Modules Agents How-to Custom agent with tool retrieval

Custom agent with tool retrieval

This notebook builds off of this notebook and assumes familiarity with how agents work.

The novel idea introduced in this notebook is the idea of using retrieval to select the set of tools to use to answer an agent query. This is useful when you have many many tools to select from. You cannot put the description of all the tools in the prompt (because of context length issues) so instead you dynamically select the N tools you do want to consider using at run time.

In this notebook we will create a somewhat contrieved example. We will have one legitimate tool (search) and then 99 fake tools which are just nonsense. We will then add a step in the prompt template that takes the user input and retrieves tool relevant to the query.

Set up environment

Do necessary imports, etc.

```
from langchain.agents import (
    Tool,
    AgentExecutor,
    LLMSingleActionAgent,
    AgentOutputParser,
)
from langchain.prompts import StringPromptTemplate
from langchain import OpenAI, SerpAPIWrapper, LLMChain
from typing import List, Union
from langchain.schema import AgentAction, AgentFinish
import re
```

API Reference:

- Tool from `langchain.agents`
- AgentExecutor from `langchain.agents`
- LLMSingleActionAgent from `langchain.agents`
- AgentOutputParser from `langchain.agents`

- StringPromptTemplate from `langchain.prompts`
- AgentAction from `langchain.schema`
- AgentFinish from `langchain.schema`

Set up tools

We will create one legitimate tool (search) and then 99 fake tools

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


def fake_func(inp: str) -> str:
    return "foo"


fake_tools = [
    Tool(
        name=f"foo-{i}",
        func=fake_func,
        description=f"a silly function that you can use to get more information about the number {i}",
    )
    for i in range(99)
]
ALL_TOOLS = [search_tool] + fake_tools
```

Tool Retriever

We will use a vectorstore to create embeddings for each tool description. Then, for an incoming query we can create embeddings for that query and do a similarity search for relevant tools.

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.schema import Document
```

API Reference:

- FAISS from `langchain.vectorstores`
- OpenAIEmbeddings from `langchain.embeddings`
- Document from `langchain.schema`

```
docs = [
    Document(page_content=t.description, metadata={"index": i})
    for i, t in enumerate(ALL_TOOLS)
]
```

```
vector_store = FAISS.from_documents(docs, OpenAIEmbeddings())
```

```
retriever = vector_store.as_retriever()


def get_tools(query):
    docs = retriever.get_relevant_documents(query)
    return [ALL_TOOLS[d.metadata["index"]] for d in docs]
```

We can now test this retriever to see if it seems to work.

```
get_tools("whats the weather?")
```

```
    [Tool(name='Search', description='useful for when you need to answer
questions about current events', return_direct=False, verbose=False,
callback_manager=<langchain.callbacks.shared.SharedCallbackManager object
at 0x114b28a90>, func=<bound method SerpAPIWrapper.run of
SerpAPIWrapper(search_engine=<class 'serpapi.google_search.GoogleSearch'>,
params={'engine': 'google', 'google_domain': 'google.com', 'gl': 'us',
'hl': 'en'}, serpapi_api_key='', aiosession=None)>, coroutine=None),
     Tool(name='foo-95', description='a silly function that you can use to
get more information about the number 95', return_direct=False,
verbose=False, callback_manager=
<langchain.callbacks.shared.SharedCallbackManager object at 0x114b28a90>,
func=<function fake_func at 0x15e5bd1f0>, coroutine=None),
     Tool(name='foo-12', description='a silly function that you can use to
get more information about the number 12', return_direct=False,
verbose=False, callback_manager=
<langchain.callbacks.shared.SharedCallbackManager object at 0x114b28a90>,
func=<function fake_func at 0x15e5bd1f0>, coroutine=None),
     Tool(name='foo-15', description='a silly function that you can use to
get more information about the number 15', return_direct=False,
verbose=False, callback_manager=
<langchain.callbacks.shared.SharedCallbackManager object at 0x114b28a90>,
func=<function fake_func at 0x15e5bd1f0>, coroutine=None)]
```

```
get_tools("whats the number 13?")
```

```
    [Tool(name='foo-13', description='a silly function that you can use to
get more information about the number 13', return_direct=False,
verbose=False, callback_manager=
<langchain.callbacks.shared.SharedCallbackManager object at 0x114b28a90>,
func=<function fake_func at 0x15e5bd1f0>, coroutine=None),
     Tool(name='foo-12', description='a silly function that you can use to
get more information about the number 12', return_direct=False,
verbose=False, callback_manager=
<langchain.callbacks.shared.SharedCallbackManager object at 0x114b28a90>,
func=<function fake_func at 0x15e5bd1f0>, coroutine=None),
     Tool(name='foo-14', description='a silly function that you can use to
get more information about the number 14', return_direct=False,
verbose=False, callback_manager=
<langchain.callbacks.shared.SharedCallbackManager object at 0x114b28a90>,
func=<function fake_func at 0x15e5bd1f0>, coroutine=None),
     Tool(name='foo-11', description='a silly function that you can use to
get more information about the number 11', return_direct=False,
verbose=False, callback_manager=
<langchain.callbacks.shared.SharedCallbackManager object at 0x114b28a90>,
func=<function fake_func at 0x15e5bd1f0>, coroutine=None)]
```

Prompt Template

The prompt template is pretty standard, because we're not actually changing that much logic in the actual prompt template, but rather we are just changing how retrieval is done.

```
# Set up the base template
template = """Answer the following questions as best you can, but speaking as a pirate might speak. 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

Begin! Remember to speak as a pirate when giving your final answer. Use lots of "Arg"s

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

The custom prompt template now has the concept of a tools_getter, which we call on the input to select the tools to use

```
from typing import Callable


# Set up a prompt template
class CustomPromptTemplate(StringPromptTemplate):
    # The template to use
    template: str
    ############## NEW ######################
    # The list of tools available
    tools_getter: Callable

    def format(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
        ############## NEW ######################
        tools = self.tools_getter(kwargs["input"])
        # Create a tools variable from the list of tools provided
        kwargs["tools"] = "\n".join(
            [f"{tool.name}: {tool.description}" for tool in tools]
        )
        # Create a list of tool names for the tools provided
        kwargs["tool_names"] = ", ".join([tool.name for tool in tools])
        return self.template.format(**kwargs)
```

```
prompt = CustomPromptTemplate(
    template=template,
    tools_getter=get_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 unchanged from the previous notebook, since we are not changing anything about the output format.

```
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, stop sequence, and the agent

Also the same as the previous notebook

```
llm = OpenAI(temperature=0)
```

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

```
tools = get_tools("whats the weather?")
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("What's the weather in SF?")
```

```


    > Entering new AgentExecutor chain...
    Thought: I need to find out what the weather is in SF
    Action: Search
    Action Input: Weather in SF

    Observation:Mostly cloudy skies early, then partly cloudy in the
afternoon. High near 60F. ENE winds shifting to W at 10 to 15 mph.
Humidity71%. UV Index6 of 10. I now know the final answer
    Final Answer: 'Arg, 'tis mostly cloudy skies early, then partly cloudy
in the afternoon. High near 60F. ENE winds shiftin' to W at 10 to 15 mph.
Humidity71%. UV Index6 of 10.

    > Finished chain.




    "'Arg, 'tis mostly cloudy skies early, then partly cloudy in the
```

```
afternoon. High near 60F. ENE winds shiftin' to W at 10 to 15 mph.
Humidity71%. UV Index6 of 10."
```