Custom LLM Agent (with a ChatModel)

This notebook goes through how to create your own custom agent based on a chat model.

An LLM chat agent consists of three parts:

- PromptTemplate: This is the prompt template that can be used to instruct the language model on what to do
- ChatModel: This is the language model that powers the agent
- `stop` sequence: Instructs the LLM to stop generating as soon as this string is found
- OutputParser: This determines how to parse the LLMOutput into an AgentAction or AgentFinish object

The LLMAgent is used in an AgentExecutor. This AgentExecutor can largely be thought of as a loop that:

1. Passes user input and any previous steps to the Agent (in this case, the LLMAgent)
2. If the Agent returns an `AgentFinish`, then return that directly to the user
3. If the Agent returns an `AgentAction`, then use that to call a tool and get an `Observation`
4. Repeat, passing the `AgentAction` and `Observation` back to the Agent until an `AgentFinish` is emitted.

`AgentAction` is a response that consists of `action` and `action_input`. `action` refers to which tool to use, and `action_input` refers to the input to that tool. `log` can also be provided as more context (that can be used for logging, tracing, etc).

`AgentFinish` is a response that contains the final message to be sent back to the user. This should be used to end an agent run.

In this notebook we walk through how to create a custom LLM agent.

Set up environment

Do necessary imports, etc.

```
pip install langchain
pip install google-search-results
pip install openai
```

```
from langchain.agents import Tool, AgentExecutor, LLMSingleActionAgent,
AgentOutputParser
from langchain.prompts import BaseChatPromptTemplate
from langchain import SerpAPIWrapper, LLMChain
from langchain.chat_models import ChatOpenAI
from typing import List, Union
from langchain.schema import AgentAction, AgentFinish, HumanMessage
import re
from getpass import getpass
```

Set up tool

Set up any tools the agent may want to use. This may be necessary to put in the prompt (so that the agent knows to use these tools).

```
SERPAPI_API_KEY = getpass()
```

```
# Define which tools the agent can use to answer user queries
search = SerpAPIWrapper(serpapi_api_key=SERPAPI_API_KEY)
tools = [
    Tool(
        name = "Search",
        func=search.run,
        description="useful for when you need to answer questions about
current events"
    )
]
```

Prompt Template

This instructs the agent on what to do. Generally, the template should incorporate:

- `tools`: which tools the agent has access and how and when to call them.
- `intermediate_steps`: These are tuples of previous (`AgentAction`, `Observation`) pairs. These are generally not passed directly to the model, but the prompt template formats them in a specific way.
- `input`: generic user input

```
# Set up the base template
template = """Complete the objective as best you can. You have access to the following tools:

{tools}

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

These were previous tasks you completed:



Begin!

Question: {input}
{agent_scratchpad}"""
```

```
# Set up a prompt template
class CustomPromptTemplate(BaseChatPromptTemplate):
    # The template to use
    template: str
    # The list of tools available
    tools: List[Tool]

    def format_messages(self, **kwargs) -> str:
        # Get the intermediate steps (AgentAction, Observation tuples)
```

```
        # Format them in a particular way
        intermediate_steps = kwargs.pop("intermediate_steps")
        thoughts = ""
        for action, observation in intermediate_steps:
            thoughts += action.log
            thoughts += f"\nObservation: {observation}\nThought: "
        # Set the agent_scratchpad variable to that value
        kwargs["agent_scratchpad"] = thoughts
        # Create a tools variable from the list of tools provided
        kwargs["tools"] = "\n".join([f"{tool.name}: {tool.description}"
for tool in self.tools])
        # Create a list of tool names for the tools provided
        kwargs["tool_names"] = ", ".join([tool.name for tool in
self.tools])
        formatted = self.template.format(**kwargs)
        return [HumanMessage(content=formatted)]
```

```
prompt = CustomPromptTemplate(
    template=template,
    tools=tools,
    # This omits the `agent_scratchpad`, `tools`, and `tool_names`
variables because those are generated dynamically
    # This includes the `intermediate_steps` variable because that is
needed
    input_variables=["input", "intermediate_steps"]
)
```

Output Parser

The output parser is responsible for parsing the LLM output into `AgentAction` and `AgentFinish`. This usually depends heavily on the prompt used.

This is where you can change the parsing to do retries, handle whitespace, etc

```
class CustomOutputParser(AgentOutputParser):

    def parse(self, llm_output: str) -> Union[AgentAction, AgentFinish]:
        # Check if agent should finish
        if "Final Answer:" in llm_output:
```

```
            return AgentFinish(
                # Return values is generally always a dictionary with a
single `output` key
                # It is not recommended to try anything else at the moment
:)
                return_values={"output": llm_output.split("Final Answer:")
[-1].strip()},
                log=llm_output,
            )
        # Parse out the action and action input
        regex = r"Action\s*\d*\s*:(.*?)\nAction\s*\d*\s*Input\s*\d*\s*:
[\s]*(.*)"
        match = re.search(regex, llm_output, re.DOTALL)
        if not match:
            raise ValueError(f"Could not parse LLM output:
`{llm_output}`")
        action = match.group(1).strip()
        action_input = match.group(2)
        # Return the action and action input
        return AgentAction(tool=action, tool_input=action_input.strip("
").strip('"'), log=llm_output)
```

```
output_parser = CustomOutputParser()
```

Set up LLM

Choose the LLM you want to use!

```
OPENAI_API_KEY = getpass()
```

```
llm = ChatOpenAI(openai_api_key=OPENAI_API_KEY, temperature=0)
```

Define the stop sequence

This is important because it tells the LLM when to stop generation.

This depends heavily on the prompt and model you are using. Generally, you want this to be whatever token you use in the prompt to denote the start of an `Observation` (otherwise, the LLM may hallucinate an observation for you).

Set up the Agent

We can now combine everything to set up our agent

```
# LLM chain consisting of the LLM and a prompt
llm_chain = LLMChain(llm=llm, prompt=prompt)
```

```
tool_names = [tool.name for tool in tools]
agent = LLMSingleActionAgent(
    llm_chain=llm_chain,
    output_parser=output_parser,
    stop=["\nObservation:"],
    allowed_tools=tool_names
)
```

Use the Agent

Now we can use it!

```
agent_executor = AgentExecutor.from_agent_and_tools(agent=agent,
tools=tools, verbose=True)
```

```
agent_executor.run("Search for Leo DiCaprio's girlfriend on the
internet.")
```

```


    > Entering new AgentExecutor chain...
    Thought: I should use a reliable search engine to get accurate
information.
```

```
    Action: Search
    Action Input: "Leo DiCaprio girlfriend"

    Observation:He went on to date Gisele Bündchen, Bar Refaeli, Blake
Lively, Toni Garrn and Nina Agdal, among others, before finally settling
down with current girlfriend Camila Morrone, who is 23 years his junior.
    I have found the answer to the question.
    Final Answer: Leo DiCaprio's current girlfriend is Camila Morrone.

    > Finished chain.
```

```
    "Leo DiCaprio's current girlfriend is Camila Morrone."
```