Prompts in LangChain

1. Correction Regarding LLM Temperature Parameter

- **Initial (Incorrect) Explanation:** "अगर आप जीरो के आसपास रखते हो तो आपके एलएलएम का जो आउटपुट है वो बहुत डिटरमिनिस्टिक आता है और अगर आप टू के आसपास रखते हो तो आपका एलएलएम का जो वैल्यू है बहुत क्रिएटिव आता है" (If you keep it around zero, your LLM's output is very deterministic, and if you keep it around two, your LLM's value becomes very creative).
- **Corrected Explanation: Temperature = 0 or close to 0:** For the "same given input, your LLM will always give the same output." This is useful for applications requiring consistent responses.
- **Temperature = 1.5-2:** For the "same input, your output will be quite different, creative, every time." This is suitable for applications where varied or imaginative responses are desired.
- **Example:** Demonstrates changing temperature from 0 to 0.5 to 1.5 when asking GPT-4 to write a five-line poem on cricket, showing increased creativity/variability in output with higher temperature.

2. Understanding Prompts

- **Definition:** A prompt is "the message you send to the LLM when you interact with it." The presenter notes that they extensively used prompts in previous videos without explicitly using the term "prompt."
- **Quote:** "बेसिकली आप किसी एलएलएम से जब इंटरेक्ट करते हो तो वहां पर आप जो मैसेज भेजते हो एलएलएम को उसी को हम प्रोमट बुलाते हैं" (Basically, when you interact with an LLM, the message you send to the LLM is what we call a prompt).
- **Modality:** While the video focuses on **text-based prompts** (which account for 99% of current use cases), the presenter acknowledges the existence of **multimodal prompts** (e.g., using images, sound, or video as input, as seen in ChatGPT).
- **Importance:** Prompts are "super important" because the LLM's output is highly dependent on them. Slight changes in prompts can significantly alter the output.
- **Prompt Engineering:** The critical role of prompts has led to a new job profile called "Prompt Engineering."

3. Static vs. Dynamic Prompts

The video makes a crucial distinction between static and dynamic prompts, highlighting the limitations of the former and the benefits of the latter.

- **Static Prompts:**
- **Definition:** Prompts where the user manually types the entire prompt for each interaction.
- **Problem:** Gives "quite a lot of control" to the user, leading to potential issues:
- **Sensitivity:** LLM output is highly sensitive to prompt changes.
- **Inaccuracies/Hallucinations:** Users might make mistakes (e.g., typos in research paper names), leading the LLM to hallucinate or provide undesirable outputs.
- **Lack of Consistency:** Difficult to ensure a consistent user experience (e.g., always including analogies or specific details in summaries) if the user has full control over the prompt.
- **Example:** User directly typing "Summarize the Attention Is All You Need paper in simple fashion" into an input box.

- **Dynamic Prompts:**
- **Solution:** To mitigate the problems of static prompts, it's recommended to "prepare a template beforehand." The user provides specific "fill-in-the-blank" values, which are then inserted into the pre-defined template.
- **Benefits: Controlled Output:** Ensures consistent output characteristics (e.g., always including mathematical details, code snippets, or analogies) regardless of user input.
- **Reduced Errors:** By providing structured inputs (e.g., dropdowns for paper names or explanation styles), the chances of user-induced errors are significantly reduced.
- **Flexibility with Consistency:** A single template can be used to generate diverse summaries (e.g., any paper, any style, any length).
- **Example Template:** "Please summarize the research paper titled **{paper_input}** with the following specifications: Explanation style **{explanation_style}** and explanation length **{explanation_length}**. Include relevant mathematical equations... Also add analogies... Ensure the summary is clear, accurate and aligned with the provided style and length."
- **Implementation:** Demonstrated using a Streamlit UI with dropdowns for paper selection, explanation style, and length.

4. PromptTemplate Class in LangChain

The PromptTemplate class is used to create and manage dynamic prompts in LangChain.

- **Usage:**

1. Import PromptTemplate from langchain_core.prompts.
2. Define the template string with placeholders (e.g., {paper_input}).
3. Define input_variables as a list of strings corresponding to the placeholders.
4. Call template.invoke() with a dictionary mapping input_variables to user-provided values.
5. The resulting prompt (a filled string) is then sent to the LLM.

- **Why PromptTemplate over f-strings?** (This section answers a common doubt about why not simply use Python's f-strings for dynamic prompts)
- **Built-in Validation:**PromptTemplate offers automatic validation. If placeholders in the template don't match the input_variables provided, it throws an error during development, preventing runtime failures.
- **Quote:** "अगर आप प्रोमट टेंप्लेट क्लास की हेल्प से प्रोमट बनाओगे तो आपको बाय डिफॉल्ट कुछ वैलिडेशन मिलता है" (If you create a prompt with the help of PromptTemplate class, you get some validation by default).
- **Reusability:**PromptTemplate objects can be easily saved as JSON files (using template.save()) and loaded back (using load_prompt() from langchain_core.prompts).
- This promotes modularity and allows templates to be reused across different parts of a large application or in different files, preventing code bulkiness.
- **Tight Coupling with LangChain Ecosystem (Chains):**PromptTemplate seamlessly integrates with other LangChain components, particularly "Chains."
- **Example:** A PromptTemplate object and an LLM model can be directly linked in a chain (e.g., chain = template | model), allowing a single chain.invoke() call to handle both prompt formatting and LLM interaction. This is not possible with simple f-strings.
- **Quote:** "जो लैंग चन का पूरा यूनिवर्स है उसमें बहुत अच्छे से फिट होता है प्रोम टेंप्लेट" (The entire LangChain universe fits PromptTemplate very well).

5. Building a Simple Chatbot: The Need for Context

A simple console-based chatbot is demonstrated, which initially suffers from a critical flaw: **lack of context**.

- **Problem:** The LLM does not remember previous messages in a conversation. When a follow-up question is asked (e.g., "Now multiply the bigger number by 10" after being told "2 is greater than 0"), the LLM loses the context of what "the bigger number" refers to.
- **Quote:** "आपके यह जो चैट बॉट है इसके पास कॉन्टेक्स्ट नहीं है उसको प्रीवियस मैसेजेस याद ही नहीं है" (This chatbot of yours does not have context; it does not remember previous messages).
- **Solution:** Maintain a **chat history** (a list of all previous messages in the conversation) and send this entire history to the LLM with each new user prompt. This allows the LLM to understand the ongoing conversation.

6. LangChain Message Types for Chat History

While sending a list of raw strings as chat history works, it's problematic because the LLM doesn't know who sent which message. LangChain solves this by introducing specific message types:

- **System Message (SystemMessage):**
- Purpose: Sets the initial behavior or persona of the AI. Always sent at the beginning of a conversation.
- Example: "You are a helpful assistant." or "You are a very knowledgeable doctor."
- **Human Message (HumanMessage):**
- Purpose: Represents messages sent by the user (the "human") to the LLM.
- Example: "Tell me the capital of India."
- **AI Message (AIMessage):**
- Purpose: Represents messages generated and sent by the AI (LLM) back to the user.
- Example: "The capital of India is New Delhi."
- **Benefits of Labeled Messages:** By labeling each message with its sender type, the LLM can easily understand the flow of the conversation, even with a long chat history, preventing confusion about who said what.
- **Quote:** "हमारा हर मैसेज अब लेबल्ड है कि वह सिस्टम मैसेज है या फिर ह्यूमन मैसेज है या फिर एआई मैसेज है तो अब फ्यूचर में जब हम बात कर रहे होंगे अपने एलएलएम से और हमारा चैट कितना भी बड़ा हो जाए कोई फर्क नहीं पड़ता हमारा एलएलएम हमेशा यह समझ पाएगा कौन सी बात किसने कही है" (Now every message of ours is labeled whether it is a System Message, a Human Message, or an AI Message. So now in the future, when we are talking to our LLM, no matter how long our chat becomes, our LLM will always be able to understand who said what).

7. ChatPromptTemplate for Dynamic Multi-Turn Conversations

Similar to PromptTemplate for single-turn static prompts, ChatPromptTemplate is designed for **dynamic multi-turn conversations** (i.e., when working with a list of messages where some parts of the messages are variable).

- **Purpose:** To create dynamic lists of messages, where parts of system messages, human messages, or AI messages can be placeholders.
- **Usage:**

1. Import ChatPromptTemplate from langchain_core.prompts.
2. Define the template as a **list of tuples**, where each tuple represents a message: (role_type, message_content_with_placeholders).

- **Note on Syntax:** The presenter points out a "weird behavior" where using the SystemMessage and HumanMessage classes directly inside ChatPromptTemplate does not correctly fill placeholders. Instead, a tuple syntax ("system", "U are a helpful {domain} expert") is required.

1. Call chat_template.invoke() with a dictionary to fill the placeholders.
2. The output is a list of correctly formatted LangChain message objects ready to be sent to the LLM.

- **Analogy:** "ChatPromptTemplate का एगजैक्टली वही काम है जो PromptTemplate का काम है टू क्रिएट डायनेमिक टेंपलेट्स द ओनली डिफरेंस इज प्रॉन टेंप्लेट को आप यूज करते हो सिंगल टर्न मैसेजेस के लिए और चा चार्ट प्रोम टेंप्लेट को आप यूज करते हो मल्टी टर्न कन्वर्सेशन वाले मैसेजेस के लिए" (ChatPromptTemplate's exact function is the same as PromptTemplate's - to create dynamic templates. The only difference is that PromptTemplate is used for single-turn messages, and ChatPromptTemplate is used for multi-turn conversation messages).

8. MessagesPlaceholder for Dynamic Chat History Insertion

The MessagesPlaceholder is a specialized placeholder used within ChatPromptTemplate to dynamically insert a list of messages, typically the chat history, at runtime.

- **Problem Solved:** When a user re-engages in a conversation after a break, the previous chat history needs to be loaded (e.g., from a database) and inserted into the current prompt to provide context.

1. **Usage:**Import MessagesPlaceholder from langchain_core.prompts.
2. In the ChatPromptTemplate's list of messages, insert MessagesPlaceholder(variable_name). This acts as a slot for a list of messages.
3. When invoking the ChatPromptTemplate, pass the loaded chat history (as a list of LangChain message objects) to the MessagesPlaceholder's variable name in the dictionary.

- **Example Scenario:** An Amazon customer support chatbot. When a user asks "Where is my refund?" after a previous interaction about a refund request, the MessagesPlaceholder is used to inject the entire prior conversation history, allowing the LLM to understand the current query in context.

- **Quote:** "मैसेज प्लेस होल्डर इज अ प्लेस होल्डर जहां पे आप सेट ऑफ मैसेजेस के लिए एक प्लेस होल्डर क्रिएट करते हो जनरली आप इसको यूज करते हो टू अ रिट्रीव एंड स्टोर द चैट हिस्ट्री" (Message Placeholder is a placeholder where you create a placeholder for a set of messages. Generally, you use it to retrieve and store the chat history).