Name: Ian Jure Macalisang

Date: Nov. 26, 2024

DS314: Critical Analysis Task - LangChain Memory Module

Issues Identified in the Previous Tutorial

Deprecated Memory Module: The memory module from LangChain is deprecated and only applicable to older versions. It is now recommended to use LangGraph persistence to incorporate memory when building an application, as detailed in this article: https://python.langchain.com/docs/tutorials/chatbot/.

Cluttered User Interface: The user interface in the previous tutorial is cumbersome and unintuitive. It lacks a ChatGPT-style design, which would be more user-friendly. Additionally, it does not utilize Streamlit's chat element components, which are not only easy to implement but also designed to enhance the user experience.

Ineffective Summarization Feature: The summarization feature does not offer a distinctive or user-friendly approach to presenting summarized conversations in an organized format.

Paid Access Requirement: Users are required to obtain a paid API key to access the application, which is not ideal for a development environment.

The updated tutorial below offers solutions to address the issues outlined above.

Building a ChatGPT Clone with Summarization Option using Google Gemini

This tutorial guides you through the steps to build a ChatGPT clone using **LangChain**, the **Google Gemini 1.5 Flash Model**, and **Streamlit** for the user interface. It also includes features such as clearing conversation history and a summarization option, allowing users to either reset the chat or generate a summary of their conversation.

Project Setup

1. **Install the necessary libraries**: To start with, you'll need the following Python libraries:

```
pip install fpdf2 pillow streamlit streamlit-float langchain langgraph
langchain-google-genai
```

2. **Create `app.py`**: This will be the main file that runs the Streamlit application.

Step 1: Import Required Libraries

We will use `streamlit` for building the UI, `langchain-google-genai` for integrating the Gemini model, and the conversation memory features provided by `langgraph` to handle the chat history.

```
import time
import uuid
import streamlit as st
from PIL import Image
from fpdf import FPDF
from streamlit_float import *
from langchain_core.messages import HumanMessage
from langgraph.graph import START, StateGraph, MessagesState
from langgraph.checkpoint.memory import MemorySaver
from langchain_google_genai import ChatGoogleGenerativeAI
```

Libraries and Tools:

- **time**: Used in the text streaming function.
- **uuid**: Used to create a unique identifier for each conversation thread.
- **streamlit**: Used to build the application's user interface.
- **PIL**: Used to import the application's logo and icon.
- **fpdf**: Used to export the chat summary to a PDF file.
- **streamlit_float**: Used to create a floating button in Streamlit.
- **langchain**: Used to integrate the LLM model into the application.
- **langgraph**: Used to create persistent memory for the model.
- **langchain_google_genai**: Used to access Google Gemini model.

Step 2: Configure and Style the Application

Next, we will configure the page and utilize CSS to style the application.

```
# [STREAMLIT] PAGE CONFIGURATION
icon = Image.open("icon.png")
st.set_page_config(page_title="ChatGPT Clone", page_icon=icon)
st.logo("logo.svg")

# [STREAMLIT] HIDE MENU
hide_menu = """
    <style>
    #MainMenu {
        visibility: hidden;
    }
    footer {
        visibility: hidden;
```

```
    }
    div[data-testid="stDecoration"] {
        visibility: hidden;
        height: 0%;
        position: fixed;
    }
    div[data-testid="stStatusWidget"] {
        visibility: hidden;
        height: 0%;
        position: fixed;
    }
    [data-testid="stToolbar"] {
        display: none;
    }
    </style>
    """
st.markdown(hide_menu, unsafe_allow_html=True)

# [STREAMLIT] CENTER TOAST
cntr_toast = """
    <style>
    [data-testid="stToastContainer"] {
        display: flex;
        justify-content: center;
        align-items: center;
        position: fixed;
        top: 72%;
        left: 50%;
        transform: translate(-50%, -50%);
        z-index: 999999;
    }
    [data-testid="stToast"] {
        left: 48%;
    }
    </style>
    """
st.markdown(cntr_toast, unsafe_allow_html=True)

# [STREAMLIT] ADJUST ICON SIZE
icon = """
    <style>
    [data-testid="stChatMessageAvatarCustom"] {
        width: 2.2rem;
        height: 2.2rem;
        background-color: #212121;
        border: solid white 1px;
        border-radius: 2.5rem;
    }
    [data-testid="stIconMaterial"] {
        font-size: 1.8rem;
```

```
        margin-left: -0.3rem;
        color: white;
    }
    </style>
        """
st.markdown(icon, unsafe_allow_html=True)

# [STREAMLIT] ADJUST LOGO SIZE
logo = """
    <style>
    [data-testid="stLogo"] {
        width: 18rem;
        height: auto;
    }
    </style>
        """
st.markdown(logo, unsafe_allow_html=True)

# [STREAMLIT] ADJUST BUTTON BORDER
btn_border = """
    <style>
    [data-testid="stBaseButton-secondary"] {
        border: 2px solid #f0f0f0;
        height: 2.8rem;
    }
    </style>
        """
st.markdown(btn_border, unsafe_allow_html=True)

# [STREAMLIT] ADJUST SETTINGS BUTTON
set_btn = """
    <style>
    [class="st-emotion-cache-12607u7 ef3psqc19"] {
        border-radius: 5rem;
        width: 3rem;
        height: 3rem;
    }
    </style>
        """
st.markdown(set_btn, unsafe_allow_html=True)

# [STREAMLIT] ADJUST TOP PADDING
top = """
    <style>
    .block-container {
        padding-top: 0rem;
        padding-bottom: 0rem;
        margin-top: -5rem;
    }
    </style>
```

```
        """
st.markdown(top, unsafe_allow_html=True)

# [STREAMLIT] ADJUST HEADER
header = """
    <style>
    [data-testid="stHeader"] {
        height: 7rem;
        width: auto;
        z-index: 1;
    }
    </style>
        """
st.markdown(header, unsafe_allow_html=True)

# [STREAMLIT] ADJUST USER CHAT ALIGNMENT
reverse = """
    <style>
    [class="stChatMessage st-emotion-cache-janbn0 eeusbqq4"] {
        flex-direction: row-reverse;
        text-align: right;
        background-color: #2f2f2f;
    }
    </style>
        """
st.markdown(reverse, unsafe_allow_html=True)

# [STREAMLIT] HIDE USER ICON
hide_icon = """
    <style>
    [data-testid="stChatMessageAvatarUser"] {
        display: none;
    }
    </style>
        """
st.markdown(hide_icon, unsafe_allow_html=True)

# [STREAMLIT] ADJUST CHAT INPUT PADDING
bottom = """
    <style>
    [data-testid="stBottom"] {
        padding-bottom: 2rem;
        background: #212121;
    }
    </style>
        """
st.markdown(bottom, unsafe_allow_html=True)

# [STREAMLIT] CHAT INPUT BORDER
chat_border = """
```

```
    <style>
    [data-testid="stChatInput"] {
        border: 2px solid #f0f0f0;
        border-radius: 1rem;
    }
    </style>
        """
st.markdown(chat_border, unsafe_allow_html=True)

# [STREAMLIT] TEXT AREA BORDER
text_border = """
    <style>
    [data-baseweb="textarea"] {
        border: 1px;
    }
    </style>
        """
st.markdown(text_border, unsafe_allow_html=True)
```

Step 3: Initialize Session State Variables

To manage state between different interactions, we will initialize the following session variables:

- **conversation**: Stores the conversation context.
- **thread_id**: Stores a UUID (Universally Unique Identifier) for each conversation thread.
- **messages**: Stores chat messages between the user and the bot.
- **API_Key**: Stores the user's API key.

```
# [STREAMLIT] INITIALIZING SESSION STATES
if 'conversation' not in st.session_state:
    st.session_state['conversation'] = None
if 'thread_id' not in st.session_state:
    st.session_state['thread_id'] = str(uuid.uuid4())
if 'messages' not in st.session_state:
    st.session_state['messages'] = []
if 'API_Key' not in st.session_state:
    st.session_state['API_Key'] = ""
```

Step 4: Define the `stream` Function

To replicate ChatGPT's response style, we will implement a function that streams the bot's text responses, creating a text-streaming effect for an interactive experience.

```
# [STREAMLIT] STREAM CHAT RESPONSE
def stream(content):
```

```
    for word in content.split(" "):
        yield word + " "
        time.sleep(0.03)
```

Step 5: Define the `generate_pdf` Function

The `generate_pdf` function will export the entire conversation into a well-structured PDF file. This ensures the output is both readable and compatible with Streamlit's components.

```
# [FPDF] GENERATE A PDF FILE FROM CONVERSATION
def generate_pdf():
    pdf = FPDF()
    pdf.set_auto_page_break(auto=True, margin=15)
    pdf.add_page()
    pdf.set_font("Arial", size=12)
    pdf.set_left_margin(10)
    pdf.set_right_margin(10)

    pdf.set_font("Arial", size=16, style="B")
    pdf.cell(200, 10, txt="Chat Conversation", ln=True, align="C")
    pdf.ln(10)

    pdf.set_font("Arial", size=12)
    for message in st.session_state['messages']:
        if message['role'] == "assistant":
            pdf.set_text_color(0, 100, 0)
            pdf.multi_cell(0, 10, txt=f"Assistant:
{message['content']}")
        else:
            pdf.set_text_color(0, 0, 255)
            pdf.multi_cell(0, 10, txt=f"User: {message['content']}")
        pdf.ln(1)

    pdf_output = pdf.output(dest="S")
    return bytes(pdf_output)
```

Step 6: Creating the Chat UI

We will leverage Streamlit's built-in `chat elements` to build the main user interface. This simplifies implementation while ensuring the interface is visually appealing.

```
# [STREAMLIT] CHAT BOT GREETINGS
with st.chat_message("assistant", avatar=":material/token:"):
    st.markdown("How can I assist you? ")

# [STREAMLIT] DISPLAY THE EXISTING CHAT HISTORY
for message in st.session_state['messages']:
```

```
    if message["role"] == "assistant":
        with st.chat_message(message["role"],
avatar=":material/token:"):
            st.markdown(message["content"])
    else:
        with st.chat_message(message["role"]):
            st.markdown(message["content"])

# [STREAMLIT] USER CHAT INPUT
user_input = st.chat_input("Say something.")
```

Key UI Components:

- **chat_input**: Captures the user's input, which is then used to generate a bot response.
- **chat_message**: Displays either the user's or the bot's messages in a clean, readable format.

Step 7: Handle State and Generate Model Response

To ensure conversations persist between requests, we will store the conversation history and chat messages. This prevents the conversation from resetting with every page reload and ensures efficient API usage.

```
# [STREAMLIT] IF SEND BUTTON IS CLICKED
if user_input:

    # [STREAMLIT] CHECK IF API KEY IS INPUTTED
    if st.session_state['API_Key'] != "":

        # [STREAMLIT] SHOW USER MESSAGE
        with st.chat_message("user"):
            st.markdown(user_input)

        # [STREAMLIT] STORE USER MESSAGE IN SESSION STATE
        st.session_state['messages'].append({"role": "user",
                                             "content": user_input})

        # [LANGGRAPH] INITIALIZE LANGGRAPH AND MEMORY
        if st.session_state['conversation'] is None:

            # [LANGGRAPH] DEFINE THE STATE GRAPH
            workflow = StateGraph(state_schema=MessagesState)
            model = ChatGoogleGenerativeAI(model="gemini-1.5-flash",
                    google_api_key=st.session_state['API_Key'])

            # [LANGGRAPH] DEFINE THE LLM NODE
            def call_model(state: MessagesState):
                response = model.invoke(state["messages"])
```

```
                return {"messages": response}

            workflow.add_edge(START, "model")
            workflow.add_node("model", call_model)

            # [LANGGRAPH] ADD MEMORY SAVER
            memory = MemorySaver()
            app = workflow.compile(checkpointer=memory)

            st.session_state['conversation'] = app

        # [LANGGRAPH] SEND THE INPUT TO LANGGRAPH
        app = st.session_state['conversation']
        thread_id = st.session_state['thread_id']
        config = {"configurable": {"thread_id": thread_id}}

        # [STREAMLIT] SHOW AI RESPONSE
        input_message = HumanMessage(content=user_input)
        response_text = ""
        for event in app.stream({"messages": [input_message]},
                                config, stream_mode="values"):
            response_text = event["messages"][-1].content
        with st.chat_message("assistant", avatar=":material/token:"):
            st.write(stream(response_text))

        # [STREAMLIT] STORE AI RESPONSE IN SESSION STATE
        st.session_state['messages'].append({"role": "assistant",
                                             "content":
response_text})

    else:
        st.toast("**API key not found**. Please set your API key in
the chat options.", icon=")
```

Flow of Interaction:

Every time a user sends a message to the bot, the following process occurs:

- **API Key Validation**: Check if the user's API key is stored in `st.session_state['API_Key']`.
- **Display User's Message**: Show the user's message using `st.chat_message()`.
- **Store User's Message**: Save the user's message in `st.session_state['messages']`.
- **Initialize Conversation**: Verify if a conversation exists in `st.session_state['conversation']`. If not, initialize a LangGraph workflow and memory.
- **Generate Bot Response**: Send the user's message to the LangGraph workflow and memory to generate a response from the model.
- **Display Bot's Response**: Show the bot's response using `st.chat_message()`.

- **Store Bot's Response**: Save the bot's response in `st.session_state['messages']`.

Why LangGraph?

Since LangChain's ConversationChain and memory modules are deprecated, we use LangGraph, which provides built-in persistence and supports advanced features like human-in-the-loop interactions and memory management.

Step 8: Create Chat History Options

To implement the clearing and summarizing features and handle the user's API key, we will create a chat options section using Streamlit's modal dialog component and the `streamlit-float` library.

```
# [STREAMLIT] CHAT HISTORY OPTIONS
float_init()

@st.dialog("Chat Options")
def open_options():
    col1, col2 = st.columns(2)

    # [STREAMLIT] CLEAR SESSION STATES
    with col1:
        st.write("Clear Conversation")
        if len(st.session_state['messages']) != 0:
            clear = st.button("**CLEAR**", type="primary",
key="clear",
                              use_container_width=True)
            if clear:
                st.session_state['messages'] = []
                st.session_state['conversation'] = None
                st.rerun()
        else:
            st.button("**CLEAR**", type="primary", key="clear",
                      disabled=True, use_container_width=True)

    # [STREAMLIT] DOWNLOAD CONVERSATION SUMMARY
    with col2:
        st.write("Summarize Conversation")
        if len(st.session_state['messages']) != 0:
            download = st.download_button(label="**SUMMARIZE**",
type="primary",
                                          key="summarize",
data=generate_pdf(),

file_name="conversation.pdf",
                                          mime="application/pdf",
```

```
                                                use_container_width=True)
            if download:
                st.rerun()
        else:
            st.button("**SUMMARIZE**", type="primary",
key="summarize",
                      disabled=True, use_container_width=True)

    # [STREAMLIT] INPUT API KEY
    if st.session_state['API_Key'] == '':
        user_API = st.text_input("Enter Your Gemini API Key",
type="password")
        if user_API:
            if st.button("**SAVE KEY**", type="secondary",
                         key="save", use_container_width=True):
                st.session_state['API_Key'] = user_API
                st.rerun()
        else:
            st.button("**SAVE KEY**", type="secondary", key="save",
                      disabled=True, use_container_width=True)
    else:
        user_API = st.text_input("Enter Your Gemini API Key",
                                 value=st.session_state['API_Key'],
                                 type="password")
        if user_API:
            if st.button("**SAVE KEY**", type="secondary", key="save",
                         use_container_width=True):
                st.session_state['API_Key'] = user_API
                st.rerun()
        else:
            st.button("**SAVE KEY**", type="secondary", key="save",
                      disabled=True, use_container_width=True)

button_container = st.container()
with button_container:
    if st.button("⚙", type="secondary"):
        open_options()

button_css = float_css_helper(width="1.8rem", height="2rem",
right="3rem",
                              top="2rem", transition=0)
button_container.float(button_css)
```

Feature Implementation:

Clear Button:

Resets the conversation and chat history by performing the following actions:

- Set `st.session_state['messages']` to an empty list.

- Set `st.session_state['conversation']` to None.

Summarize Button:

Calls the `generate_pdf` function to generate a summary of the conversation in PDF format.

API Key Handling:

When the user inputs an API key, it will be stored in `st.session_state['API_Key']`. This ensures the key persists across application reruns.

Step 9: Running the App

To run the application, simply execute the following command:

```
streamlit run app.py
```

Summary:

This tutorial provides a step-by-step guide to building a ChatGPT-like chatbot application using LangChain, the Google Gemini 1.5 Flash Model, and Streamlit for the user interface. Key features include conversation persistence, chat summarization, and an interactive user experience with streaming text responses. Using LangGraph for memory management ensures efficient state handling, while session variables maintain conversation context and API key storage across interactions. The application supports clearing chat history, generating PDF summaries, and leveraging Streamlit's components for a visually appealing interface. By combining these tools, the tutorial demonstrates how to create a robust and user-friendly chatbot with advanced capabilities.