Report on Adaptive Educational Chatbot Using Retrieval-Augmented Generation (RAG) and Vector Database

Created By - Ankit Goyal

Introduction

The purpose of this project was to develop an adaptive educational assistant chatbot capable of helping undergraduate students understand complex concepts through short, memorable stories. This chatbot integrates Retrieval-Augmented Generation (RAG) with a vector database to provide precise and contextually relevant information.

Approach Taken

1. System Design and Architecture

The chatbot was designed with the following components:

- **OpenAI's GPT-3.5-turbo Model**: Used for generating conversational responses.
- **Chroma Vector Database**: Employed to store and retrieve documents based on their semantic similarity.
- **Streamlit**: Utilised for creating the user interface.

2. Integration of Retrieval-Augmented Generation (RAG)

RAG combines retrieval-based methods and generation-based methods:

- **Document Retrieval**: Relevant documents are fetched from the vector database based on user queries.
- **Response Generation**: The model generates responses using both the query and the retrieved documents.

3. Implementation Details

- **Database Loading**: PDF documents are uploaded and processed to create vector representations using the Chroma vector database.
- **Custom Prompts**: Each user query is combined with a system message to maintain the educational context.
- **Context Handling**: The chat history is managed to ensure that the chatbot can maintain context over multiple interactions without exceeding token limits.

Challenges Faced and Solutions

1. Token Limit Exceedance

Challenge: The maximum context length for the model is 16,385 tokens. Long chat histories risk exceeding this limit, leading to errors.

Solution: Implemented a mechanism to truncate chat history dynamically to ensure it stays within the token limit while preserving essential context. This involved:

- Calculating the total length of the chat history.
- Truncating older interactions when the limit is approached.

2. Document Repetition in Responses

Challenge: Source documents were repeatedly displayed for multiple queries, causing redundancy and confusion.

Solution: Ensured that only the documents relevant to the most recent query are displayed by clearing the list of relevant documents before processing each new query.

3. Ensuring Educational Context

Challenge: Maintaining the educational context and preventing the chatbot from addressing non-educational queries.

Solution: Incorporated a persistent system message that reinforces the educational role of the chatbot. Custom prompts were crafted for each query to remind the model of its educational purpose.

Implementation Details

System Message

A consistent system message was used to maintain context:

```
system_message_content = """
You are an educational assistant. Your job is to help undergraduate students understand concepts and definitions by explaining them through short, memorable stories that even a 10-year-old can understand. If a user asks a question unrelated to education, kindly notify them that the chatbot is designed for educational purposes only.
"""
```

Loading the Database

```
def load_db(file, chain_type, k=3):
  loader = PyPDFLoader(file)
  documents = loader.load()
  text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=150)
  docs = text_splitter.split_documents(documents)
  embeddings = OpenAIEmbeddings()
  vectordb = Chroma(persist_directory=persist_directory,
embedding_function=embeddings)
  vectordb.add_documents(docs)
  retriever = vectordb.as_retriever(search_type="mmr", search_kwargs={"k": k})
  qa = ConversationalRetrievalChain.from_llm(
    llm=ChatOpenAI(model_name=llm_name, temperature=0.0),
    chain_type=chain_type,
    retriever=retriever,
    return_source_documents=True,
    return_generated_question=True
  )
  return qa, docs, vectordb
```

Handling Queries and Responses

```
def convchain(self, query):
    if not query:
        return ""

    # Clear relevant_docs before processing a new query
    self.relevant_docs = []

    # Check if the length of chat history exceeds 16,000 characters
    chat_history_length = sum(len(item[0]) + len(item[1]) for item in self.chat_history)
    if chat_history_length > 16000:
        self.chat_history = []

    # Truncate the chat history to fit within the context length limit
    token_limit = 16000
    truncated_history = []
    current_length = 0

    for q, a in reversed(self.chat_history):
        length = len(q) + len(a)
        if current_length + length <= token_limit:
            truncated_history.insert(0, (q, a))
            current_length += length
        else:
            break

    # Create a custom prompt based on the system message and current chat history
     custom_prompt = system_message_content + "\n" + "\n".join([f"User: {q}\nAssistant: {a}"
for q, a in truncated_history]) + f"\nUser: {query}\nAssistant:"

      result = self.qa({"question": query, "chat_history": truncated_history, "custom_prompt":
custom_prompt})
    self.chat_history.extend([(query, result["answer"])])
    self.relevant_docs = result["source_documents"]
    return result
```

Streamlit Interface

The chatbot was integrated with Streamlit to provide a user-friendly interface:

python

Copy code

```
tab1, tab2, tab3 = st.tabs(["Upload PDF", "View Data", "Chatbot"])

with tab1:
    uploaded_file = st.file_uploader("Choose a PDF file", type="pdf")
    if uploaded_file is not None:
        with open("temp.pdf", "wb") as f:
            f.write(uploaded_file.getbuffer())
        cb.call_load_db("temp.pdf")
        st.success("File loaded successfully!")

with tab2:
    if cb.docs:
        vectors = cb.get_all_vectors()
        if vectors:
            st.write("Some Example Vectors in the vector database:")
            for vector in vectors:
                st.write(vector)
        else:
            st.write("No vectors found in the vector database.")
    else:
        st.write("No documents loaded yet.")

    if st.button("Clear Database"):
        cb.clear_db()
        st.success("Chroma database cleared!")
        st.experimental_rerun()

with tab3:
    if cb.loaded_file:
        st.markdown(f"**Loaded File:** {cb.loaded_file}")

    query = st.text_input("Enter your question")
    if st.button("Generate Response"):
        if query:
            response = cb.convchain(query)
            if response:
                st.markdown(f"**User:** {query}")
                st.markdown(f"**ChatBot:** {response['answer']}")
                st.markdown("**Source Documents:**")
                for doc in cb.relevant_docs:
                    st.markdown(f"- {doc.page_content}")
            else:
                st.warning("Please enter a question to get a response.")

    if st.button("Clear Chat History"):
        cb.chat_history = []
        st.success("Chat history cleared!")
```

Conclusion

This project successfully developed an adaptive educational assistant chatbot using RAG and a vector database. By addressing key challenges such as token limit exceedance and document repetition, the chatbot can now provide precise, contextually relevant, and educationally focused responses. Future enhancements could include more advanced context handling and continuous learning mechanisms to further personalize and improve the chatbot's interactions.