Relatório 12 - Leitura: Chatboot + LangChain (III)

Vitor Eduardo de Lima Kenor

Descrição da atividade

Neste card iremos analisar uma documentação e dois artigos. A documentação é a do LangChain que ensina a construir um chatbot. Já os artigos, um deles se trata da criação de um chatbot e o outro apresenta o chatbot Sahaay, que usa LangChain e LLMs para automatizar o atendimento ao cliente.

Começando pela documentação do LangChain, nela é ensinado através de trechos de código como implementar um chatbot passo a passo. Eu repliquei os trechos apresentados na documentação em um notebook Colab que irei explicar brevemente o que acontece em cada célula de código.

```
from IPython.display import clear_output

!pip install langchain
!pip install -U langchain-openai
!pip install langchain_community

clear_output()
```

```
[ ] import os
    from langchain_openai import ChatOpenAI
    from langchain_core.messages import AIMessage, HumanMessage
    from langchain_core.chat_history import BaseChatMessageHistory, InMemoryChatMessageHistory
    from langchain_core.runnables.history import RunnableWithMessageHistory
```

Na primeira célula de código é feita a instalação das bibliotecas necessárias. Já na segunda é feita a importação de todas as bibliotecas usadas no notebook.

```
[ ] os.environ["OPENAI_API_KEY"] = "chave da API"

    model = ChatOpenAI(model="gpt-4o-mini")
```

```
[ ] response = model.invoke([HumanMessage(content="Oi! meu nome é Vitor")])

    response.content
```

```
'Oi, Vitor! Como posso ajudá-lo hoje?'
```

```
[ ] response = model.invoke([HumanMessage(content="Qual é o meu nome?")])

    response.content
```

```
'Desculpe, mas não tenho acesso a informações pessoais e não posso saber o seu nome. Como posso ajudar você hoje?'
```

A terceira eu defino minha chave de API no ambiente e seleciono o modelo de LLM que irei usar. Na quarta eu mando uma mensagem que eu falo meu nome, só que na quinta célula ele não sabe responder meu nome mesmo eu tendo falado anteriormente. Isso aconteceu porque ainda não implementamos uma memória para ele, logo ele não sabe o que foi perguntado e respondido anteriormente.

```python
store = {}

def get_session_history(session_id: str) -> BaseChatMessageHistory:
    if session_id not in store:
        store[session_id] = InMemoryChatMessageHistory()
    return store[session_id]
```

```python
with_message_history = RunnableWithMessageHistory(model, get_session_history)
config = {"configurable": {"session_id": "abc2"}}

response = with_message_history.invoke(
    [HumanMessage(content="Oi! meu nome é Vitor")],
    config=config,
)

response.content
```

```
'Oi, Vitor! Como posso ajudar você hoje?'
```

```python
response = with_message_history.invoke(
    [HumanMessage(content="Qual é o meu nome?")],
    config=config,
)

response.content
```

```
'Seu nome é Vitor! Posso ajudar com mais alguma coisa?'
```

Na sexta célula, criamos um dicionário que irá guardar o histórico de mensagens e uma função que verifica se determinado histórico já existe através de seu id. Na sétima célula se define um id para identificar o histórico da conversa e fizemos o teste se apresentando novamente. Podemos ver que foi um sucesso na oitava célula, já que agora ele consegue identificar o nome dito na chamada anterior.

```python
response = with_message_history.invoke(
    [HumanMessage(content="Quanto é 2 + 2 / 2?")],
    config=config,
)

print(response.content)
```

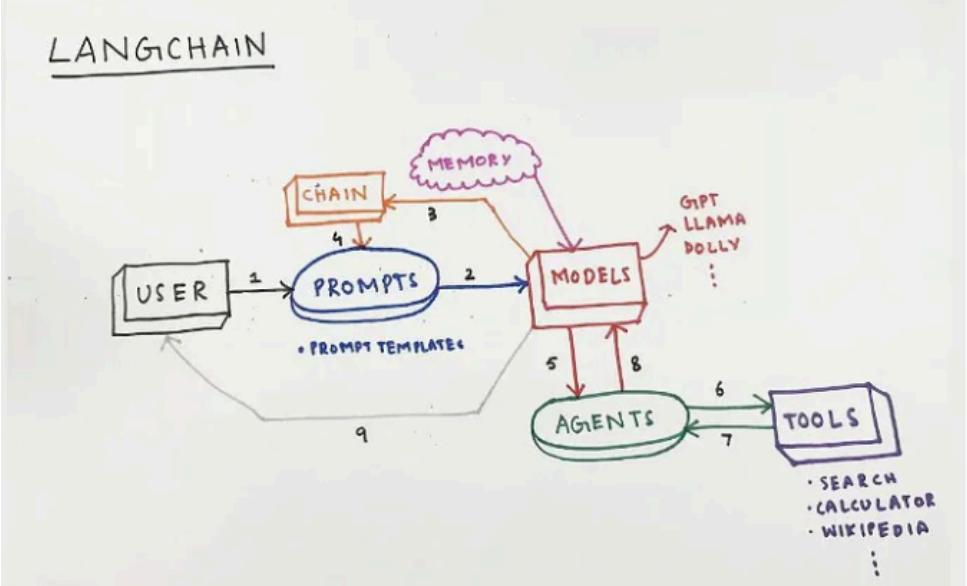
```
Para resolver a expressão 2 + 2 / 2, é importante seguir a ordem das operações matemáticas (também conhecida como PEMDAS/BODMAS):

1. Primeiro, realizamos a divisão: \( 2 / 2 = 1 \).
2. Depois, somamos: \( 2 + 1 = 3 \).

Portanto, \( 2 + 2 / 2 = 3 \).
```

```python
response = with_message_history.invoke(
    [HumanMessage(content="Não entendi, poderia me explicar de uma maneira mais simples?")],
    config=config,
)

print(response.content)
```

```
Claro! Vamos simplificar:

Quando você tem a expressão \( 2 + 2 / 2 \), precisa seguir uma regra chamada ordem das operações.

1. **Divisão primeiro:** Olhe para a parte que tem uma divisão. \( 2 / 2 \) é igual a \( 1 \).
2. **Agora substitua:** A expressão agora fica \( 2 + 1 \).
3. **Faça a soma:** \( 2 + 1 \) é igual a \( 3 \).

Por isso, \( 2 + 2 / 2 \) é igual a \( 3 \). Se precisar de mais ajuda, é só avisar!
```

Essas últimas células de código são só um exemplo mostrando que a memória foi implementada com sucesso no chatbot.

Depois de concluir a documentação vamos para o artigo “Build Chatbot with LLMs and LangChain”. Neste artigo o autor compartilha a experiência dele na construção de um chatbot para a empresa Dash Company. O chatbot foi criado para ajudar os usuários a analisar os dados de suas lojas digitais e aconselhar a como melhorar sua oficina. No artigo o autor explica o que são LLMs, o que é Langchain e o porque o escolheu como ferramenta no projeto.



Depois ele ilustra essa cadeia simples no LangChain e diz como é importante essa conexão lógica para o funcionamento da ferramenta.

```
de langchain.prompts importar PromptTemplate
de langchain.llms importar HuggingFace
de langchain.chains importar LLMChain

prompt = PromptTemplate(
    input_variables=[ "city" ],
    template= "Descreva um dia perfeito em {city}?" ,
)

llm = HuggingFace(
          model_name= "gpt-neo-2.7B" ,
          temperature= 0.9 )

llmchain = LLMChain(llm=llm, prompt=prompt)
llmchain.run( "Paris" )
```

Em seguida é dado esse exemplo em código de como funcionam essas ligações lógicas que são chamadas de correntes. Depois ele mostra como adicionar memória no seu agente, um passo importante para que sua aplicação consiga manter uma conversa. Logo após o autor fala sobre as tools do LangChain, elas são funções que os agentes podem usar para

interagir com o mundo, depois de explicar ele mostra como criar uma tool personalizada que serve para o agente saber a data atual. Finalizando o artigo, o autor combina tudo o que ensinou em um chatbot que analisa dados do usuário pela API do aplicativo e responde às perguntas.

Agora vamos ver o que é abordado no artigo “Automating Customer Service using LangChain”. O artigo é bem curto e nele é abordado o futuro do atendimento ao cliente, destacando o uso de LLMs para automatizar e personalizar o suporte. Ele propõe um framework open source baseado em LangChain, capaz de extrair informações de um website para responder às consultas em tempo real. A metodologia inclui a coleta de dados via web scraping, uso de embeddings para representar conhecimento e a integração com modelos como Google Flan-T5 XXL. Finalizando o artigo conclui dizendo que essa implementação melhora a experiência do usuário, tornando o atendimento mais rápido, eficiente e acessível.

Conclusões

A análise da documentação do LangChain e dos dois artigos nos permitiu entender melhor como construir e aprimorar chatbots utilizando LLMs. Com isso, aprendemos que o LangChain é uma ferramenta poderosa para a construção de chatbots avançados, permitindo integrações com diversas fontes de dados e tornando as interações mais naturais e eficazes.

Referências

Link da documentação do LangChain:
https://python.langchain.com/v0.2/docs/tutorials/chatbot/

Link do artigo “Build Chatbot with LLMs and LangChain”:
https://medium.com/@dash.ps/build-chatbot-with-llms-and-langchain-9cf610a156ff

Link do artigo “Automating Customer Service using LangChain”:
https://arxiv.org/pdf/2310.05421