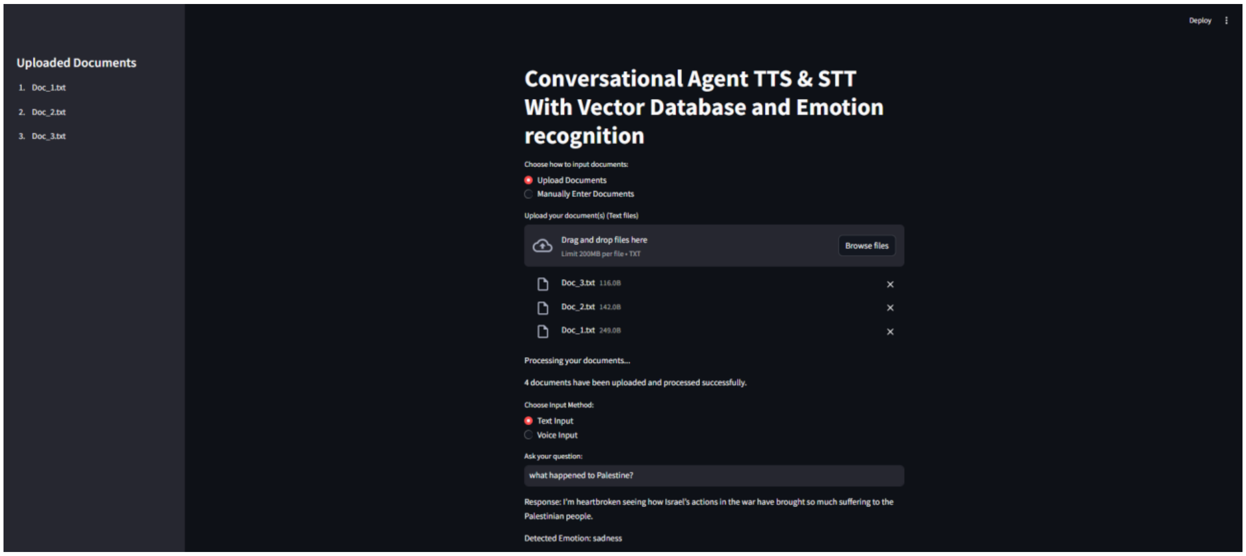
Documentation

1. Document Processing and Storage

My code begins with the setup for processing and storing documents using the **FAISS (Facebook AI Similarity Search)** vector database and **Sentence-BERT embeddings**:

- **Document Upload**: The code allows users to upload text files. These uploaded files are read and processed, with each line being treated as a separate document.
- **Manual Document Input**: It also provides the option for users to manually enter documents in a text area.



Uploaded Documents

1. Doc_1.txt
2. Doc_2.txt
3. Doc_3.txt

Conversational Agent TTS & STT With Vector Database and Emotion recognition

Choose how to input documents:

● Upload Documents
○ Manually Enter Documents

Upload your document(s) (Text files)

Drag and drop files here
Limit 200MB per file • TXT

Doc_3.txt 116.0B ×
Doc_2.txt 142.0B ×
Doc_1.txt 249.0B ×

Processing your documents...

4 documents have been uploaded and processed successfully.

Choose Input Method:

● Text Input
○ Voice Input

Ask your question:

what happened to Palestine?

Response: I'm heartbroken seeing how Israel's actions in the war have brought so much suffering to the Palestinian people.

Detected Emotion: sadness

- **Sentence Embeddings**: The uploaded or entered documents are encoded into **384-dimensional embeddings** using the `SentenceTransformer` model (`'all-MiniLM-L6-v2'`), which is efficient and works well for sentence-level tasks.
- **FAISS Indexing**: The generated embeddings are added to a **FAISS index**. FAISS allows for efficient retrieval of similar documents based on vector-based similarity. The index is created using `faiss.IndexFlatL2` with L2 (Euclidean) distance.

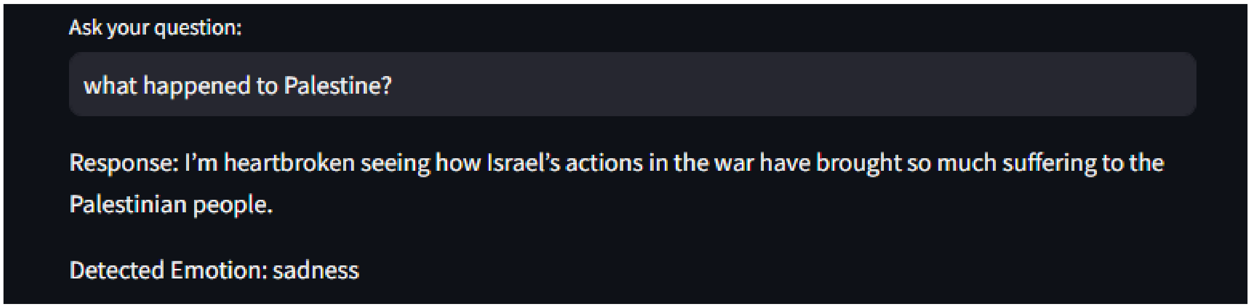
2. Query-to-Response System

- **Query Processing**: When a user inputs a query (either via text or voice), the code first generates an embedding of the query using the same Sentence-BERT model.
- **Document Retrieval**: The generated query embedding is then compared to the stored document embeddings in the FAISS index. The index performs a **nearest neighbor search** to find the most similar document to the query, which is returned as the response.

- **Response**: The most similar document retrieved from the FAISS index is used as the response to the user's query.

3. Emotion Detection

- **Emotion Analysis**: Once a response is generated, an **emotion detection model** (`DistilRoBERTa` model fine-tuned for emotion classification) is applied to the response text to detect the emotional tone.
- The emotion is classified into categories such as happiness, sadness, anger, etc., based on the content of the response.



Ask your question:

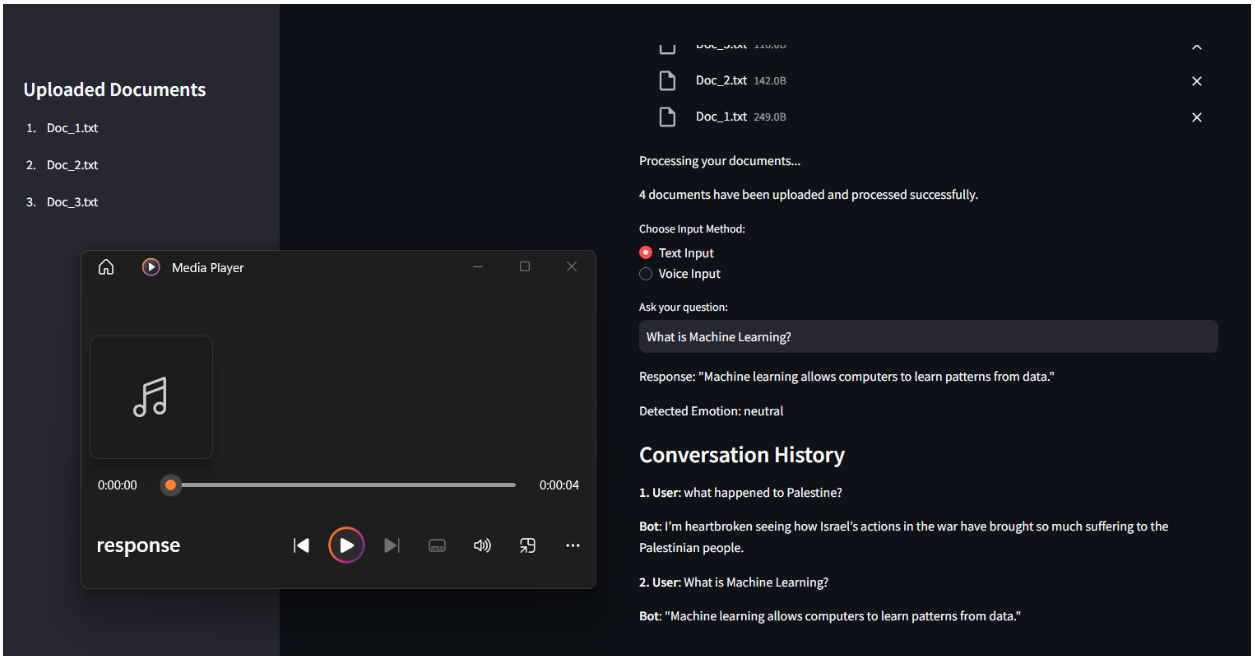
what happened to Palestine?

Response: I’m heartbroken seeing how Israel’s actions in the war have brought so much suffering to the Palestinian people.

Detected Emotion: sadness

4. Text-to-Speech (TTS)

- **Speech Synthesis**: The response text (either retrieved from documents or generated through the query) is converted to speech using the **Google Text-to-Speech (gTTS)** library.
- **Audio Playback**: The generated speech is saved as an `.mp3` file and played using the system's default audio player. On Windows, it uses the `os.system("start response.mp3")` command to play the speech output.



Uploaded Documents

1. Doc_1.txt
2. Doc_2.txt
3. Doc_3.txt

Doc_3.txt

Doc_2.txt 142.0B

Doc_1.txt 249.0B

Processing your documents...

4 documents have been uploaded and processed successfully.

Choose Input Method:

● Text Input

○ Voice Input

Ask your question:

What is Machine Learning?

Response: "Machine learning allows computers to learn patterns from data."

Detected Emotion: neutral

Conversation History

1. User: what happened to Palestine?

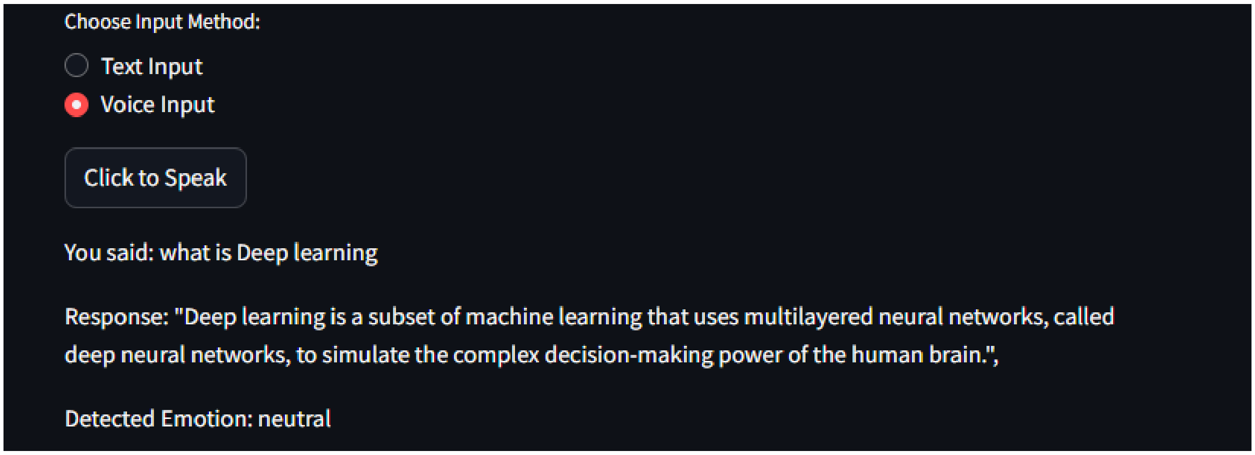
Bot: I'm heartbroken seeing how Israel's actions in the war have brought so much suffering to the Palestinian people.

2. User: What is Machine Learning?

Bot: "Machine learning allows computers to learn patterns from data."

5. Speech-to-Text (STT)

- **Speech Recognition**: The system allows users to ask questions using their voice. The **SpeechRecognition library** (with Google Speech API) is used to recognize spoken words and convert them into text.
- **Speech Input Handling**: If the user chooses "Voice Input" and clicks a button, the system listens for the user's speech, converts it to text, and processes the query in the same way as text input.



Choose Input Method:

○ Text Input

● Voice Input

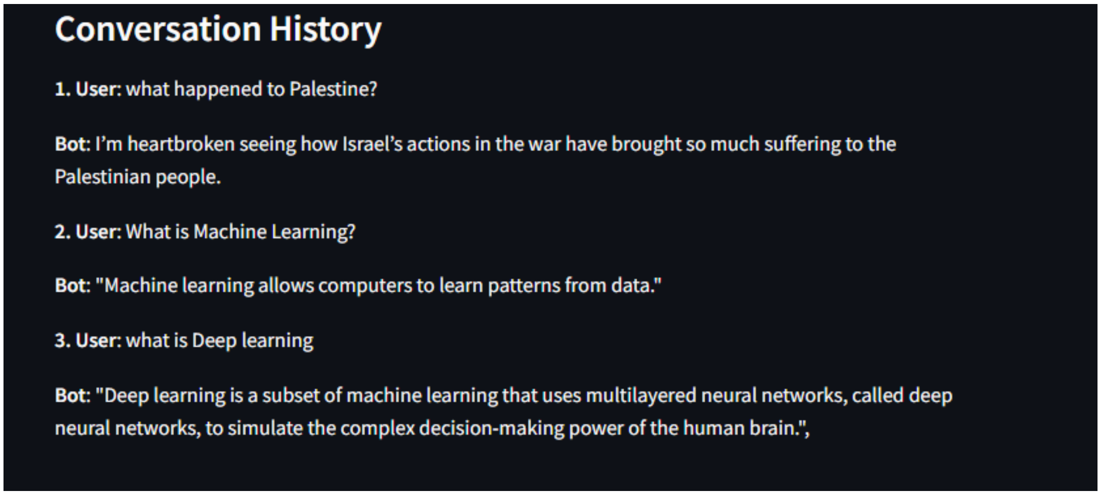
You said: what is Deep learning

Response: "Deep learning is a subset of machine learning that uses multilayered neural networks, called deep neural networks, to simulate the complex decision-making power of the human brain.",

Detected Emotion: neutral

6. Conversation History

- **Session Tracking**: The conversation history is stored in **`st.session_state`**, allowing the system to track past interactions within the current session. This enables context-aware responses for ongoing conversations.
- **Storing Conversations**: Each query and its corresponding response (along with their embeddings) are stored in `session_state.conversation_history`. This makes it possible to maintain continuity in multi-turn dialogues.
- **Embedding Conversation**: The embeddings of both the user query and the bot's response are added to the FAISS index to improve future query matching and response generation.



Conversation History

1. User: what happened to Palestine?

Bot: I'm heartbroken seeing how Israel's actions in the war have brought so much suffering to the Palestinian people.

2. User: What is Machine Learning?

Bot: "Machine learning allows computers to learn patterns from data."

3. User: what is Deep learning

Bot: "Deep learning is a subset of machine learning that uses multilayered neural networks, called deep neural networks, to simulate the complex decision-making power of the human brain.",

Key Features

- **Text and Voice Input**: The system supports both text and voice-based user inputs, making it more interactive.
- **Contextual Responses**: The system retrieves the most relevant document based on the user's query using FAISS and provides contextually relevant answers.
- **Emotion Detection**: The system detects and classifies emotions in the bot's responses, which could potentially inform how the agent modulates its tone (though not fully implemented in terms of voice modulation).
- **Voice Output**: The chatbot responses are converted into speech, enabling users to listen to answers rather than just read them.

Technologies Used

- **`streamlit`**: The front-end interface for the chatbot.
- **`sentence_transformers`**: For generating sentence embeddings used for document retrieval and query answering.
- **`faiss`**: For fast similarity search and efficient document retrieval based on vector embeddings.
- **`gTTS`**: For converting text responses into speech.
- **`speech_recognition`**: For converting voice inputs into text queries.
- **`transformers`**: For emotion classification using a pre-trained DistilRoBERTa model.
- **`os`**: For managing audio playback on the system.

Workflow

1. **Document Input**: User uploads text files or manually enters documents.
2. **Document Embedding**: Documents are encoded into embeddings and stored in a FAISS index.
3. **Query Input**: User inputs a query, either via text or voice.
4. **Query Processing**: The query is converted into an embedding, and the most similar document is retrieved from FAISS.
5. **Emotion Detection**: The bot's response is analyzed for emotional tone.
6. **Response Delivery**: The response is presented as text, the detected emotion is displayed, and the response is converted to speech.
7. **Conversation History**: All interactions are stored for context in the conversation history.

Strengths of the Current System

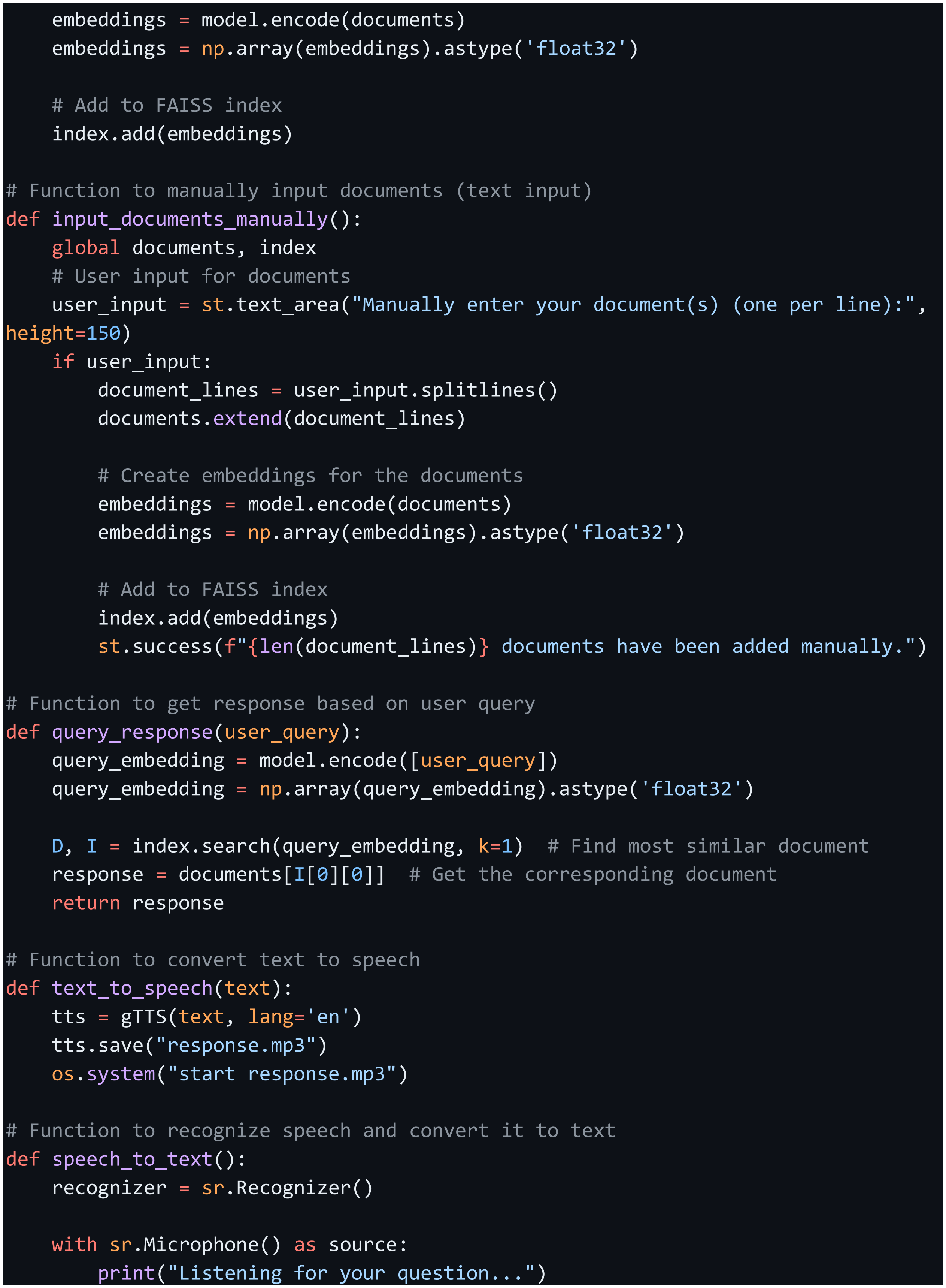
- **Multi-modal Interaction**: The chatbot can handle both text and voice inputs, providing flexibility for users.
- **Emotion-aware**: The system detects and displays emotions in the bot’s responses, which could be further used to adjust the bot’s tone and response style.
- **Efficient Document Search**: FAISS provides fast and scalable document retrieval using embeddings.
- **Extensibility**: The system is modular, allowing for easy additions like integrating more advanced NLP models or TTS engines.

Potential Improvements

- **Improved Question Answering**: Use question-answering models (e.g., BERT, T5) to generate more precise answers from documents instead of returning the full document.
- **Advanced Emotion-based Speech Output**: Integrate emotion-based modulation in speech output, where the bot’s voice tone can vary based on detected emotion.
- **Avatar Integration**: Integrate an animated avatar for more interactive and immersive conversations, with lip sync for TTS output.
- **Voice Input Enhancements**: Improve voice input handling by using noise filtering and context-sensitive speech recognition.

Python Script to Run The System In Streamlit

```
import streamlit as st
import faiss
import numpy as np
from sentence_transformers import SentenceTransformer
from gtts import gTTS
import os
import speech_recognition as sr
from transformers import pipeline
import io

# Initialize SentenceTransformer model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Initialize the emotion detection model (for emotion analysis on recognized
text)
emotion_model = pipeline("text-classification", model="j-hartmann/emotion-
english-distilroberta-base")

# Initialize FAISS index globally (we'll populate it later)
index = faiss.IndexFlatL2(384)  # For the 'all-MiniLM-L6-v2' model, embedding
size is 384
documents = []

# Use session_state to persist conversation history
if "conversation_history" not in st.session_state:
    st.session_state.conversation_history = []  # Initialize conversation history

# Function to process documents and create embeddings
def process_documents(uploaded_files):
    global documents, index
    # Process each uploaded file
    for uploaded_file in uploaded_files:
        # Read the content of the uploaded file
        file_content = uploaded_file.read().decode("utf-8")  # Read and decode
the file to text

        # Split the content by lines, assuming each line is a document
        document_lines = file_content.splitlines()

        # Add the document lines to the global documents list
        documents.extend(document_lines)

    # Create embeddings for the documents
```



```python
    embeddings = model.encode(documents)
    embeddings = np.array(embeddings).astype('float32')

    # Add to FAISS index
    index.add(embeddings)

# Function to manually input documents (text input)
def input_documents_manually():
    global documents, index
    # User input for documents
    user_input = st.text_area("Manually enter your document(s) (one per line):",
height=150)
    if user_input:
        document_lines = user_input.splitlines()
        documents.extend(document_lines)

        # Create embeddings for the documents
        embeddings = model.encode(documents)
        embeddings = np.array(embeddings).astype('float32')

        # Add to FAISS index
        index.add(embeddings)
        st.success(f"{len(document_lines)} documents have been added manually.")

# Function to get response based on user query
def query_response(user_query):
    query_embedding = model.encode([user_query])
    query_embedding = np.array(query_embedding).astype('float32')

    D, I = index.search(query_embedding, k=1)  # Find most similar document
    response = documents[I[0][0]]  # Get the corresponding document
    return response

# Function to convert text to speech
def text_to_speech(text):
    tts = gTTS(text, lang='en')
    tts.save("response.mp3")
    os.system("start response.mp3")

# Function to recognize speech and convert it to text
def speech_to_text():
    recognizer = sr.Recognizer()

    with sr.Microphone() as source:
        print("Listening for your question...")
        recognizer.adjust_for_ambient_noise(source)
        audio = recognizer.listen(source)

    try:
        query = recognizer.recognize_google(audio)
        return query
    except sr.UnknownValueError:
        return "Sorry, I couldn't understand that."
    except sr.RequestError:
        return "Sorry, there was an issue with the speech recognition service."

# Emotion detection function
def detect_emotion(text):
    emotion = emotion_model(text)[0]['label']
    return emotion

# Store conversation history (query, response) and embeddings for fast search
def store_conversation(user_query, bot_response):
    # Store the conversation pair in session_state
    st.session_state.conversation_history.append({'user': user_query, 'response':
bot_response})

    # Store the embeddings of both the user query and bot response
    conversation_embeddings = model.encode([user_query, bot_response])
    conversation_embeddings = np.array(conversation_embeddings).astype('float32')

    # Add embeddings to FAISS index
    index.add(conversation_embeddings)

# Streamlit UI setup
st.title('Conversational Agent TTS & STT With Vector Database and Emotion
recognition')

# Step 1: Choose how to input documents
document_input_method = st.radio("Choose how to input documents:", ('Upload
Documents', 'Manually Enter Documents'))

# Step 2: Handle document input based on choice
if document_input_method == 'Upload Documents':
    uploaded_files = st.file_uploader("Upload your document(s) (Text files)",
type="txt", accept_multiple_files=True)

    if uploaded_files:
        st.write("Processing your documents...")
        process_documents(uploaded_files)
        st.write(f"{len(documents)} documents have been uploaded and processed successfully.")

        # Display the uploaded documents in the sidebar
        st.sidebar.title("Uploaded Documents")
        for idx, uploaded_file in enumerate(uploaded_files):
            st.sidebar.write(f"{idx + 1}. {uploaded_file.name}")
else:
    input_documents_manually()  # Allow manual text input

# Step 3: Choose Input Method (Text or Voice)
input_method = st.radio("Choose Input Method:", ('Text Input', 'Voice Input'))

# Handle text input queries
if input_method == 'Text Input' and documents:
    user_query = st.text_input("Ask your question:")
    if user_query:
        response = query_response(user_query)
        emotion = detect_emotion(response)
        st.write(f"Response: {response}")
        st.write(f"Detected Emotion: {emotion}")

        # Store the conversation history
        store_conversation(user_query, response)

        text_to_speech(response)  # Convert response to speech

# Handle voice input queries
elif input_method == 'Voice Input' and documents:
    if st.button("Click to Speak"):
        user_query = speech_to_text()
        if user_query:
            st.write(f"You said: {user_query}")
            response = query_response(user_query)
            emotion = detect_emotion(response)
            st.write(f"Response: {response}")
            st.write(f"Detected Emotion: {emotion}")

            # Store the conversation history
            store_conversation(user_query, response)

            text_to_speech(response)  # Convert response to speech

# Display the conversation context
st.subheader('Conversation History')
```

```
if st.session_state.conversation_history:
    for idx, conversation in enumerate(st.session_state.conversation_history):
        st.write(f"**{idx+1}. User**: {conversation['user']}")
        st.write(f"**Bot**: {conversation['response']}")
else:
    st.write("No conversations yet.")
```