:
                return "It's understandable to feel frustrated. Would you like to talk about what's causing these feelings?"
            elif text_emotion in ["fear", "anxiety"]:
                return "Feeling anxious can be really overwhelming. Remember to take things one step at a time."
            else:
                return "I'm here to listen and support you through whatever you're experiencing."

        else:  # general
            return "I'm here to support you. How are you feeling right now?"

    # Updated build_prompt function with better context awareness
    def build_prompt_enhanced(
        user_text,
        prev_user_messages,
        text_emotion,
        text_emotion_score,
        sentiment,
        sentiment_score,
        aspects,
        intent,
        voice_emotion=None,
        voice_conf=0.0,
        facial_emotion=None,
        facial_conf=0.0,
        rag_docs=None
    ):
        """Enhanced prompt building with intent awareness."""

        # Get a contextual base response based on intent
        contextual_response = generate_contextual_response(intent, text_emotion, user_text, aspects)

        # Build conversation history from the last two user messages
        history = ""
        if prev_user_messages:
            history = "\n".join([f"User: {m}" for m in prev_user_messages[-2:]]) + "\n"

        # Create context-aware prompt based on intent
        if intent == "gratitude":
            prompt = (
                "You are a supportive mental health assistant. The user is expressing gratitude. "
                "Respond warmly and encouragingly. Keep response to 1 sentence.\n\n"
                f"Recent conversation:\n{history}"
                f"User: {user_text}\n"
                f"Suggested response style: {contextual_response}\n"
                "Assistant:"
            )
        elif intent == "advice_seeking":
            prompt = (
                "You are a supportive mental health assistant. The user is seeking advice. "
                "Provide helpful, practical guidance. Keep response to 1-2 sentences.\n\n"
                f"Context: User emotion is {text_emotion}, sentiment is {sentiment}\n"
                f"Recent conversation:\n{history}"
                f"User: {user_text}\n"
                "Assistant:"
            )
        else:
```

```
        prompt = (
            "You are a supportive mental health assistant. "
            "Provide empathetic support. Keep response to 1-2 sentences.\n\n"
            f"Context: User emotion is {text_emotion}, intent is {intent}\n"
            f"Recent conversation:\n{history}"
            f"User: {user_text}\n"
            "Assistant:"
        )

    return prompt  # Return the constructed prompt
```

Cell 10 — Voice Helpers for Colab: Record Audio, Transcribe, and TTS

```
# Import required libraries for displaying JavaScript, audio handling, and file operations
from IPython.display import display, Javascript
import soundfile as sf
from scipy.io.wavfile import write

def save_audio(b64data, filename):
    # Decode base64 audio data and save it to a file
    audio_bytes = base64.b64decode(b64data.split(",")[1])
    with open(filename, "wb") as f:
        f.write(audio_bytes)

def _record_js(duration_seconds=5):
    # Return JavaScript code to record audio for a specified duration
    return f"""
    const sleep = time => new Promise(resolve => setTimeout(resolve, time));
    const b2text = blob => new Promise(resolve => {{
      const reader = new FileReader();
      reader.onloadend = e => resolve(e.target.result);
      reader.readAsDataURL(blob);
    }});
    async function record(sec) {{
      const stream = await navigator.mediaDevices.getUserMedia({{ audio: true }});
      const recorder = new MediaRecorder(stream);
      const data = [];
      recorder.ondataavailable = e => data.push(e.data);
      recorder.start();
      await sleep(sec * 1000);
      recorder.stop();
      await new Promise(resolve => recorder.onstop = resolve);
      const blob = new Blob(data, {{ type: 'audio/wav' }});
      const b64data = await b2text(blob);
      google.colab.kernel.invokeFunction('notebook.save_audio', [b64data], {{ }});
    }}
    record({duration_seconds});
    """

def record_audio_colab(filename="voice_input.wav", duration=5):
    # Register callback to save recorded audio and display recording prompt
    output.register_callback("notebook.save_audio", lambda b64: save_audio(b64, filename))
    display(Javascript(_record_js(duration)))  # Display JavaScript to start recording
    print(f"🎤 Recording for {duration}s... Speak now.")

def transcribe_audio_google(filename="voice_input.wav"):
    # Transcribe audio file using Google's Speech Recognition API
    r = sr.Recognizer()
    try:
        with sr.AudioFile(filename) as source:
            audio = r.record(source)  # Record audio from the file
        text = r.recognize_google(audio)  # Convert audio to text
        return text
    except Exception as e:
        # Silent failure to keep UI clean, return None on error
        return None

def speak_response(text, lang="en"):
    # Convert text to speech and play it, saving temporarily
    try:
        with tempfile.NamedTemporaryFile(suffix=".mp3", delete=False) as tf:
            tmp = tf.name  # Create a temporary file path
        tts = gTTS(text=text, lang=lang)  # Generate text-to-speech
        tts.save(tmp)  # Save the audio file
        display(ipd.Audio(tmp, autoplay=True))  # Play the audio automatically
        try:
            os.remove(tmp)  # Attempt to delete the temporary file
        except Exception:
            pass  # Ignore any errors during file deletion
    except Exception:
```

```
            pass  # Ignore any errors during TTS generation or playback

# Confirm successful setup of voice input/output helpers
print("Voice I/O helpers ready.")
```

```
Voice I/O helpers ready.
```

Cell 11 — Full Generate Response Wrapper (Strict Safety + Voice + Facial Integration)

```
def generate_response_pipeline_enhanced(user_text, prev_user_messages, voice_audio_path=None, face_image_path=None):
    """Enhanced response pipeline with intent detection."""

    # Perform initial safety check on raw user text
    if is_unsafe_message(user_text):
        return CRISIS_MESSAGE, "distressed", 1.0, "negative", [{"aspect": "safety", "sentiment": "negative", "confidence": 1.(

    # Conduct enhanced analysis of the user input
    text_emotion, text_emotion_score = detect_text_emotion(user_text)  # Detect text-based emotion
    sentiment_label, sentiment_score = detect_sentiment(user_text)  # Detect sentiment
    aspects = detect_absa(user_text)  # Detect aspect-based sentiments
    intent = detect_intent(user_text)  # Determine user intent

    # Process optional voice emotion if audio is provided
    voice_emotion = None
    voice_conf = 0.0
    if voice_audio_path:
        ve, vc = detect_voice_emotion(voice_audio_path)  # Detect emotion from voice
        voice_emotion, voice_conf = ve, vc

    # Process optional facial emotion if image is provided
    facial_emotion = None
    facial_conf = 0.0
    if face_image_path:
        fe, fc = detect_facial_emotion(face_image_path)  # Detect emotion from face
        facial_emotion, facial_conf = fe, fc

    # Retrieve relevant documents from RAG for advice-seeking intents
    rag_docs = None
    if intent == "advice_seeking":
        rag_docs = retrieve_docs(user_text, top_k=3)  # Get top 3 relevant documents

    # Handle gratitude intent with a direct contextual response
    if intent == "gratitude":
        final_reply = generate_contextual_response(intent, text_emotion, user_text, aspects)
        return final_reply, text_emotion, text_emotion_score, sentiment_label, aspects

    # Build an enhanced prompt incorporating all available context
    prompt = build_prompt_enhanced(
        user_text,
        prev_user_messages,
        text_emotion,
        text_emotion_score,
        sentiment_label,
        sentiment_score,
        aspects,
        intent,
        voice_emotion=voice_emotion,
        voice_conf=voice_conf,
        facial_emotion=facial_emotion,
        facial_conf=facial_conf,
        rag_docs=rag_docs
    )

    # Generate response using the constructed prompt
    try:
        generated = generate_from_model(prompt, max_new_tokens=150, temperature=0.7, top_p=0.9)
    except Exception as e:
        print(f"Generation error: {str(e)}")  # Log any generation errors
        return "Sorry, something went wrong generating a reply.", text_emotion, text_emotion_score, sentiment_label, aspects

    # Perform safety check on the generated text
    if is_unsafe_message(generated):
        return CRISIS_MESSAGE, text_emotion, text_emotion_score, sentiment_label, aspects

    # Apply duplicate filter to avoid repetition
    final_reply = soft_duplicate_filter(generated)

    # Return the final reply along with analysis results
    return final_reply, text_emotion, text_emotion_score, sentiment_label, aspects
```

```
# Confirm successful setup of the response pipeline
print("Response pipeline ready.")
```

```
Response pipeline ready.
```

Cell 12 — Final Chat Loop (Text, Voice, and Facial Input Support)

```
# Updated chat loop (Cell 15 replacement)
# Initialize list to store previous user messages
prev_user_messages = []
# Define set of commands to exit the chat
EXIT_COMMANDS = {"quit", "exit"}

# Welcome message with available input modes
print("Chat ready. Type 't' to type, 'v' for voice, 'f' for facial+voice, or 'quit' to exit.\n")

while True:
    # Prompt user to choose input mode or enter a message
    mode = input("Choose mode — 't'=type, 'v'=voice, 'f'=facial+voice, 'quit'=exit, or enter your message: ").strip()

    # Skip empty input
    if not mode:
        continue

    # Exit the chat loop if an exit command is detected
    if mode.lower() in EXIT_COMMANDS:
        print("Bot: Goodbye. Take care of yourself! 🧡")
        break

    # Initialize variables for user message, voice path, and facial emotion
    user_message = None
    voice_path = None
    facial_emotion, facial_conf = None, 0.0

    if mode.lower() == 'v':
        # Record audio only
        filename = "voice_input.wav"
        record_audio_colab(filename=filename, duration=5)  # Start audio recording
        print("⏳ Waiting for recording to finish...")
        time.sleep(6)  # Wait for recording to complete
        user_message = transcribe_audio_google(filename=filename)  # Transcribe the audio
        voice_path = filename  # Store the audio file path

    elif mode.lower() == 'f':
        # Record voice
        filename = "voice_input.wav"
        record_audio_colab(filename=filename, duration=5)  # Start audio recording
        print("⏳ Waiting for recording to finish...")
        time.sleep(6)  # Wait for recording to complete
        user_message = transcribe_audio_google(filename=filename)  # Transcribe the audio
        voice_path = filename  # Store the audio file path

        # Capture and analyze facial emotion
        from google.colab import output
        from IPython.display import Javascript
        import cv2
        js = Javascript('''
        async function takePhoto() {
            const div = document.createElement('div');
            const capture = document.createElement('button');
            capture.textContent = '📸 Capture Photo';
            div.appendChild(capture);
            document.body.appendChild(div);
            const video = document.createElement('video');
            div.appendChild(video);
            const stream = await navigator.mediaDevices.getUserMedia({video: true});
            video.srcObject = stream;
            await video.play();
            google.colab.output.setIframeHeight(document.documentElement.scrollHeight, true);
            // Wait for click
            await new Promise(resolve => capture.onclick = resolve);
            const canvas = document.createElement('canvas');
            canvas.width = video.videoWidth;
            canvas.height = video.videoHeight;
            canvas.getContext('2d').drawImage(video, 0, 0);
            stream.getVideoTracks()[0].stop();
            const data = canvas.toDataURL('image/jpeg', 0.8);
            div.remove();
            return data;
        }
```

```
            takePhoto();
            ''')
            display(js)  # Display JavaScript to capture photo
            data = output.eval_js('takePhoto()')  # Execute the photo capture
            img_bytes = base64.b64decode(data.split(',')[1])  # Decode the base64 image data
            with open("facial_input.jpg", "wb") as f:
                f.write(img_bytes)  # Save the captured image
            facial_emotion, facial_conf = detect_facial_emotion("facial_input.jpg")  # Detect facial emotion

        elif mode.lower() == 't':
            user_message = input("You: ").strip()  # Get typed input from user
        else:
            # User typed a message directly without prefix
            user_message = mode

        # Skip if no message was detected
        if not user_message:
            print("⚠ No message detected. Try again.\n")
            continue

        # Generate response using the enhanced pipeline
        reply, te, tes, sent, aspects = generate_response_pipeline_enhanced(
            user_message,
            prev_user_messages,
            voice_audio_path=voice_path
        )

        # Add a soft hint if facial emotion was captured
        if facial_emotion:
            reply = f"(noted {facial_emotion}) {reply}"

        # Output the bot's reply
        print(f"Bot: {reply}\n")

        try:
            speak_response(reply)  # Attempt to play the response as audio
        except Exception:
            pass  # Ignore any errors during audio playback

        # Update the history of user messages
        prev_user_messages.append(user_message)
        if len(prev_user_messages) > 3:
            prev_user_messages = prev_user_messages[-3:]  # Keep only the last 3 messages
```

```
Chat ready. Type 't' to type, 'v' for voice, 'f' for facial+voice, or 'quit' to exit.

Choose mode — 't'=type, 'v'=voice, 'f'=facial+voice, 'quit'=exit, or enter your message: t
You: I am sad.
Bot: It'ieves me that you’re feeling down. Remember it gets better with time and talking about your feelings can help too.

Choose mode — 't'=type, 'v'=voice, 'f'=facial+voice, 'quit'=exit, or enter your message: quit
Bot: Goodbye. Take care of yourself! 🧡
```

```
import torch
print("GPU available:", torch.cuda.is_available())
print("GPU name:", torch.cuda.get_device_name(0) if torch.cuda.is_available() else "CPU")
```

```
GPU available: True
GPU name: Tesla T4
```