Welcome to Part 6 of our course, "Building LLM-based Web Applications."

---

In this part, we will focus on building a variety of LLM-based web applications.

Our journey will begin with creating task-specific AI assistants that can handle specialized tasks, ex. Culinary tasks, marketing, customer assistant, etc.

Next, we'll develop a simple AI chatbot to understand the basics of conversational interfaces.

Moving forward, we'll explore the construction of RAG (Retrieval Augmented Generation)-based AI chatbots, which combine retrieval mechanisms with generation capabilities for enhanced responses.

Finally, we'll investigate building agent-based AI chatbots, which offer more sophisticated and autonomous interaction capabilities.

By the end of this part, you'll have hands-on experience in developing these diverse and powerful AI tools.

---

For the development of LLM-based web applications, I will use a combination of Python backend and Python frontend frameworks

On the backend side, for the demonstration purpose, the hardware and computation constraints, we will leverage the OpenAI API for GPT-3.5 as our large language model, orchestrate our LLM with LangChain, and use FAISS as our vector database. Additionally, we will integrate external tools such as Wikipedia and Tavily to enrich our applications.

On the front-end side, we will use Gradio to create interactive user interfaces for our web-applications.

This combination of powerful backend and frontend technologies will enable us to build quick, efficient and user-friendly AI applications.

---

Let’s first explore building task-specific AI Assistants

Task-specific AI assistants operate by leveraging large language models (LLMs) trained on publicly available, non-private data.

When a user makes a specific request related to a particular task, the AI assistant utilizes a customized prompt tailored to that request.

The LLM processes this input and generates an appropriate response, which is then delivered back to the user.

This interaction enables the AI assistant to provide precise and relevant answers, enhancing the user's experience and efficiency in handling specialized tasks.

Through this mechanism, we can build intelligent systems that are highly effective in addressing targeted needs.

---

Task-specific AI assistants can be developed for a wide range of applications to cater to various needs and industries. In this section we will learn some examples of task-specific AI assistant applications:

* **Culinary AI Assistant:** Helps users with recipes, cooking tips.
* **Marketing AI Assistant:** Assists in creating marketing content,
* **Customer AI Assistant:** Provides customer support, answers queries,
* **SQL-querying AI Assistant:** Assists with writing and optimizing SQL queries for database management.
* **Travel AI Assistant:** Provide travel or specialties recommendations.
* **Summarization AI Assistant:** Summarizes lengthy documents, articles, and reports.
* **Interview AI Assistant:** Prepares users for interviews by providing common questions and feedback.

These examples illustrate the versatility and potential of task-specific AI assistants in enhancing productivity and providing specialized support in various fields.

---

Before we jump into a specific AI assistant, here is a general code snippet for creating an AI assistant using a large language model (LLM) like GPT-3.5-turbo. This example demonstrates how to load environment variables, initialize the chat model, define the prompt template, create the prompt, and get the response from the model.

This script outlines the fundamental steps needed to set up an AI assistant. By customizing the prompt template, you can tailor the assistant to various specific tasks, making it a versatile tool for diverse applications.

---

Let’s talk about the Culinary AI Assistant.

Let’s see how a Culinary AI Assistant works. This AI assistant, powered by the OpenAI GPT-3.5-turbo API, LangChain, Wikipedia Search Tool, and Gradio, helps users in cooking by providing detailed recipes for various dishes.

The user can select a dish from a dropdown menu, such as Pizza Margherita, Spaghetti Carbonara, or Lasagna. Once a dish is selected and the number of persons is specified, clicking the button labeled ‘Tell me about the dish’s ingredients, recipe & interesting stories’ prompts the assistant to provide tailored ingredients and recipes. Additionally, it offers interesting stories and facts related to the dish, enhancing the user's culinary experience with educational content.

This assistant is designed to make cooking more accessible and enjoyable, offering precise and relevant information to help users create delicious meals while also learning about the history and cultural significance of the dishes they prepare.

---

**Parameterizing Prompt**

To dynamically generate a recipe based on user inputs, such as the selected dish and the number of persons, let’s see how we can parameterize prompts. We will use a prompt template like this:

“Present the recipe of {dish} for {no\_per} persons. Return the response in 2 parts: Ingredients and Recipe. For the Ingredients part, use the format ingredient name: ingredient quantity. For the Recipe part, make it clearly step-by-step.”

This prompt template dynamically incorporates user-specified parameters, such as the dish name and the number of servings. This approach allows the model to provide a tailored response that includes a list of ingredients and a step-by-step recipe, ensuring the instructions are clear and relevant to the user's needs.

---

**Search Tool + Parameterized Prompt**

Next, we will learn how to combine a search tool with parameterized prompts to enhance the Culinary AI Assistant's capabilities. The idea is to search for interesting facts and stories related to the dish based on user inputs.

First, using the dish name, we ask the Wikipedia tool to return search results with the prompt ‘origin story of [dish].’

Then, we formulate another prompt, saying:

“Present the 3 most interesting stories or facts about {dish}, one sentence for each, using this search data: {wiki\_result}.”

With this prompt enriched by the Wikipedia search results, we can ask our LLM to generate interesting stories or facts about the dish. By integrating external data sources such as Wikipedia, we enrich the assistant's responses with fascinating stories or facts about the selected dish.

By combining parameterized prompts with a search tool, the AI assistant can provide a richer and more informative user experience, offering both practical and engaging content.

---

**To Sum Up**

This app allows us to learn two strategies:

* **Firstly, parameterizing efficiently prompt based on interactive user-input interface**. This capability ensures that the assistant's output is highly relevant to the user's needs.
* **Secondly, combining parameterized prompt with search tool like Wikipedia**: Integrating search tools like Wikipedia with parameterized prompts enables the AI assistant to enrich its responses with additional information.

---

Let’s discover Marketing AI Assistant.

**Marketing AI Assistant**

The Marketing AI Assistant is designed to help create compelling marketing content for products, leveraging the power of OpenAI GPT-3.5-Turbo API, LangChain, and Gradio. This application assists marketers in generating persuasive text tailored to their product features, target clients, and desired word count.

The interface allows the user to input:

* **Product**: such as, a new line of organic skincare products.
* **Features**: such as, Natural Ingredients, Free from Harmful Chemicals, Environmentally Sustainable, Cruelty-Free.
* **Target clients**: such as, Health-conscious consumers.
* **Word Count**: such as, Adjustable between 100 and 300 words.

**Then by clicking on the button “Generate”**

The assistant produces a marketing message, ensuring it aligns with the specified features and appeals to the target audience. Here's an example output: …

Another style of this message could be: …

---

To achieve these results, I built the app based on two prompts.

For the first prompt, the interactive interface allows the user to specify details about the product, its features, target clients, and desired word count. The code then dynamically incorporates the user inputs into the prompt, ensuring the generated message is specific and relevant to the product and audience.

---

**Second Request**

For the second prompt, we use the previously generated marketing message as a dynamic input parameter and ask our model to generate a new version in a more engaging style, incorporating an upbeat tone and emojis. This demonstrates the assistant's ability to reformat and enhance content based on additional user specifications.

---

**Combining All Responses**

In this final step, we combine the original marketing message with the new style to create a comprehensive output that showcases different presentation formats for the same content.

This approach provides versatility and allows users to choose the most suitable format for their marketing needs.

---

**All-in-One Prompt**

Alternatively, instead of using two separate prompts, we can generate two distinct marketing messages for a product using a single all-in-one prompt.

First, we dynamically incorporate the user inputs into the prompt instruction, then precisely ask the model to generate two messages with different styles: one in a professional tone as a paragraph, and the other in a more engaging tone using main points and emojis.

By using this approach, the Marketing AI Assistant can provide versatile outputs in one go, making it efficient and effective for users to obtain multiple styles of content from a single input.

---

**To Sum Up**

This app provides two strategies:

1. **First, creating a Chain of Requests/Prompts**:
   * The result of the first prompt becomes the input for the second one, allowing us to create more tailored and refined responses.
2. **Second, creating Multiple Responses in Different Styles Based on a Single Prompt**:
   * Enhancing flexibility and adaptability for various needs.

By utilizing these capabilities, users can efficiently generate high-quality, versatile content that is well-suited to different contexts and audiences. This app showcases the power of prompt engineering to streamline and enhance the content creation process.

---

Now, let’s see another use-case, Customer AI Assistant

**Customer AI Assistant**

The Customer AI Assistant is designed to analyze sentiment in customer feedback and generate appropriate responses. This assistant helps businesses effectively manage customer relations by automating the sentiment analysis and response generation process.

* **Given a customer feedback**: For example, "I'm thrilled with my camera purchase! The image quality is exceptional, and it's remarkably user-friendly. This camera exceeded my expectations, and its portability is a bonus. It's a must-have for any photography enthusiast."
* **After clicking the button "Generate"**:
  + The app analyzes the sentiment of the feedback and determines it to be "POSITIVE."
  + In addition, the model generates a response email based on the analyzed sentiment.

This streamlined process allows businesses to efficiently handle customer feedback, ensuring timely and appropriate responses that enhance customer satisfaction and loyalty.

---

**Requesting Output in a Specific Format**

In this app example, the prompt is configured to process customer feedback, determine the sentiment, and generate an email response.

The assistant formats the output as a JSON object, making it easier to parse and use programmatically.

---

**What We Have Learned from this app**

This app allows us to

* First, ask different tasks within a single prompt: Streamlining the process by handling multiple requests simultaneously
* Then, format outputs in a specific format, e.g. JSON: Making it easy to integrate into applications

By combining these capabilities, the app enhances efficiency and flexibility, allowing for seamless integration into existing workflows and systems. This approach ensures that users can quickly generate and utilize structured outputs for a variety of tasks.

---

**SQL-Querying AI Assistant**

Now let’s explore another interesting AI assistant. SLQ-querying AI Assistant

The motivation to build the SQL-Querying AI Assistant stems from the need to make database querying accessible to non-technical users. This app bridges the gap between natural language and SQL, helping users interact with databases by generating SQL queries based on natural language questions.

The app is powered by the OpenAI GPT-3.5-Turbo API, LangChain, SQLite, and Gradio.

**User Interface:**

1. **Database Selection**: The app allows users to choose a database, such as Chinook.db.
2. **Query Input**: Users can ask a question about the database, like "How many employees are there?"

**Response Process:**

1. **Query Generation**: The assistant converts the natural language question into an SQL query.
2. **Execution and Interpretation**: The generated SQL query is executed against the selected database, and the result is interpreted and presented in natural language.

This AI assistant streamlines the process of interacting with databases, making it accessible even to those who may not be proficient in SQL. It enhances productivity by automating query generation and result interpretation.

---

**SQL-Querying AI Assistant - Complete Workflow**

To implement the SQL-Querying AI Assistant, we will follow the process from receiving the database name and the user's natural language question to generating a corresponding SQL query and providing a natural language response based on the query results. The main function in our app is predict\_query, and implementing this function consists of five main steps:

**Main Function: predict\_query**

1. **Initialize the Language Model and Connect to the Database**:
   * This step involves setting up the OpenAI GPT-3.5-Turbo API and establishing a connection to the specified database.
2. **Get Schema Function**:
   * Retrieve the schema information from the selected database to understand its structure and tables.
3. **Create Chain to Generate SQL Query**:
   * Use the schema and the user's natural language question to create an SQL query.
4. **Create Final Chain**:
   * Generate a natural language response using the SQL query and its results.
5. **Invoke Method**:
   * Execute the generated chains on the user question to produce the SQL query and the natural language response.

This complete implementation showcases the SQL-Querying AI Assistant's ability to convert natural language questions into SQL queries and provide comprehensive responses, making database interactions more accessible and efficient.

---

**What We Have Learned from this application**

1. **Natural Language to SQL Conversion**:
   * The assistant can effectively convert natural language questions into SQL queries.
   * This capability makes it easier for users who may not be proficient in SQL to interact with databases and retrieve the information they need.
2. **Schema Utilization for Accurate Query Generation**:
   * By retrieving and utilizing the database schema, the assistant ensures that the generated SQL queries are accurate and relevant to the database structure.
   * This process enhances the reliability and precision of the queries, reducing errors and improving the efficiency of database interactions.
3. **Integration and Response Generation**:
   * The assistant can execute the generated SQL queries and produce natural language responses based on the query results.
   * This feature makes the data retrieval process more user-friendly and accessible, providing clear and understandable results that can be easily integrated into applications or used for decision-making.

---

Now, let’s have a look at more ideas for AI assistant applications. I will provide you with ideas and an introduction to the user interface, without detailed instructions for the inner components. As always, you have the source code to play with, so feel free to explore it on your own.

---

Let’s take a look at the Travel AI Assistant

The "Travel AI Assistant" app is designed to help users plan their trips by providing travel advice and recommendations. Let’s see general idea of how it works and what the expected outputs are:

**How It Works:**

1. **User Input**: The user specifies their travel destination and can adjust a slider to indicate their number of top interests
2. **AI Processing**: Powered by OpenAI’s GPT-3.5-Turbo API, along with Langchain and DuckDuckGoSearch tools, the app processes the user's input to generate relevant travel advice.

**Expected Outputs:**

* **Tourist Places**: The app suggests a list of must-visit tourist attractions at the specified destination. For example, if the destination is France, it might list the Eiffel Tower, the Louvre Museum, the Palace of Versailles, Mont Saint-Michel, and the French Riviera.
* **Famous Foods**: The app also provides recommendations for famous local dishes that the user should try. In the case of France, it might include dishes like croissants, Coq au Vin, Ratatouille, Bouillabaisse, and Crème Brûlée.

**Summary:**

To sum up, the Travel AI Assistant helps users by offering curated travel recommendations tailored to their interests and destination choices. This helps users quickly identify key attractions and local culinary experiences, making their travel planning more efficient and enjoyable.

---

Now let’s look at Summarization AI Assistant

The "Summarization AI Assistant" is an application designed to generate concise summaries of long texts. Let’s see general idea of how it works and what the expected outputs are:

**How It Works:**

1. **User Input**: The user inputs a lengthy text into the provided text box.
2. **AI Processing**: Using the OpenAI GPT-3.5-Turbo API, along with Langchain and Gradio, the app processes the input text to extract the most critical points and condense the information.

**Expected Outputs:**

* **Concise Summary**: The output is a summarized version of the input text. It includes the essential points, making the original information easier and quicker to understand. For example, the text about climate change is summarized into bullet points highlighting the main threats, causes, and necessary actions to address climate change.

**Summary:**

The Summarization AI Assistant helps users by transforming long and detailed texts into brief, digestible summaries. This assists in quickly understanding the core message of the content, which is particularly useful for reviewing large volumes of information efficiently.

---

Now let’s look at Interview AI Assistant

The "Interview AI Assistant" is an application designed to help users prepare for job interviews by generating relevant interview questions and answers tailored to specific job positions and programming languages. Let’s see the general idea of how it works and what the expected outputs are:

**How It Works:**

1. **User Input**: The user selects a job position (e.g., AI Engineer) and a programming language (e.g., Python). They can also specify the number of question-answer pairs they want to generate.
2. **AI Processing**: Utilizing the OpenAI GPT-3.5-Turbo API, along with Langchain and Gradio, the app processes this information to generate a list of relevant interview questions and corresponding answers.

**Expected Outputs:**

* **Interview Questions**: A list of interview questions related to the job position and programming language chosen by the user. For instance, questions might cover fundamental concepts, practical applications, and specific technical skills.
* **Detailed Answers**: The app provides comprehensive answers to each of the generated questions, helping the user understand the key points that should be covered in their responses.

**Summary:**

The Interview AI Assistant aids users in preparing for job interviews by generating relevant and customized interview questions and answers. This tool helps candidates anticipate potential questions and practice their responses, thereby improving their confidence and readiness for actual interviews.

---

Now let’s talk about AI chatbot based on Large Language Model. Let’s start with a simple AI Chatbot.

---

This diagram illustrates the basic interaction between a user and an AI chatbot powered by large language models. The LLM, trained on publicly available, non-private data up to a specific point in time, processes the user's questions and generates relevant answers.

---

Now let's take a closer look at a simple AI chatbot demo. This demo showcases an AI chatbot powered by OpenAI's GPT-3.5-Turbo API, integrated with Langchain and Gradio for enhanced functionality.

In this example, the user asks, "What is climate change?" The chatbot responds with a detailed explanation, describing climate change as long-term alterations in temperature, precipitation, and other atmospheric conditions, primarily driven by human activities such as burning fossil fuels and deforestation.

This example highlights the chatbot's ability to provide comprehensive, informative answers to user queries, demonstrating its potential in facilitating user interactions and delivering accurate information efficiently.

---

Let’s see how we can simply implement the chat interface with Gradio.

On the left, we have a snippet of code defining the chat interface using the gr.ChatInterface function from the Gradio library. The first argument is the predict function, which generates responses for the user’s questions. The second argument can be example queries such as "What is climate change?" and "What are the benefits of renewable energy?" to guide users in their interactions.

On the right, we see the visual representation of the chat interface, where users can type their questions and receive responses. This setup demonstrates how easily Gradio helps us create a chat interface.

---

Now, let’s see how to implement the backend functions of our AI chatbot.

We will define two functions:

* **First Function**: This function creates the LLM chain, which encompasses the prompt, chat model, and memory that serves as the conversational history. The variables chain and memory are set as global variables.
* **Second Function**: The predict function receives the user message and history from the chat interface as inputs.

It then reconstructs the input as a dictionary with the key "question" and the value as the user message input.

It then calls the invoke method of the chain on the inputs to generate the response and updates the memory. Finally, the predict function returns the response.

This setup highlights how incorporating memory into the chatbot enables it to deliver more personalized and context-aware interactions, enhancing the user experience by remembering prior conversations.

---

Let’s now test to see whether our AI chatbot can remember things.

I first introduce myself, saying, “I’m Quang,” and the chatbot greets me warmly. When I later ask, "What is my name?" the chatbot correctly recalls and responds, "Your name is Quang."

This example demonstrates the chatbot's ability to retain and utilize information from previous interactions, enhancing the conversational experience. By maintaining context and remembering key details, the chatbot can provide more personalized and coherent responses, which is particularly valuable for extended or ongoing user interactions.

---

Before we explore a new type of AI chatbot: the RAG-based AI Chatbot, let's first discuss the limitations of simple AI chatbots.

These chatbots are typically trained on publicly available, non-private data up to a specific point in time. Consequently, they cannot provide answers to questions related to private information or private documents.

This limitation highlights the need for the integration of secure, private data handling mechanisms to enhance the chatbot's accuracy and personalization capabilities.

---

To handle the above limitation, let’s explore a new type of AI chatbot: the Retrieval-Augmented Generation (RAG)-based AI chatbot.

Unlike traditional AI chatbots, a RAG-based chatbot can access private data sources, such as text files, PDFs, tables, and Word documents. The diagram illustrates how the information retrieval process works: the AI chatbot, trained on publicly available, non-private data up to a certain point, can fetch relevant information from these external data sources to provide accurate private responses.

This approach addresses the limitations of standard chatbots by integrating private knowledge, thus enhancing the chatbot's ability to deliver precise and contextually relevant answers to user queries.

---

Building a RAG-based AI chatbot involves three key steps.

First, we generate a vector database, which involves converting textual data into vectors that the model can understand and process.

Second, we implement information retrieval, enabling the chatbot to fetch relevant data from various sources, such as documents, databases, and other repositories.

Lastly, we integrate augmented generation with a chat user interface (UI). This step combines the retrieved information with the AI model's capabilities to generate coherent and contextually accurate responses, which are then presented through a user-friendly chat interface.

These steps collectively ensure that the RAG-based AI chatbot can provide precise answers on private knowledge, leveraging both the pre-trained model and real-time information retrieval.

---

The first step in building a RAG-based AI chatbot is generating a vector database.

This process begins with loading data from various private data sources, such as text files, PDFs, tables, and Word documents.

Once the data is loaded, it undergoes vector embedding, where the textual information is converted into numerical vectors that the AI model can process.

These vectors are then stored in a vector database.

This step is crucial as it prepares the data in a format that facilitates efficient retrieval and processing during the chatbot's operations, enabling it to provide accurate and relevant responses based on the most current and specific information available.

---

The second step in building a RAG-based AI chatbot is information retrieval. When a user poses a question, it is first converted into an embedding, a numerical representation that captures the semantic meaning of the query.

This embedding is then used to perform a similarity search against the vector database we generated in the first step.

The similarity search identifies the top-k pieces of information, or chunks of text, that are most relevant to the user's query.

These retrieved pieces of information are then used by the AI chatbot to generate a precise and contextually appropriate response. This process ensures that the chatbot can provide answers based on the most relevant and current data available.

---

The final step in building a RAG-based AI chatbot is augmented generation combined with a chat user interface (UI).

Once the top-k retrieved information chunks have been identified through the similarity search, these chunks are used by the AI chatbot, which is powered by large language models (LLMs) trained on publicly available data.

The chatbot processes this information to generate accurate and contextually relevant answers to user questions.

These answers are then delivered to the user through a seamless chat interface, facilitating an interactive and user-friendly experience.

This integration ensures that the chatbot not only leverages its pre-trained knowledge but also incorporates up-to-date and specific information, providing more comprehensive and reliable responses.

---

Now let’s take a detailed look at the implementation of a RAG-based AI chatbot for a data source containing PDF files.

The process begins by importing necessary modules from the LangChain library, such as ChatOpenAI, FAISS, PromptTemplate, and PyPDFLoader.

1. **Loading and Splitting Documents**:
   * The first step involves creating chunks of text from documents loaded using PyPDFLoader.
   * These documents are then split into manageable chunks using CharacterTextSplitter.
2. **Storing Chunks in Vector Store**:
   * The text chunks are embedded and stored in a vector store using FAISS.
3. **Setting Up a Retriever**:
   * Next, we set up a retriever from the vector store to handle search queries.
4. **Initializing the OpenAI Model**:
   * We initialize the OpenAI model, defining a prompt template for the chatbot to use retrieved context for answering questions.
5. **Creating the LangChain**:
   * The LangChain chain is created, linking the context retriever, prompt, and the language model (LLM).
6. **Generating Responses**:
   * A function, make\_llm\_response, generates a response by invoking the RAG chain with the user's question.
   * We can start asking questions related to the document, like “What is the topic of this document?”
   * The function make\_llm\_response is called on this question to get the relevant response.

This detailed implementation showcases the integration of various components to build an effective RAG-based AI chatbot capable of providing contextually enriched responses by leveraging both pre-trained knowledge and custom information retrieval.

---

Now let’s see a RAG-based AI chatbot in action.

The demo showcases how the chatbot can interact with a PDF file, leveraging the power of OpenAI's GPT-3.5-Turbo API, Langchain, and Gradio.

In this example, a user uploads a PDF document containing a recipe for Margherita pizza.

The chatbot processes the document and successfully retrieves the context to answer the user's question, "What is the topic of this document?"

The chatbot responds accurately by identifying the document's topic as a recipe for Margherita pizza.

This demo illustrates the chatbot's capability to handle and interpret documents, providing users with precise and contextually relevant information based on the content of the uploaded files.

---

To extend the capacity of a RAG-based AI chatbot for different data types, we can leverage various data loaders from LangChain.

If our data source contains different document or data types, such as PDF, DOCX, TXT, webpage URLs, or CSV files, we can use the appropriate DataLoader from LangChain, like PyPDFLoader, UnstructuredWordDocumentLoader, TextLoader, WebBaseLoader, CSVLoader, etc.

These loaders enable seamless integration and processing of diverse data types within LangChain, enhancing data handling capabilities for developers.

---

Let’s summarize the key learnings from our session.

First, we gained an understanding of the basic pipeline of a Retrieval-Augmented Generation (RAG) system, which is crucial for building intelligent AI chatbots.

Additionally, we reviewed the source code needed to construct a simple RAG-based AI chatbot specifically designed for processing PDF documents.

Finally, we explored the various data loaders available in LangChain, which allow us to handle different types of data sources, including PDF, DOCX, TXT, webpage URLs, and CSV files.

These tools collectively enhance our ability to create versatile and robust AI Chatbot solutions.

---

Now let’s look at Agent-based AI Chatbot

--

What are other limitations of AI Chatbot,

While AI chatbots have made significant strides in recent years, they still have some notable limitations.

One challenge is their ability to access and process daily news. Since chatbots are typically trained on data available up to a certain point in time, they may not have the latest information on current events unless specifically designed to retrieve and process real-time data.

Another limitation is their capacity to perform complex mathematical calculations.

While they can handle basic arithmetic, more advanced mathematical operations often require additional programming and integration with specialized tools.

These limitations highlight the areas where ongoing improvements and integrations are necessary to enhance the functionality and reliability of AI chatbots.

---

Let’s see concrete examples of the limitations of AI chatbots.

The first example shows a chatbot wrongly calculating the result of 68 raised to the 0.36 power.

The second example, when asked about the winner of the men's singles at Roland Garros 2024, the chatbot responds that it does not have real-time information and suggests checking a reliable sports news source.

These examples illustrate that AI chatbots struggle with up-to-date information retrieval and complex mathematical computations, emphasizing the need for continuous improvements and integrations.

---

To handle these limitations of LLM-based AI chatbot, let’s explore another type of AI chatbot: agent-based AI chatbot, which leverages external tools to enhance its functionality.

Unlike traditional LLM-based chatbots, which rely solely on pre-trained data up to a certain point in time, agent-based AI chatbots can query external tools such as search engines and mathematical computation tools to retrieve real-time information and perform complex calculations.

When a user asks a question, the AI chatbot can determine if external tools are needed to generate an accurate response. It then sends a query to these tools, retrieves the necessary data, and integrates this information into its response back to the user.

This approach significantly expands the chatbot's capabilities, allowing it to provide more current and precise answers by accessing a broader range of resources.

---

Now let’s see how to implement LLM-based AI chatbot with Basic Math tool

---

It starts with defining the main function where the language model (GPT-3.5-Turbo) is initialized with conversational memory capabilities using ConversationBufferMemory.

The math tool is set up with the Tool.from\_function method, allowing the chatbot to perform mathematical calculations.

Then, the tools, including the math tool, are defined and initialized with the agent. The agent is configured to handle parsing errors, apply early stopping methods, and use conversational memory to maintain context throughout interactions.

Next, we implement predict function.

This function firstly ensures the agent is initialized if it is not already.

It then runs the agent with the user's message to generate a response, utilizing the integrated math tool for any necessary calculations.

This implementation demonstrates how adding a basic math tool can enhance the capabilities of an AI chatbot, allowing it to handle more complex queries involving mathematical computations effectively.

---

Now let’s evaluate the performance of our AI chatbot with the basic math tool.

For instance, when asked to calculate the result of 68 raised to the power of 0.36, the chatbot provides a precise answer of 4.567759279639116.

However, when requested to calculate the cosine of an angle of 15.78 degrees, it returns the incorrect value of -0.9974064764544136.

This indicates that our math tool is not adequate to handle all math problems.

To improve the math skills of our AI chatbot, we need to add more custom math functions.

---

Let's implement more customized math tools into an AI chatbot. The code example demonstrates the creation of a CosineDegreeTool class, which is designed to calculate the cosine of an angle given in degrees. The tool converts the angle from degrees to radians before computing the cosine using Python's math library. This customized tool is then integrated into the AI chatbot alongside the basic math tool. In the main function, the set of tools for the agent includes both the math\_tool and the newly created CosineDegreeTool. Let's re-check the math skills of our AI chatbot in the next demo.

---

For this new demo, when asked to calculate the result of 68 raised to the 0.36 power, the chatbot provides a precise answer of 4.567759279639116.

Additionally, with the integration of a customized cosine calculator, the chatbot can accurately compute the cosine of an angle of 15.78 degrees, returning a value of 0.9623129786416633.

This demonstration highlights how integrating specialized math tools can significantly enhance the functionality and accuracy of AI chatbots for math problems

---

Now let’s implement an LLM-based AI chatbot with search tools to address the limitation of not being able to access real-time information. First, we will implement the main function.

Start by initializing the chatbot using OpenAI's GPT-3.5-Turbo model and a search tool. The search tool, TavilySearchResults, is configured to return a maximum of one result. Note that to use TavilySearchResults, we need to have TAVILY\_API\_KEY set in the environment variables.

The prompt template is designed to make the chatbot a helpful assistant while optionally using external tools. Next, we create the agent using the function create\_openai\_tools\_agent, passing the chat model, search tools, and prompts as arguments. After that, an agent\_executor is created and returned from this main function.

Then, we implement the predict function, which ensures that the agent is initialized and able to invoke the search tool to fetch real-time data when needed. This integration allows the chatbot to provide up-to-date and relevant responses by leveraging external search capabilities, thereby enhancing its functionality and user experience.

---

Now let's evaluate the performance of our AI chatbot with the integrated search engine.

The interface demonstrates the chatbot's ability to retrieve up-to-date information, such as the winner of the 2024 French Open Men's singles.

By leveraging the capabilities of the OpenAI GPT-3.5-Turbo API, LangChain, Tavily, and Gradio, the chatbot successfully retrieves and displays the latest information.

This integration allows the chatbot to enhance its responses with current and accurate data, significantly improving the user experience by providing timely and relevant answers.

---

Well, let’s summarize the key points we have learned from developing Agent-based AI chatbot application.

We have gained an understanding of the limitations of LLM-based AI chatbots, particularly when it comes to handling complex math and search problems.

To address these challenges, we have leveraged specialized math tools and integrated external search tools, creating an agent-based AI chatbot that can effectively tackle both math and search-related queries.

This approach enhances the chatbot's functionality, enabling it to provide more accurate and relevant responses to user inquiries.

---

What we have learned from this part so far.

This part of the project has provided valuable insights into leveraging different frameworks to build LLM-based web application demos.

On the backend, we utilized Python, LangChain, the OpenAI API, FAISS, and various external tools like Wikipedia and Tavily.

For the frontend, we employed Gradio to create user-friendly interfaces.

We explored the development of four types of LLM-based applications: task-specific AI assistants, simple AI chatbots, RAG-based AI chatbots, and agent-based AI chatbots.

These demonstrations highlighted the versatility and capabilities of LLMs and the powerful tools offered by LangChain in creating sophisticated AI applications.