Intelligent Search Integration with LangChain

August 2024

Ivan Radonjic

Detailed Code Walkthrough and Results

This document provides a comprehensive walkthrough of the 'Intelligent Search Integration with LangChain' project, offering an in-depth analysis of each block of code. It delves into the purpose and functionality behind every line, while also highlighting the outcomes generated from each search operation, demonstrating the effectiveness and efficiency of the integrated framework.

```
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv())
```

```
from langgraph.graph import StateGraph, END
from typing import TypedDict, Annotated
import operator
from langchain_core.messages import AnyMessage, SystemMessage, HumanMessage, ToolMessage
from langchain_openai import ChatOpenAI
from langchain_community.tools.tavily_search import TavilySearchResults
```

Load the necessary libraries and environment configurations. The environment file securely stores my Tavily and OpenAI API keys, ensuring seamless access to the required services.

```
tool = TavilySearchResults(max_results = 2)
print(type(tool))
print(tool.name)
```

```
<class 'langchain_community.tools.tavily_search.tool.TavilySearchResults'>
tavily_search_results_json
```

The code snippet initializes the 'TavilySearchResults' tool, with a configuration to return a maximum of two results per search query. This is particularly useful for narrowing down search results to the most relevant ones, thereby enhancing the efficiency of the search process. Following the initialization, two print statements are executed to verify the tool's setup. The first print statement confirms that the tool object is an instance of the 'TavilySearchResults' class, which is part of the LangChain community tools. The second print statement outputs the tool's name, 'tavily_search_results_json', ensuring that the tool has been correctly instantiated and is ready for use in subsequent operations.

```python
class AgentState(TypedDict):
    messages: Annotated[list[AnyMessage], operator.add]

class Agent:
    def __init__(self, model, tools, system = ""):
        self.system = system
        graph = StateGraph(AgentState)
        graph.add_node("LLM", self.call_openai)
        graph.add_node("Action", self.take_action)
        graph.add_conditional_edges(
            "LLM",
            self.exists_action,
            {True: "Action", False: END}
        )
        graph.add_edge("Action", "LLM")
        graph.set_entry_point("LLM")
        self.graph = graph.compile()
        self.tools = {t.name: t for t in tools}
        self.model = model.bind_tools(tools)

    def call_openai(self, state: AgentState):
        messages = state['messages']
        if self.system:
            messages = [SystemMessage(content = self.system)] + messages  #prepend to list of messages
        message = self.model.invoke(messages)
        return {'messages': [message]}

    def take_action(self, state: AgentState):
        tool_calls = state['messages'][-1].tool_calls
        results = []
        for t in tool_calls:
            print(f"Calling: {t}")
            if not t['name'] in self.tools:
                print("\n...Bad tool name...")
                result = "Bad tool name, retry"
            else:
                result = self.tools[t['name']].invoke(t['args'])
            results.append(ToolMessage(tool_call_id = t['id'], name = t['name'], content = str(result)))
        print("back to the model!")
        return {'messages': results}

    #define conditonal edge
    def exists_action(self, state: AgentState):
        #take the last message
        result = state['messages'][-1]
        return len(result.tool_calls) > 0
```

AgentState Class: The AgentState class (TypedDict) stores the state of the agent. It contains a single key, ‘messages’, which is a list of messages exchanged between the agent and the LLM during execution.

Agent Class: The Agent class is the core of the framework, it gets initialized with a LLM, a set of tools and an optional system message, the class defines several methods for interacting with the LLM and tools around it.

Constructor: This method gets initialized with a StateGraph object, which defines the structure and flow in the decision-making process. It adds two primary nodes to the

graph, one node representing the LLM, called ‘LLM’ and it is linked to the ‘call_openai’ method. This node represents the step where the agent interacts with the LLM. The second node is named ‘Action’ and it is linked to the ‘take_action’ method, this node handles the execution of actions based on the LLM output.

This method also adds a conditional edge to the graph from ‘LLM’ to ‘Action’, whether this condition is met is determined whether an action exists, if not action, the graph ends. The graph is then compiled, and the tools are bound to the model for later use.

call_openai: This method sends a message to the LLM. If a system message is provided, it prepends it to the list of messages before invoking the model. The method returns the models response, which is added to the state.

take_action: The method retrieves the tool calls from the last message in the state. It iterated through each tool call, checks if the tool exists and invokes the appropriate tool. If the tool is not found, an error message is returned.

exists_action: This method checks whether there are any tool calls in the last message. It returns a boolean indicating whether an action should be taken, or the process should end.

```
prompt = """You are a smart research assistant. Use the search engine to look up information. \
You are allowed to make multiple calls (either together or in sequence). \
Only look up information when you are sure of what you want. \
If you need to look up some information before asking a follow up question, you are allowed to do that!
"""

model = ChatOpenAI(model="gpt-3.5-turbo")
abot = Agent(model, [tool], system=prompt)
```

The following segment of code establishes the foundational setup for your intelligent agent, the prompt is used to define the agent’s behavior. The prompt characterizes the model as a smart research assistant and outlines its operational guidelines. After the prompt is set, we initialize a model, specifically OpenAI’s gpt-3.5-turbo. We next create an instance of the agent class as ‘abot’, we pass in the previously define model, its necessary tools and the prompt as the initial system message. This setup is the cornerstone that will enable the agent to efficiently handle queries and provide accurate, relevant information based on the user's needs.

```
messages = [HumanMessage(content = "What is the weather in sf?")]
result = abot.graph.invoke({"messages": messages})
```

The segment of code above demonstrates how the initialized agent is used to process a query and obtain a result by invoking the state graph. We create a ‘message’ variable as a list containing a ‘HumanMessage’ object. This object represents the message from a user. The ‘result’ variable is assigned to the output from invoking the graph.

```
result['messages'][-1].content
```

```
'The current weather in San Francisco is sunny with a temperature of 56.3°F (13.5°C). The wind speed is 10.4 km/h coming from the sout
hwest direction. The humidity is at 84%, and the visibility is 6.0 miles.'
```

The line of code above accesses the content of the last message in the ‘result’ dictionary’s ‘messages’ list, that is the final output from the sequence of interactions. We can see that the output is able to give us valuable information about the weather in San Francisco. The output demonstrated the agent’s ability to retrieve real-time information and hen present it in a clear and comprehensive format.

```
messages = [HumanMessage(content = "What is the weather in SF and LA?")]
result = abot.graph.invoke({"messages": messages})
```

```
Calling: {'name': 'tavily_search_results_json', 'args': {'query': 'weather in San Francisco'}, 'id': 'call_4XX1WnYbsxARGO73GODjtuWG',
'type': 'tool_call'}
Calling: {'name': 'tavily_search_results_json', 'args': {'query': 'weather in Los Angeles'}, 'id': 'call_lbqVwSmLbO6vl3xLDUKt25bK', 't
ype': 'tool_call'}
back to the model!
```

Above, we add ab additional layer to the initial query, we want to illustrate the process of querying multiple pieces of information in one go, demonstrating the agents ability to handle more complex queries. We can see that the Agent knew to make two tool calls to Tavily, one for San Francisco and one for Los Angeles, then the Agent prints “back to the model!” indicating that control returns to the LLM to generate a cohesive response covering both cities weather.

```
result['messages'][-1].content
```

```
'The current weather in San Francisco is sunny with a temperature of 56.3°F (13.5°C). The wind speed is 10.4 kph coming from the south
west direction. The humidity level is at 84%.\n\nIn Los Angeles, the weather is also sunny with a temperature of 75.3°F (24.0°C). The
wind speed is 5.4 kph coming from the south-southeast direction. The humidity level is at 57%.'
```

Once again, the messages are stored in the result dictionary and we retrieve the content from the last one sent in, which would be our system output for the user. In this case, the output provides detailed weather information for both San Francisco and Los Angeles. The response describes the current weather in San Francisco as sunny with a temperature of 56.3°F (13.5°C), along with wind speed, direction, and humidity levels. Similarly, it provides the weather conditions for Los Angeles, stating that it is also sunny with a temperature of 75.3°F (24.0°C), along with the corresponding wind speed and humidity.

This output demonstrates the agent's ability to effectively manage multiple queries within a single interaction, providing a cohesive and informative response that addresses all parts of the user's request.

```
query = "Who won the super bowl in 2024? \
What is the GDP of that state? Answer each question."
messages = [HumanMessage(content=query)]

model = ChatOpenAI(model="gpt-4o")  # requires more advanced model
abot = Agent(model, [tool], system=prompt)
result = abot.graph.invoke({"messages": messages})
```

```
Calling: {'name': 'tavily_search_results_json', 'args': {'query': 'Super Bowl winner 2024'}, 'id': 'call_1vHMe5OYlJCxSYWJh73jb8f6', 't
ype': 'tool_call'}
back to the model!
Calling: {'name': 'tavily_search_results_json', 'args': {'query': 'GDP of Missouri 2024'}, 'id': 'call_wvKgp3zZzwaDkyiWyNwU8xSV', 'typ
e': 'tool_call'}
back to the model!
```

In this example, the agent utilizes the advanced "gpt-4o" model to handle a complex query that involves multiple steps. The query asks, "Who won the Super Bowl in 2024?" and "What is the GDP of that state?" The agent processes this by first creating a ‘HumanMessage’ object with the query content. The agent then breaks down the task, first calling the ‘tavily_search_results_json’ tool to retrieve the Super Bowl winner. Once this information is obtained, it makes a second tool call to find the GDP of the relevant state, assumed to be Missouri. After each tool call, control returns to the LLM to synthesize the retrieved information.

This process demonstrates the agent's ability to manage multi-part and even more complex queries than the last example by leveraging external tools and integrating diverse data points into a cohesive response, showcasing the effectiveness of using a more advanced model for intricate tasks.

```
print(result['messages'][-1].content)
```

```
### Super Bowl Winner 2024
The Kansas City Chiefs won the Super Bowl in 2024, defeating the San Francisco 49ers in an overtime victory.

### GDP of Missouri
The real gross domestic product (GDP) of Missouri in 2024 is not directly available, but in 2023, it was approximately $344.12 billion
USD. Given the context, it is likely to have seen a modest increase in 2024.
```

The final step of the process involves retrieving and displaying the content of the last message in the result, which contains the response to the user's complex query. The output provides detailed answers to both parts of the query. It reveals that the Kansas City Chiefs won the Super Bowl in 2024, defeating the San Francisco 49ers in an overtime victory. Additionally, it estimates the GDP of Missouri, noting that while the exact figure for 2024 is not directly available, the GDP in 2023 was approximately $344.12 billion USD, with a likely modest increase in 2024. This final output highlights the agent's capability to gather and integrate information from multiple sources, delivering a comprehensive and accurate response to the user's query.