Langchain Tools

an open-source replacement for OpenAI functions

Jan Palášek

2023-12-17

Although LLM are very capable models, they have a difficulty with certain tasks. One of them can be counting the number of words in the input. To increase their reliability, we can prepare functions that help them overcome these difficult tasks. OpenAI Functions are one way to do this. In the input, we describe which functions it can call in the input. Output of the model is call of the function and its parameters. However, these functions are OpenAI-specific and other LLM models do not implement them. Hence, we can use a very similar technology that could be utilized by all open-source models: Langchain Tools. In this post, we explore how to use them.

Task

Our task will be the following: we will try to spell the input number one-by-one using a LLM model. First, we try to solve it directly with the LLM. In our second attempt, we will use a Langchain Tool.

Direct Approach

First, we need to install appropriate packages.

```
!pip install openai langchain
```

Now, we can try to solve this task using GPT directly.

```
from langchain.prompts import PromptTemplate, SystemMessagePromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.output_parsers import OutputFixingParser
```

```
template = r"""Your goal is to return the slightly modified input sentence. If the input c

---
EXAMPLES

Input: The first ten digits of PI are 1415926535.
Output: The first ten digits of PI are one four one five nine two six five three five.
---

Input: {human_input}
Output: """
prompt = PromptTemplate.from_template(template)

chain = (
  prompt
  | ChatOpenAI(model="gpt-3.5-turbo-1106")
)

response = chain.invoke({"human_input": "Shall we meet in 627391 seconds?"})

print(response.content)
```

```
Shall we meet in six two seven three nine one seconds?
```

The correct answer is:

```
Shall we meet in six two seven three nine one seconds?
```

Using Tools

First we define the tool. The tool can be a normal Python function with one slight difference: it is necessary to document it [1] and we need to annotate it using `@tool` decorator. That's all!

```
from langchain.tools import tool

@tool
```

[1]Documentation needs to be done using Google style docs.

```python
def spell_number(number: int) -> str:
  """
  Purpose of this tool is to spell the input number digit after digit.

  Args:
    number (int): The input number. For example: 164

  Returns:
    Spelled number one digit after another. For example: one six four
  """
  number = str(number)

  nr_to_word_map = {
    "0": "zero",
    "1": "one",
    "2": "two",
    "3": "three",
    "4": "four",
    "5": "five",
    "6": "six",
    "7": "seven",
    "8": "eight",
    "9": "nine"
  }

  result = " ".join([nr_to_word_map[c] for c in number])
  return result
```

We need to offer this tool to the model. For this, we will define `convert_tools` function.

```python
import json

def convert_tools(tools) -> str:
    """
    Converts tools definitions into a string.

    Args:
        tools: List of tools.

    Returns:
        String representation of the tools.
    """
    tools_descr = []

    for tool in tools:
        tool_dict = {
            "name": tool.name,
            "description": tool.description
        }
        tools_descr.append(tool_dict)

    return json.dumps(tools_descr)
```

```
convert_tools([spell_number])
```

```
'[{"name": "spell_number", "description": "spell_number(number: int) -> str - Purpose of thi
```

Finally, we want to structure output of our model. This will allow us to better parse it.

```
from pydantic import BaseModel, Field
from typing import Dict
from langchain.output_parsers import PydanticOutputParser

class SelectedTool(BaseModel):
  """
  Specifies how to call the selected tool.
  """
  name: str = Field(..., description="Name of the tool that you want to use")
  args: dict = Field(..., description="Dictionary mapping name of the argument that we wan

parser = PydanticOutputParser(pydantic_object=SelectedTool)
```

Now we are ready to overcome the task. In our prompt we:

- Describe the model how to use the tools.
- Specify available tools.
- Specify output format, specifically that it should follow schema of our `SelectedTool` pydantic model.

```
from langchain.prompts import PromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.output_parsers import OutputFixingParser
```

```
template = r"""Your goal is to extract the first number from the input sentence and then d

---
AVAILABLE TOOLS: in this section we specify all tools that can be called. Each tool specif

- name: name of the tool.
- description: description of the tool. Describes the tool's purpose, its arguments with t

{tools}
---

FORMAT

{output_format}
---

Input: {human_input}
Output: """

prompt = PromptTemplate.from_template(template=template)


chain = (
  prompt.partial(tools=convert_tools([spell_number]), output_format=parser.get_format_inst
  | ChatOpenAI(model="gpt-3.5-turbo-1106")
  | parser
)


resp = chain.invoke({"human_input": "Shall we meet in 627391 seconds?"})
print(resp)
```

```
name='spell_number' args={'number': 627391}
```

We can see that our output looks great! Now we just need to call the function described by the tool and make the LLM model compose it back into the sentence.

Calling the function can be done using the following code. It will obtain the function by name. Then we try to call it using the args.

```python
tool = globals().get(resp.name)
tool.func(**resp.args)
```

```
'six two seven three nine one'
```

```python
from operator import itemgetter
from langchain_core.runnables import RunnableLambda

# template to compose the tool result
compose_template = r"""Your goal is to:

1. Read the input sentence.
2. Replace the number defined in its input by the function call result.
3. Return the modified output.

---
FUNCTION CALL RESULT: `{func_result}`
---

Input: {human_input}
Output: """
compose_prompt = PromptTemplate.from_template(compose_template)

chain = (
    {
        # branch that passes through
        "human_input": itemgetter("human_input"),
        # branch that extracts information about how to call the tool function
        "func_result": (
                prompt.partial(
                    tools=convert_tools([spell_number]),
                    output_format=parser.get_format_instructions())
                | ChatOpenAI(model="gpt-3.5-turbo-1106")
                | parser
                | RunnableLambda(lambda x: globals().get(x.name).func(**x.args))
        )
    }
    # compose the information
    | compose_prompt
    | ChatOpenAI(model="gpt-3.5-turbo-1106")
)
```

```
result = chain.invoke({"human_input": "Shall we meet in 627391 seconds?"})
print(result.content)
```

```
Shall we meet in six two seven three nine one seconds?
```

The correct answer is:

```
Shall we meet in six two seven three nine one seconds?
```

Summary

In this article, we looked at the possibility of offloading difficult tasks to Langchain Tools.