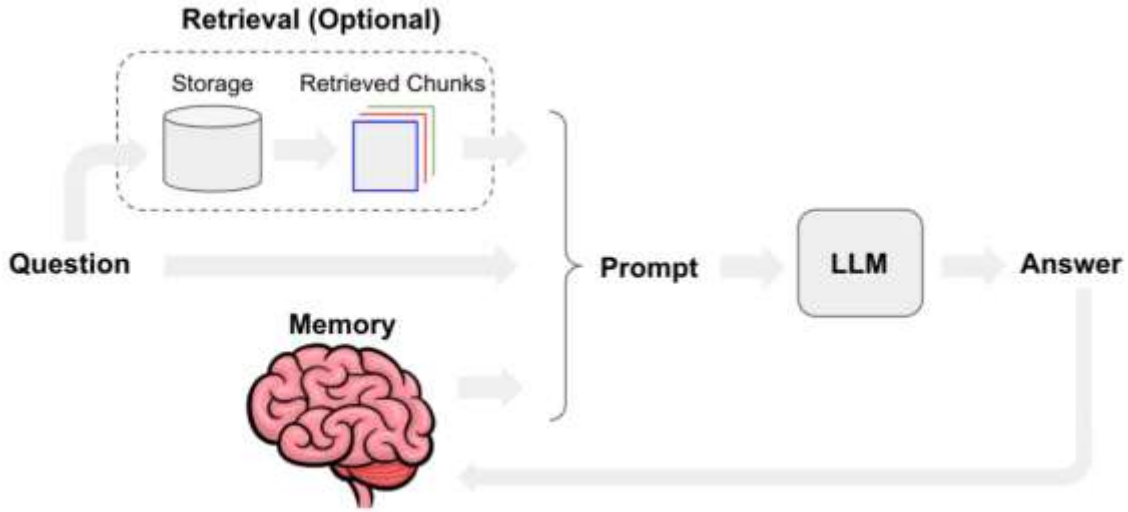
第十章　会話式Langchainベースのチャットボット実戦

Chapter 5では、質問に答えるアプリケーションを開発しました。ただし、これは LLM の非常に基本的なアプリケーションにすぎません。このChapterでは、その後の学習内容を組み合わせて、記憶して会話できる、よりリッチなチャット アプリケーションを開発します。

チャットボット（Chatbot）は、LLMとLangChainの主要なユースケースの一つです。多くの人々が、大規模言語モデルを学び、LangChainを学ぶのは、より良く、ユーザーの意図を理解できるチャットボットを開発するためです。チャットボットの主要な特徴は、長時間にわたる対話が可能であり、ユーザーが知りたい情報にアクセスできる点です。



第十章　会話式Langchainベースのチャットボット実戦

➢　実践プロジェクトの説明

図の通り、チャットボットの設計プロセスにおける主要なコンポーネントは以下の通りです：

チャットモデル：これは対話の基盤となり、自然な対話スタイルに重点を置いています。LangChainの関連ドキュメントにあるサポートされているチャットモデルのリストを参考にできます。大規模言語モデル（LLM）もチャットボットに応用可能ですが、専用のチャットモデルは対話シーンにより適しています。

プロンプトテンプレート：デフォルトのメッセージ、ユーザー入力、過去のやり取り、または検索時に必要なコンテキストを統合するのに役立ちます。

メモリ：これは、ユーザーとの以前のやり取りをボットが記憶することを可能にし、対話の一貫性を向上させます。

リトリーバー：特定の領域の知識を提供する必要があるボットに特に適したオプションのコンポーネントです。

第十章　会話式Langchainベースのチャットボット実戦

➢　実践プロジェクトの説明

このチャットボットの実装プロセスでは、アジャイル開発の原則に従います。まず、基本バージョンのボット開発に集中し、チャットができるという最も重要な機能を実現します。その後、段階的に機能を追加していきます。例えば、企業の知識ベースに基づいて検索できる機能を追加します。

第1歩: LangChainのConversationChainを利用して、最も基本的なチャット対話ツールを実現します。

第2歩: LangChainのメモリ機能を使い、このチャットボットがユーザーの過去の発言を記憶できるようにします。

第3歩: LangChainの検索機能を活用して、内部ドキュメント資料を統合します。これにより、チャットボットは自身の知識に加えて、業務フローに基づいた専門的な回答を提供できるようにします。

第4歩: 便利なツールを使用して、このチャットボットをオンラインでデプロイし、公開します。

第十章　会話式Langchainベースのチャットボット実戦

➢ OpenAI API vs ローカルモデル

大規模言語モデルはChatGPTだけではありません。OpenAIのAPIを使用するのは確かに便利で効率的ですが、他のモデル（例えば、オープンソースのLlama）を使用したい場合、または自分の機械でゼロから新しいモデルをトレーニングし、LangChainで自分のモデルを使用したい場合、どうすればよいでしょうか？

大規模モデルのファインチューニング（またはチューニング）、事前学習、再学習、さらにはゼロからのトレーニングについては、かなり大きなテーマです。これには十分な知識と経験だけでなく、大量のコーパスデータ、GPUハードウェア、そして強力なエンジニアリング能力も必要です。

これまでのチュートリアルではOpenAIのAPIを参考にしてきましたが、補足として、本章の開発ケースではローカルモデルを使用してみましょう。

第十章　会話式Langchainベースのチャットボット実戦

➢　大規模言語モデルの歴史

言語モデルの発展を理解するためには、その歴史を知ることが有益です。

Googleの2018年の論文「Attention is All You Need」で提案されたTransformerアーキテクチャは、AIの進展を促しました。Transformerは事前学習モデルの基盤となるアーキテクチャで、これに基づく大規模言語モデルは「基盤モデル」と呼ばれます。

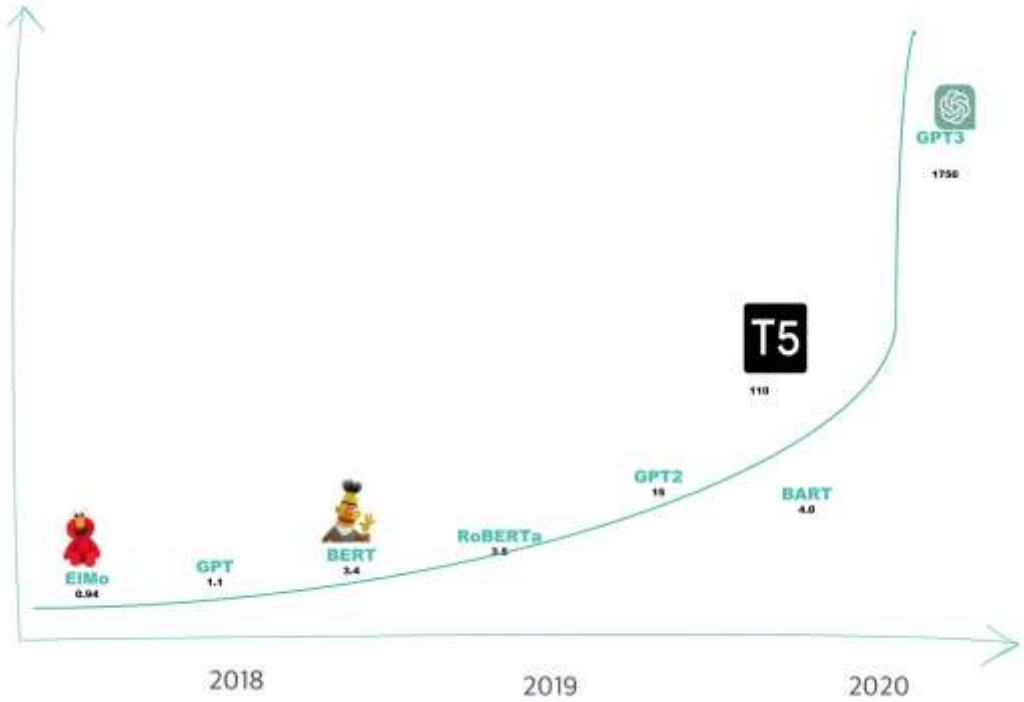
このプロセスで、モデルは語彙、文法、文の構造、文脈情報を学習し、これが感情分析やテキスト分類、質問応答システムなどの下流タスクに役立ちます。

初期の事前学習モデルとしてBERTは非常に代表的で影響力があり、その後、多くの大規模事前学習モデルが登場し、自然言語処理（NLP）の新時代を迎えました。これにより、翻訳やテキスト要約、チャット対話などの多くの問題が解決されました。

第十章　会話式Langchainベースのチャットボット実戦

➢　大規模言語モデルの歴史

もちろん、現在の事前学習モデルの傾向は、パラメータが増え、モデルがどんどん大きくなっており、1回のトレーニングに数百万ドルの費用がかかることもあります。これほどの高額なコストとリソースの消費は、世界のトップ企業だけが負担できるものであり、一般的な学術機関や大学がこの分野で科学的な突破口を開くのは難しくなっています。このような現象は、一般の研究者から批判されるようになっています。



第十章　会話式Langchainベースのチャットボット実戦

- **事前学習 vs ファインチューニングのモデル**

とはいえ、大規模事前学習モデルはエンジニアにとって大変な福音です。なぜなら、事前学習された大モデルが習得したセマンティック情報や言語知識は、下流タスクに非常に容易に転移できるからです。NLPアプリケーションのユーザーは、モデルのヘッドや一部のパラメータを自身のニーズに応じて適応的に調整することができ、これには通常、比較的小規模なラベル付きデータセットでの監視学習が含まれ、モデルが特定のタスクの要求に適応します。

これが事前学習モデルのファインチューニングです。ファインチューニングのプロセスは、ゼロからモデルをトレーニングするよりもはるかに速く、必要なデータ量も少なくて済むため、エンジニアとしては、さまざまなNLPソリューションをより効率的に開発・展開することが可能です。

第十章　会話式Langchainベースのチャットボット実戦

➢　事前学習 vs ファインチューニングのモデル

「具体的なタスク」は「具体的な分野」とも言えます。要約すると：

事前学習：大規模なラベルなしデータでモデルをトレーニングし、自然言語の基本的な表現や意味知識を学習させます。
ファインチューニング：事前学習モデルを基に、特定の分野やタスクに合わせてモデルを微調整します。これにより、オープンソースモデルを分野特化型にカスタマイズすることが可能です。

この事前学習+ファインチューニングのアプローチには明らかな利点があります。まず、事前学習モデルは汎用的な言語知識を下流タスクに移転でき、コーパスの収集やゼロからのトレーニングを不要にします。次に、ファインチューニングにより特定のタスクに迅速に適応でき、モデルのデプロイが簡単になります。最後に、このアーキテクチャは高いスケーラビリティを持ち、様々なNLPタスクに容易に適用できるため、実際のアプリケーションでの利用が大幅に向上します。

第十章　会話式Langchainベースのチャットボット実戦

➢ Ollama Toolkits

ローカルで大規模モデルを使用する際には、ハードウェアとして高性能なGPUが必須であるだけでなく、LangChain以外のオープンソースツールも使用する必要があります。ここではOllamaを例に挙げます。Ollamaは強力なツールで、ローカルでさまざまな大型言語モデルを実行できるようにします。

まず、LangChainライブラリからOllamaクラスをインポートする必要があります。

このインポート文は通常、他のインポート文と一緒にPythonファイルの先頭に置きます。

```
from langchain_ollama import Ollama

from langchain.chains import ConversationChain
```

第十章　会話式Langchainベースのチャットボット実戦

➢ Ollama Toolkits

次に、Ollamaモデルのインスタンスを初期化する必要があります。これは通常、メインプログラムのロジック内で行います。

```
llm = Ollama(model="llama3.1:70b")
```

ここでは、「llama3.1:70b」が使用するモデルの名前です。必要に応じて、他のOllamaがサポートしているモデルに変更できます。初期化が完了すると、このllmインスタンスを使用してテキストを生成したり、質問に回答したりできます。

```
response = llm("What is the capital of Japan?")
print(response)


>”The capital of Japan is Tokyo."
```

第十章　会話式Langchainベースのチャットボット実戦

➢ Ollama Toolkits

最終的なプロジェクトでは、OllamaモデルをLangChainの対話チェーンに統合する必要があります。

```
from langchain_ollama import Ollama
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

llm = Ollama(model="llama3.1:70b")

memory = ConversationBufferMemory()
conversation = ConversationChain(
    llm=llm,
    memory=memory,
    verbose=True
)
```

これで、conversation.predict()メソッドを使用して対話の応答を生成できるようになります。

第十章　会話式Langchainベースのチャットボット実戦

➢ Ollama Toolkits

このプロセスには以下の注意事項があります。

- Ollamaを使用する前に、ローカルにOllamaサービスをインストールして実行していることを確認してください。
- 異なるモデルは異なる性能とリソースの要件を持つ可能性があるため、ハードウェアの構成に応じて適切なモデルを選択してください。
- Ollamaモデルはローカルで動作するため、インターネット接続やAPIキーは必要ありませんが、応答時間はクラウドサービスのAPIより少し長くなる場合があります。

これらの手順を経て、OllamaをLangChainプロジェクトに統合し、チャットボットに強力な言語モデルサポートを提供しました。

第十章　会話式Langchainベースのチャットボット実戦

➢　ナレッジベースを統合する

検索強化生成（RAG）は、チャットボットが特定の知識ベースに基づいて質問に回答できる強力な技術です。

このセクションでは、プロジェクトにRAG機能を実装し、ローカル知識ベースをロードする方法を示します。

まず、RAGに必要な追加コンポーネントをインポートする必要があります。

```
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.document_loaders

import TextLoader from langchain.text_splitter
import CharacterTextSplitter
from langchain.chains import ConversationalRetrievalChain
```

第十章　会話式Langchainベースのチャットボット実戦

➢ ナレッジベースを統合する

次に、知識ベースの文書をロードし、それを小さなブロックに分割してトークンとして扱い、Embeddingsを生成して保存する必要があります。

```
def load_documents(file_path):
    loader = TextLoader(file_path)
    documents = loader.load()
    text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
    texts = text_splitter.split_documents(documents)
    return texts

texts = load_documents("あなたの文書のPath")
```

その後、効率的に検索するためのベクトル情報を格納する変数を作成する必要があります。

```
def create_vector_store(texts):
    embeddings = HuggingFaceEmbeddings()
    vectorstore = FAISS.from_documents(texts, embeddings)
    return vectorstore

vectorstore = create_vector_store(texts)
```

第十章　会話式Langchainベースのチャットボット実戦

➢ ナレッジベースを統合する

現在、言語モデル、ベクトルストレージ、および対話履歴を組み合わせたConversationalRetrievalChainを作成できます。

```
def load_chain(vectorstore):
    llm = Ollama(model="llama2")
    memory = ConversationBufferMemory(memory_key="chat_history", return_messages=True)
    chain = ConversationalRetrievalChain.from_llm(
        llm=llm,
        retriever=vectorstore.as_retriever(),
        memory=memory
    )
    return chain

chain = load_chain(vectorstore)
```

最後に、このチェーンを使用して知識ベースに基づいた回答を生成できます。

```
response = chain({"question": "君の質問こと"})
print(response['answer'])
```

第十章　会話式Langchainベースのチャットボット実戦

➢ ナレッジベースを統合する

この部分にもいくつか注意すべき点がありますので、簡単に列挙します。

- 適切なテキスト分割サイズの選択は、RAGの性能にとって非常に重要です。
- ベクトルストレージの作成には時間がかかる場合があり、特に大規模な文書ではそうです。
- 知識ベースの文書が最新かつ関連性があることを確認し、正確な回答を提供できるようにします。
- RAGは文書検索を必要とするため、応答時間が長くなる可能性があります。

これらのステップを経て、チャットボットにRAG機能を実装し、特定の知識ベースに基づいて質問に回答できるようにしました。これにより、回答の関連性と正確性が大幅に向上しました。

第十章　会話式Langchainベースのチャットボット実戦

➢ Streamlitを使ってGUIを実装する

まず、Streamlitをインポートし、ページの設定を行う必要があります。

```
import streamlit as st

st.set_page_config(
    page_title="AI Chatbot",
    page_icon=":robot_face:",
    layout="wide"
)
```

第十章　会話式Langchainベースのチャットボット実戦

➢ Streamlitを使ってGUIを実装する

Streamlitは、st.chat_message()とst.chat_input()関数を提供しており、これらを使って簡単にチャットインターフェースを作成することができます。

```
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

if prompt := st.chat_input("What's on your mind?"):
    st.session_state.messages.append({"role": "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)
```

上記のコードの最初の部分はユーザーの履歴を表示し、第二の部分はユーザーの入力を処理する部分です。

第十章　会話式Langchainベースのチャットボット実戦

➢ Streamlitを使ってGUIを実装する

AIの応答を得た後は、それをチャットインターフェースに追加することができます。

```
with st.chat_message("assistant"):
    with st.spinner("Thinking..."):
        response = chain({"question": prompt})
        st.markdown(response['answer'])
st.session_state.messages.append({"role": "assistant", "content":
response['answer']})
```

ここでの role、assistant、および content についての説明は、前のコースで既に触れられていますので、理解することに注意してください。

第十章　会話式Langchainベースのチャットボット実戦

➢ Streamlitを使ってGUIを実装する

次に、ページを適化しましょう。サイドバーを使用して、いくつかのコントロールオプションを追加することができます。

```
with st.chat_message("assistant"):
    with st.spinner("Thinking..."):
        response = chain({"question": prompt})
        st.markdown(response['answer'])
st.session_state.messages.append({"role": "assistant", "content":
response['answer']})
```

列レイアウトを使用することで、ページのコンテンツをより良く整理することができます。

```
col1, col2 = st.columns([2, 1])
with col1:
    st.title("Chat with AI")
with col2:
    st.subheader("Information")
    st.info("This is an AI assistant powered by Ollama and LangChain.")
```

第十章　会話式Langchainベースのチャットボット実戦

➢ Streamlitを使ってGUIを実装する

Streamlitは、ボタンやファイルアップロードなど、さまざまなインタラクティブな要素を提供しています。

```
if st.button("Clear Chat"):
    st.session_state.messages = []

uploaded_file = st.file_uploader("Choose a file")
```

CSSを使用して、Streamlitの外観をカスタマイズすることができます。

```
st.markdown("""
<style>
.stApp {
    background-color: #f0f0f0;
}
</style>
""", unsafe_allow_html=True)
```

第十章　会話式Langchainベースのチャットボット実戦

➢ Streamlitを使ってGUIを実装する

Streamlitの session_state を使用すると、ページのリロード間で状態を保持することができます。

```
if "messages" not in st.session_state:
    st.session_state.messages = []
```

最後に、次のコマンドを使用してStreamlitアプリを実行できます：

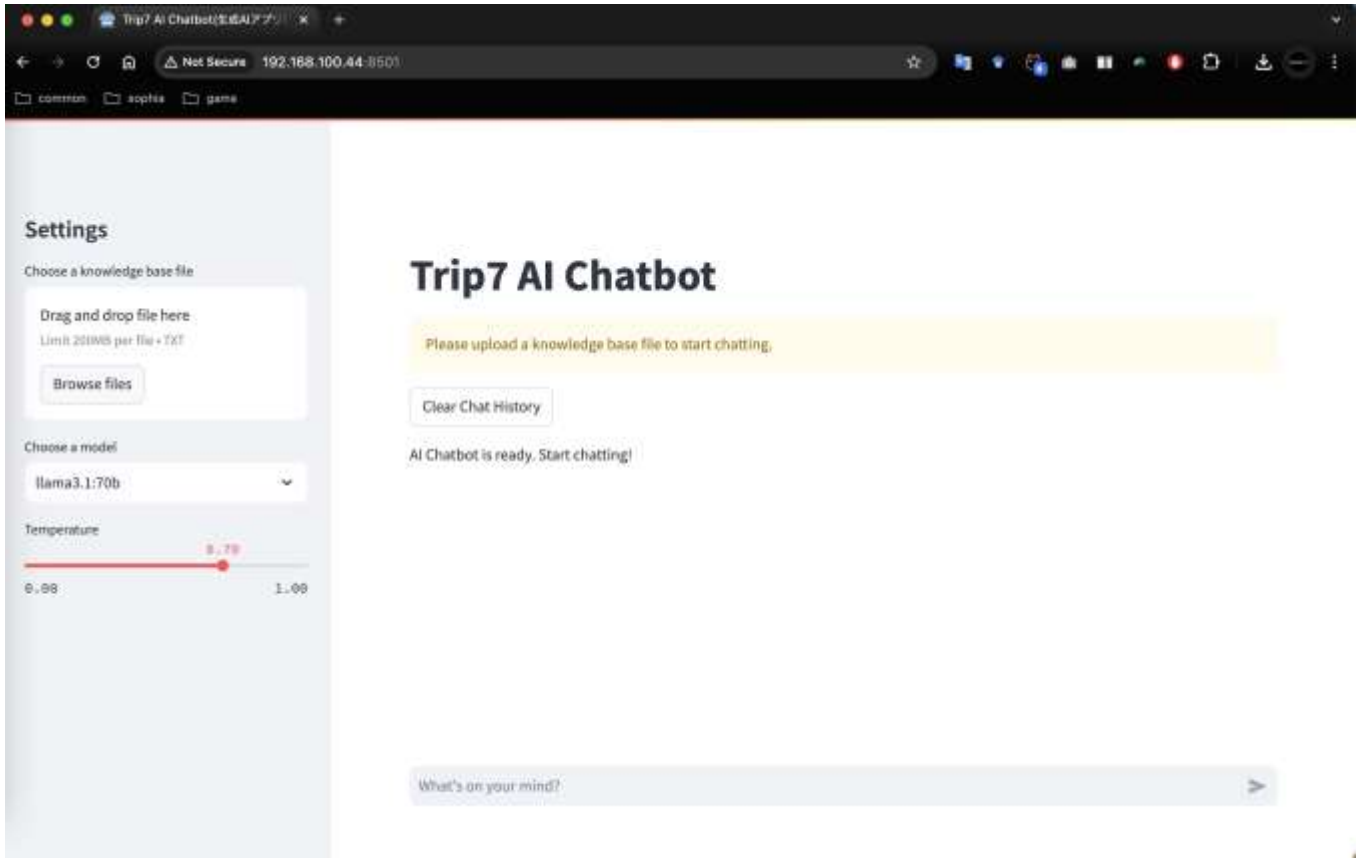
```
$streamlit run your_application_name.py
```

これらのステップを通じて、機能的で魅力的なチャットボットのインターフェースを作成しました。StreamlitのシンプルなAPIにより、複雑なWebアプリの構築が容易になり、コア機能の実装に集中することができます。

第十章　会話式Langchainベースのチャットボット実戦

➢ Streamlitを使ってGUIを実装する

最後に、アプリケーションの画面は以下の図のようになります。



第十章　会話式Langchainベースのチャットボット実戦

➢　まとめ

私たちのチャットボットは基本的に完成しました。多くの機能を備えており、その中にはモデル自体から得られるものもあります。たとえば、世界知識、要約、会話などです。それに加えて、記憶機能やドキュメントの検索機能も備えています。基本的なプロンプトとLLMの他に、記憶と検索はチャットボットの核心的なコンポーネントです。これらの機能により、チャットボットは過去のインタラクションを思い出すだけでなく、最新の特定分野の情報を提供することができます。

このチャットボットの構築過程で、LangChainの対話モデル、プロンプトテンプレート、記憶の実装、さらに検索機能とRAG機能の実装を再確認しました。

さらに、実用的なWeb UIツールとして「Streamlit」も紹介しました。Streamlitは、データアプリケーション、ダッシュボード、および可視化のために設計されています。さまざまなウィジェットを提供し、ユーザーがデータやモデルとインタラクションできるようにします。非常にPythonicであり、つまり使用方法が非常に自然で、Pythonに慣れている人にとっては直感的です。