Can LLMs Replace Data Analysts? Building An LLM-Powered Analyst

Part 1: empowering ChatGPT with tools

Mariya Mansurova · Follow
Published in Towards Data Science
19 min read · Dec 11, 2023

Listen Share

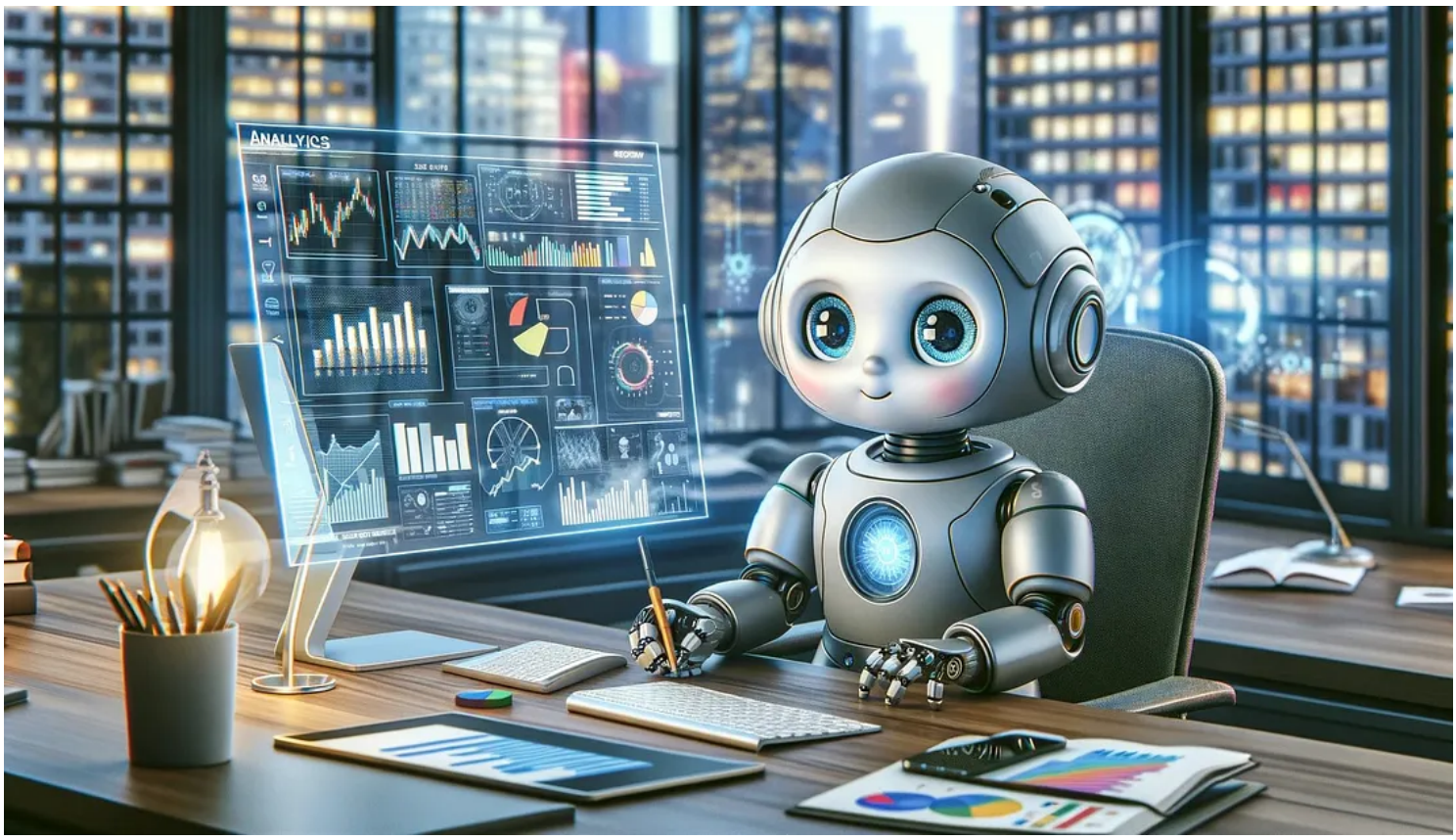

Image by DALL-E 3

I think each of us has wondered at least once over the past year if (or rather when) ChatGPT will be able to replace your role. I'm no exception here.

We have a somewhat consensus that the recent breakthroughs in Generative AI will highly affect our personal lives and work. However, there is no clear view yet of how our roles will change over time.

Spending lots of time thinking about different possible future scenarios and their probabilities might be captivating, but I suggest an absolutely different approach — to try to build your prototype yourself. First, it's rather challenging and fun. Second, it will help us to look at our work in a more structured way. Third, it will give us an opportunity to try in practice one of the most cutting-edge approaches — LLM agents.

In this article, we will start simple and learn how LLMs can leverage tools and do straightforward tasks. But in the following articles, we will dive deeper into different approaches and best practices for LLM agents.

So, let the journey begin.

What is data analytics?

Before moving on to the LLMs, let's try defining what analytics is and what tasks we do as analysts.

My motto is that the goal of the analytical team is to help the product teams make the right decisions based on data in the available time. It's a good mission, but to define the scope of the LLM-powered analyst, we should decompose the analytical work further.

I like the framework proposed by Gartner. It identifies four different Data and Analytics techniques:

- **Descriptive analytics** answers questions like "What happened?". For example, what was the revenue in December? This approach includes reporting tasks and working with BI tools.
- **Diagnostic analytics** goes a bit further and asks questions like "Why did something happen?". For example, why revenue decreased by 10% compared to the previous year? This technique requires more drill-down and slicing & dicing of your data.

- **Predictive analytics** allows us to get answers to questions like "What will happen?". The two cornerstones of this approach are forecasting (predicting the future for business-as-usual situations) and simulation (modelling different possible outcomes).

- **Prescriptive analytics** impacts the final decisions. The common questions are "What should we focus on?" or "How could we increase volume by 10%?".

Usually, companies go through all these stages step by step. It's almost impossible to start looking at forecasts and different scenario analyses if your company hasn't mastered descriptive analytics yet (you don't have a data warehouse, BI tools, or metrics definitions). So, this framework can also show the company's data maturity.

Similarly, when an analyst grows from junior to senior level, she will likely go through all these stages, starting from well-defined reporting tasks and progressing to vague strategic questions. So, this framework is relevant on an individual level as well.

If we return to our LLM-powered analyst, we should focus on descriptive analytics and reporting tasks. It's better to start from the basics. So, we will focus on learning LLM to understand the basic questions about data.

We've defined our focus for the first prototype. So, we are ready to move on to the technical questions and discuss the concept of LLM agents and tools.

LLM agents and tools

When we were using LLMs before (for example, to do topic modelling here), we described the exact steps ourselves in the code. For example, let's look at the chain below. Firstly, we asked the model to determine the sentiment for a customer review. Then, depending on the sentiment, extract from the review either the advantages or disadvantages mentioned in the text.

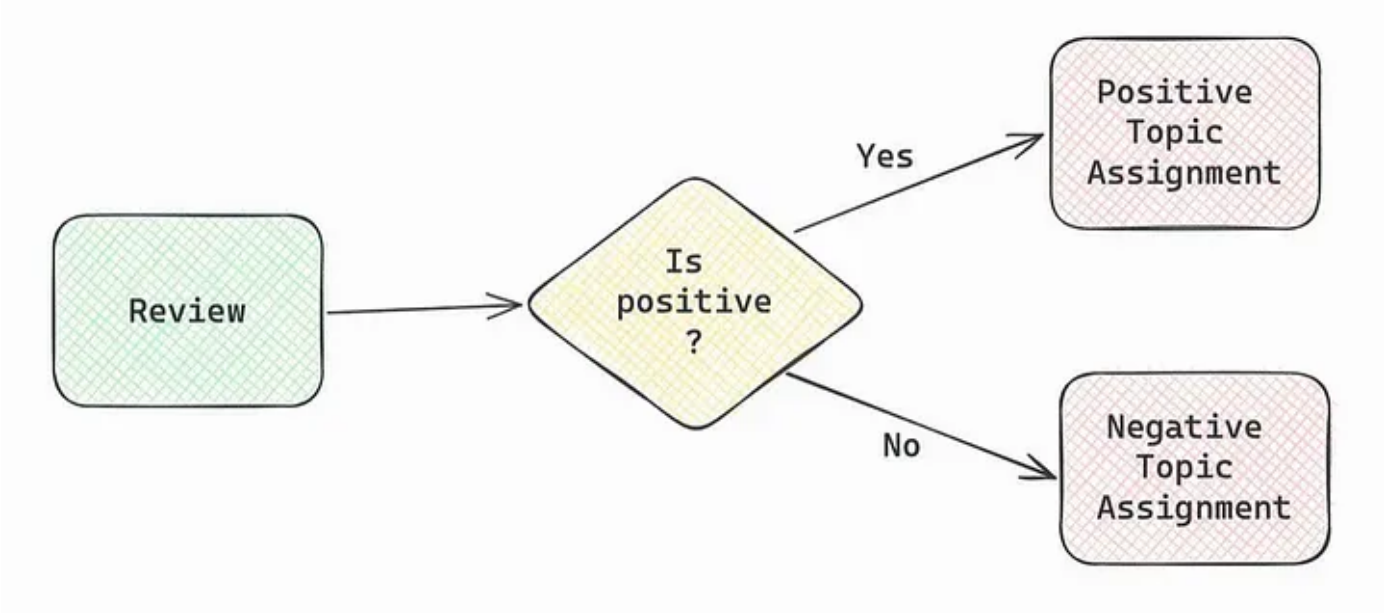

Illustration by author

In this example, we clearly defined the LLM's behaviour, and the LLM solved this task pretty well. However, this approach won't work if we build something more high-level and vague, like an LLM-powered analyst.

If you've ever worked as or with an analyst for at least one day, you would know that analysts are getting a vast range of different questions and asks, starting from basic questions (like "How many customers did we have on our site yesterday?" or "Could you make a graph for our Board meeting tomorrow?") to very high-level ones (for example, "What are the main customer pain points?" or "What market should we launch next?"). It goes without saying it's not feasible to describe all possible scenarios.

However, there's an approach that could help us — agents. The core idea of the agents is to use LLMs as a reasoning engine that could choose what to do next and when it's time to return the final answer to the customer. It sounds pretty close to our behaviour: we get a task, define needed tools, use them, and then come back with the final answer when ready.

The essential concept related to agents (that I've already mentioned above) is tools. Tools are functions that LLM could invoke to get missing information (for example, execute SQL, use a calculator or call a search engine). Tools are crucial because they allow you to bring LLMs to the next level and interact with the world. In this article, we will primarily focus on OpenAI functions as tools.

OpenAI has fine-tuned models to be able to work with functions so that:

- You can pass to the model the list of functions with descriptions;
- If it's relevant to your query, the model will return you a function call — function name and input parameters to call it.

You can find more info and the up-to-date list of models that support functions in the documentation.

There are two prominent use cases to use functions with LLMs:

- Tagging & extraction — in these cases, functions are used to ensure the output format of the model. Instead of the usual output with content, you will get a structured function call.
- Tools & routing — this is a more exciting use case that allows you to create an agent.

Let's start with the more straightforward use case of extraction to learn how to use OpenAI functions.

Use Case #1: Tagging & Extraction

You might wonder what is the difference between tagging and extraction. These terms are pretty close. The only difference is whether the model extracts info presented in the text or labels the text providing new information (i.e. defines language or sentiment).

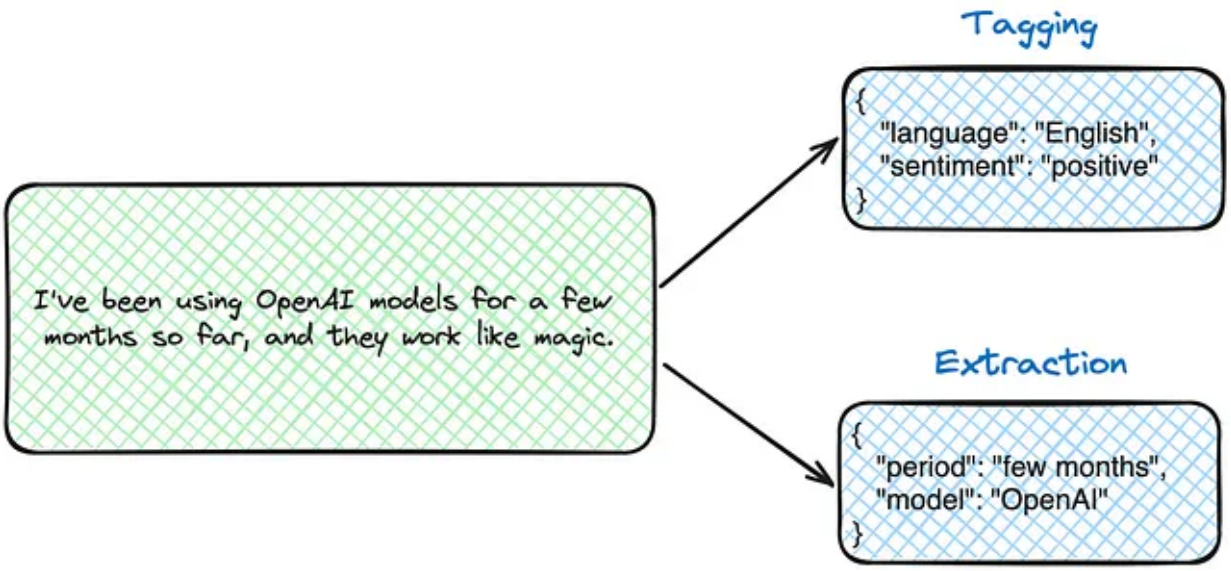

Illustration by author

Since we've decided to focus on descriptive analytics and reporting tasks, let's use this approach to structure incoming data requests and pull the following components: metrics, dimensions, filters, period and desired output.

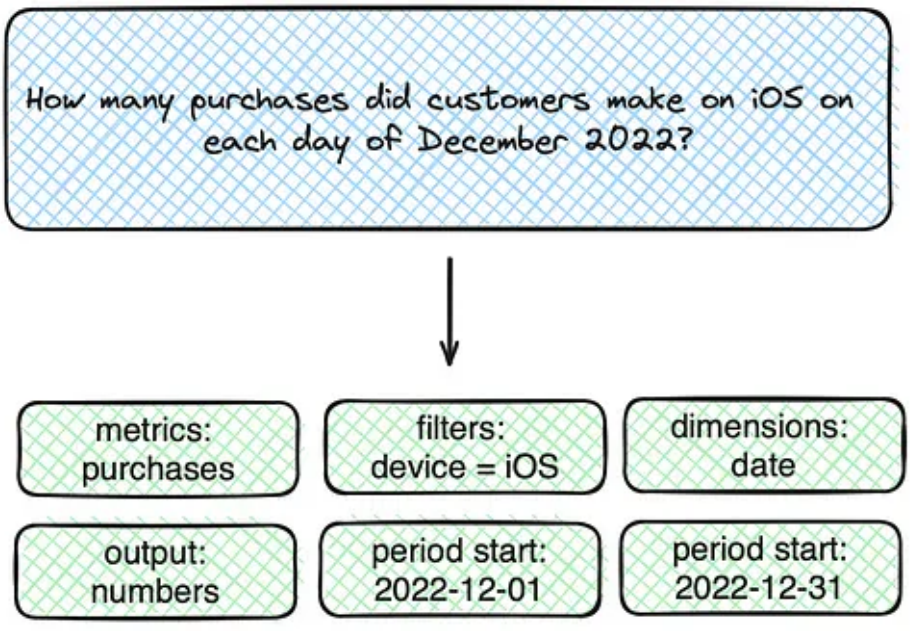

Illustration by author

It will be an example of extraction since we only need information present in the text.

OpenAI Completion API basic example

First, we need to define the function. OpenAI expects a function description as a JSON. This JSON will be passed to LLM, so we need to tell it all the context: what this function does and how to use it.

Here is an example of a function JSON. We've specified:

- `name` and `description` for the function itself,
- `type` and `description` for each argument,
- the list of required input parameters for the function.

```
                    "description": "the desired output",
                    "enum": ["number", "visualisation"]
                }
            },
            "required": ["metric"],
        },
    }
]
```

There's no need to implement the function itself in this use case because we won't be using it. We only get LLM responses in a structured way as function calls.

Now, we could use the standard OpenAI Chat Completion API to call the function. We passed to the API call:

- model — I've used the latest ChatGPT 3.5 Turbo that can work with functions,
- list of messages — one system message to set up the context and a user request,
- list of functions we've defined earlier.

```
print(response)
```

As a result, we got the following JSON.


```
{
  "id": "chatcmpl-8TqGWvGAXZ7L43gYjPyxsWdOTD2n2",
  "object": "chat.completion",
  "created": 1702123112,
  "model": "gpt-3.5-turbo-1106",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": null,
        "function_call": {
          "name": "extract_information",
          "arguments": "{\"metric\":\"number of users\",\"filters\":\"platform='i
        }
      },
      "finish_reason": "function_call"
    }
  ],
  "usage": {
    "prompt_tokens": 159,
    "completion_tokens": 53,
    "total_tokens": 212
  },
  "system_fingerprint": "fp_eeff13170a"
}
```


> *Remember that functions and function calls will be counted into the tokens limits and be billed.*

The model returned a function call instead of a common response: we can see that the

- `metric = "number of users"`,
- `filters = "platform = 'iOS'"`,
- `dimensions = "date"`,
- `period_start = "2021-01-01"`,
- `period_start = "2021-12-31"`,
- `output_type = "visualisation"`.

The model did a pretty good job. The only problem is that it presumed the period out of nowhere. We can fix it by adding more explicit guidance to the system message, for example, `"Extract the relevant information from the provided request. Extract ONLY the information presented in the initial request; don't add anything else. Return partial information if something is missing."`

By default, models decide whether to use functions independently (`function_call = 'auto'`). We can require it to return a specific function call every time or not to use functions at all.

```
# always calling extract_information function
response = openai.ChatCompletion.create(
    model = "gpt-3.5-turbo-1106",
    messages = messages,
    functions = extraction_functions,
    function_call = {"name": "extract_information"}
)

# no function calls
response = openai.ChatCompletion.create(
    model = "gpt-3.5-turbo-1106",
    messages = messages,
    functions = extraction_functions,
    function_call = "none"
)
```

We've got the first working program that uses LLM functions. That's awesome. However, it's not very convenient to describe functions in a JSON. Let's discuss how to do it easier.

Using Pydantic to define functions

To define functions more conveniently, we can leverage Pydantic. Pydantic is the most popular Python library for data validation.

> *We've already used Pydantic to define LangChain Output Parser.*

First, we need to create a class inheriting from the `BaseModel` class and define all the fields (arguments of our function).

```
from pydantic import BaseModel, Field
from typing import Optional

class RequestStructure(BaseModel):
  """extracts information"""
  metric: str = Field(description = "main metric we need to calculate, for example
  filters: Optional[str] = Field(description = "filters to apply to the calculatio
  dimensions: Optional[str] = Field(description = "parameters to split your metric
  period_start: Optional[str] = Field(description = "the start day of the period
  period_end: Optional[str] = Field(description = "the end day of the period for a
  output_type: Optional[str] = Field(description = "the desired output", enum = ['
```

Then, we can use LangChain to convert the Pydantic class into the OpenAI function.

```
from langchain.utils.openai_functions import convert_pydantic_to_openai_function
extract_info_function = convert_pydantic_to_openai_function(RequestStructure,
    name = 'extract_information')
```

LangChain validates the class we provided. For example, it ensures that the function description is specified since LLM needs it to be able to use this tool.

As a result, we got the same JSON to pass to LLM, but now we express it as a Pydantic class.

```
{'name': 'extract_information',
 'description': 'extracts information',
 'parameters': {'title': 'RequestStructure',
  'description': 'extracts information',
  'type': 'object',
  'properties': {'metric': {'title': 'Metric',
    'description': "main metric we need to calculate, for example, 'number of use
    'type': 'string'},
   'filters': {'title': 'Filters',
    'description': 'filters to apply to the calculation (do not include filters o
    'type': 'string'},
   'dimensions': {'title': 'Dimensions',
    'description': 'parameters to split your metric by',
    'type': 'string'},
   'period_start': {'title': 'Period Start',
    'description': 'the start day of the period for a report',
    'type': 'string'},
   'period_end': {'title': 'Period End',
    'description': 'the end day of the period for a report',
    'type': 'string'},
   'output_type': {'title': 'Output Type',
    'description': 'the desired output',
    'enum': ['number', 'visualisation'],
    'type': 'string'}},
  'required': ['metric']}}
```

Now, we could use it in our call to OpenAI. Let's switch from OpenAI API to LangChain to make our API calls more modular.

Defining LangChain chain

Let's define a chain to extract needed information from the requests. We will use LangChain since it's the most popular framework for LLMs. If you haven't worked with it before, I recommend you learn some basics in one of my previous articles.

Our chain is simple. It consists of an Open AI model and prompt with one variable `request` (a user message).

We've also used the `bind` function to pass `functions` argument to the model. The `bind` function allows us to specify constant arguments for our models that are not part of the input (for example, `functions` or `temperature`).

```
from langchain.prompts import ChatPromptTemplate
from langchain.chat_models import ChatOpenAI

model = ChatOpenAI(temperature=0.1, model = 'gpt-3.5-turbo-1106')\
  .bind(functions = [extract_info_function])

prompt = ChatPromptTemplate.from_messages([
    ("system", "Extract the relevant information from the provided request. \
            Extract ONLY the information presented in the initial request. \
            Don't add anything else. \
            Return partial information if something is missing."),
    ("human", "{request}")
])

extraction_chain = prompt | model
```

Now it's time to try our function. We need to use the invoke method and pass a `request`.

```
extraction_chain.invoke({'request': "How many customers visited our site on iOS i
```

In the output, we got `AIMessage` without any content but with a function call.

```
AIMessage(
  content='',
  additional_kwargs={
    'function_call': {
       'name': 'extract_information',
       'arguments': '''{
         "metric":"number of customers", "filters":"device = 'iOS'",
         "dimensions":"country", "period_start":"2023-04-01",
```

```
            "period_end":"2023-04-30", "output_type":"number"}
           '''}
    }
)
```

So, we've learned how to use OpenAI functions in LangChain to get structured output. Now, let's move on to the more interesting use case — tools and routing.

Use Case #2: Tools & Routing

It's time to use tools and empower our model with external capabilities. Models in this approach are reasoning engines, and they can decide what tools to use and when (it's called routing).

LangChain has a concept of tools — interfaces that agents can use to interact with the world. Tools can be functions, LangChain chains or even other agents.

We can easily convert tools into OpenAI functions using `format_tool_to_openai_function` and keep passing the `functions` argument to LLMs.

Defining a custom tool

Let's teach our LLM-powered analyst to calculate the difference between two metrics. We know that LLMs might make mistakes in math, so we would like to ask a model to use a calculator instead of counting on its own.

To define a tool, we need to create a function and use a `@tool` decorator.

```
from langchain.agents import tool

@tool
def percentage_difference(metric1: float, metric2: float) -> float:
    """Calculates the percentage difference between metrics"""
    return (metric2 - metric1)/metric1*100
```

Now, this function has `name` and `description` parameters that will be passed to LLMs.

```
print(percentage_difference.name)
# percentage_difference.name

print(percentage_difference.args)
# {'metric1': {'title': 'Metric1', 'type': 'number'},
# 'metric2': {'title': 'Metric2', 'type': 'number'}}

print(percentage_difference.description)
# 'percentage_difference(metric1: float, metric2: float) -> float - Calculates the
```

These parameters will be used to create an OpenAI function specification. Let's convert our tool to an OpenAI function.

```
from langchain.tools.render import format_tool_to_openai_function
print(format_tool_to_openai_function(percentage_difference))
```

We got the following JSON as the result. It outlines the structure, but field descriptions are missing.

```
{'name': 'percentage_difference',
 'description': 'percentage_difference(metric1: float, metric2: float) -> float -
 'parameters': {'title': 'percentage_differenceSchemaSchema',
  'type': 'object',
  'properties': {'metric1': {'title': 'Metric1', 'type': 'number'},
   'metric2': {'title': 'Metric2', 'type': 'number'}},
  'required': ['metric1', 'metric2']}
}
```

We can use Pydantic to specify a schema for the arguments.

```
    metric2: float = Field(description="New metric value that we compare with the

@tool(args_schema=Metrics)
def percentage_difference(metric1: float, metric2: float) -> float:
    """Calculates the percentage difference between metrics"""
    return (metric2 - metric1)/metric1*100
```

Now, if we convert a new version to the OpenAI function specification, it will include argument descriptions. It's much better since we could share all the needed context with the model.

```
{'name': 'percentage_difference',
 'description': 'percentage_difference(metric1: float, metric2: float) -> float -
 'parameters': {'title': 'Metrics',
  'type': 'object',
  'properties': {'metric1': {'title': 'Metric1',
    'description': 'Base metric value to calculate the difference',
    'type': 'number'},
   'metric2': {'title': 'Metric2',
    'description': 'New metric value that we compare with the baseline',
    'type': 'number'}},
  'required': ['metric1', 'metric2']}}
```

So, we've defined the tool that LLM will be able to use. Let's try it in practice.

Using a tool in practice

Let's define a chain and pass our tool to the function. Then, we could test it on a user request.

```
analyst_chain = prompt | model
analyst_chain.invoke({'request': "In April we had 100 users and in May only 95. Wh
```

We got a function call with the correct arguments, so it's working.

```
AIMessage(content='', additional_kwargs={
    'function_call': {
      'name': 'percentage_difference',
      'arguments': '{"metric1":100,"metric2":95}'}
  }
)
```

To have a more convenient way to work with the output, we can use `OpenAIFunctionsAgentOutputParser`. Let's add it to our chain.

```
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser
analyst_chain = prompt | model | OpenAIFunctionsAgentOutputParser()
result = analyst_chain.invoke({'request': "There were 100 users in April and 110 u
```

Now, we got output in a more structured way, and we could easily retrieve arguments for our tool as `result.tool_input`.

```
AgentActionMessageLog(
   tool='percentage_difference',
   tool_input={'metric1': 100, 'metric2': 110},
   log="\nInvoking: `percentage_difference` with `{'metric1': 100, 'metric2': 110}
   message_log=[AIMessage(content='', additional_kwargs={'function_call': {'name'
)
```

So, we could execute the function as the LLM requested like this.

```
observation = percentage_difference(result.tool_input)
print(observation)
# 10
```

If we want to get the final answer from the model, we need to pass the function execution result back. To do it, we need to define a message list to pass to the model observations.

```
from langchain.prompts import MessagesPlaceholder

model = ChatOpenAI(temperature=0.1, model = 'gpt-3.5-turbo-1106')\
  .bind(functions = [format_tool_to_openai_function(percentage_difference)])

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a product analyst willing to help your product team. You a
    ("user", "{request}"),
    MessagesPlaceholder(variable_name="observations")
])

analyst_chain = prompt | model | OpenAIFunctionsAgentOutputParser()
result1 = analyst_chain.invoke({
    'request': "There were 100 users in April and 110 users in May. How did the nu
    "observations": []
})

observation = percentage_difference(result1.tool_input)
print(observation)
# 10
```

Then, we need to add the observation to our `observations` variable. We could use `format_to_openai_functions` function to format our results in an expected way for the model.

```
from langchain.agents.format_scratchpad import format_to_openai_functions
format_to_openai_functions([(result1, observation), ])
```

As a result, we got such a message that the LLM can understand.

```
[AIMessage(content='', additional_kwargs={'function_call': {'name': 'percentage_d
                                                            'arguments': '{"metric1":100,"metric2"
 FunctionMessage(content='10.0', name='percentage_difference')]
```

Let's invoke our chain one more time, passing the function execution result as an observation.

```
result2 = analyst_chain.invoke({
    'request': "There were 100 users in April and 110 users in May. How did the nu
    "observations": format_to_openai_functions([(result1, observation)])
})
```

Now, we got the final result from the model, which sounds reasonable.

```
AgentFinish(
  return_values={'output': 'The number of users increased by 10%.'},
  log='The number of users increased by 10%.'
)
```

> *If we were working with vanilla OpenAI Chat Completion API, we could just add another message with role = tool . You can find a detailed example here.*

If we switch on debug, we can see the exact prompt that was passed to OpenAI API.

```
AI: {'name': 'percentage_difference', 'arguments': '{"metric1":100,"metric2":110}
Function: 10.0
```

To switch on LangChain debug, execute the following code and invoke your chain to see what is going on under the hood.

```
import langchain
langchain.debug = True
```

We've tried to work with one tool, but let's extend our toolkit and see how LLM could handle it.

Routing: using multiple tools

Let's add a couple more tools to our analyst's toolkit:

- get monthly active users
- using Wikipedia.

First, let's define a dummy function to calculate the audience with filters by month and city. We will again use Pydantic to specify the input arguments for our function.

```
    else:
        return int(total*random.random())
```

Then, let's use the wikipedia Python package to allow model query Wikipedia.

```
import wikipedia

class Wikipedia(BaseModel):
    term: str = Field(description="Term to search for")

@tool(args_schema=Wikipedia)
def get_summary(term: str) -> str:
    """Returns basic knowledge about the given term provided by Wikipedia"""
    return wikipedia.summary(term)
```

Let's define a dictionary with all the functions our model knows now. This dictionary will help us to do routing later.

```
toolkit = {
    'percentage_difference': percentage_difference,
    'get_monthly_active_users': get_monthly_active_users,
    'get_summary': get_summary
}

analyst_functions = [format_tool_to_openai_function(f)
  for f in toolkit.values()]
```

I've made a couple of changes to our previous setup:

- I tweaked the system prompt a bit to force LLM to consult with Wikipedia if it needs some basic knowledge.
- I've changed the model to GPT 4 because it's better for handling tasks requiring reasoning.

```
from langchain.prompts import MessagesPlaceholder

model = ChatOpenAI(temperature=0.1, model = 'gpt-4-1106-preview')\
  .bind(functions = analyst_functions)

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a product analyst willing to help your product team. You a
        You use only information provided in the initial request. \
        If you need to determine some information i.e. what is the name of the cap
    ("user", "{request}"),
    MessagesPlaceholder(variable_name="observations")
])

analyst_chain = prompt | model | OpenAIFunctionsAgentOutputParser()
```

We can invoke our chain with all the functions. Let's start with a pretty straightforward query.

```
result1 = analyst_chain.invoke({
    'request': "How many users were in April 2023 from Berlin?",
    "observations": []
})
print(result1)
```

We got in the result function call for `get_monthly_active_users` with input parameters — `{'month': '2023-04-01', 'city': 'Berlin'}` , which looks correct. The model was able to find the right tool and solve the task.

Let's try to make task a bit more complex.

```
result1 = analyst_chain.invoke({
    'request': "How did the number of users from the capital of Germany\
        change between April and May 2023?",
    "observations": []
})
```

Let's pause for a minute and think how we would like the model to reason. It's evident that there's not enough information for the model to answer straight away, so it needs to make a bunch of function calls:

- call Wikipedia to get the capital of Germany
- call the `get_monthly_active_users` function twice to get MAU for April and May
- call `percentage_difference` to calculate the difference between metrics.

It looks pretty complex. Let's see whether ChatGPT would be able to handle this question.

For the first call, LLM returned back a function call to Wikipedia with the following params — `{'term': 'capital of Germany'}`. So far, it's following our plan.

Let's provide the observation and see what the next steps will be.

```
observation1 = toolkit[result1.tool](result1.tool_input)
print(observation1)

# The capital of Germany is the  city state of Berlin. It is the seat of
# the President of Germany, whose official residence is Schloss Bellevue.
# The Bundesrat ("federal council") is the representation of the Federal States
# (Bundesländer) of Germany and has its seat at the former Prussian Herrenhaus
# (House of Lords). Though most of the ministries are seated in Berlin,
# some of them, as well as some minor departments, are seated in Bonn,
# the former capital of West Germany.
# Although Berlin is officially the capital of the Federal Republic of Germany,
# 8,000 out of the 18,000 total officials employed at the federal bureaucracy
# still work in Bonn, about 600 km (370 mi) away from Berlin.

# source: https://en.wikipedia.org/wiki/Capital_of_Germany

result2 = analyst_chain.invoke({
    'request': "How did the number of users from the capital of Germany change be
    "observations": format_to_openai_functions([(result1, observation1)])
})
```

The model wants to execute `get_monthly_active_users` with arguments `{'month': '2023-04-01', 'city': 'Berlin'}`. Let's do it and return the information to the model once again.

```
observation2 = toolkit[result2.tool](result2.tool_input)
print(observation2)
# 168

result3 = analyst_chain.invoke({
    'request': "How did the number of users from the capital of Germany change be
    "observations": format_to_openai_functions([(result1, observation1), (result2
})
```

Then, the model requests to call `get_monthly_active_users` again with arguments `{'month': '2023-05-01', 'city': 'Berlin'}`. So far, it's doing an excellent job. Let's follow its logic.

```
observation3 = toolkit[result3.tool](result3.tool_input)
print(observation3)
# 1046

result4 = analyst_chain.invoke({
    'request': "How did the number of users from the capital of Germany change be
    "observations": format_to_openai_functions(
      [(result1, observation1), (result2, observation2),
      (result3, observation3)])
})
```

The subsequent result is a function call for percentage_difference with the following arguments `{'metric1': 168, 'metric2': 1046}`. Let's calculate observation and invoke our chain one more time. Hopefully, it will be the last step.

```
observation4 = toolkit[result4.tool](result4.tool_input)
print(observation4)

# 523.27

result5 = analyst_chain.invoke({
    'request': "How did the number of users from the capital of Germany change be
    "observations": format_to_openai_functions(
      [(result1, observation1), (result2, observation2),
      (result3, observation3), (result4, observation4)])
})
```

In the end, we got the following response from the model: `The number of users from Berlin, the capital of Germany, increased by approximately 523.27% between April and May 2023.`

Here's the complete scheme of the LLM calls for this question.

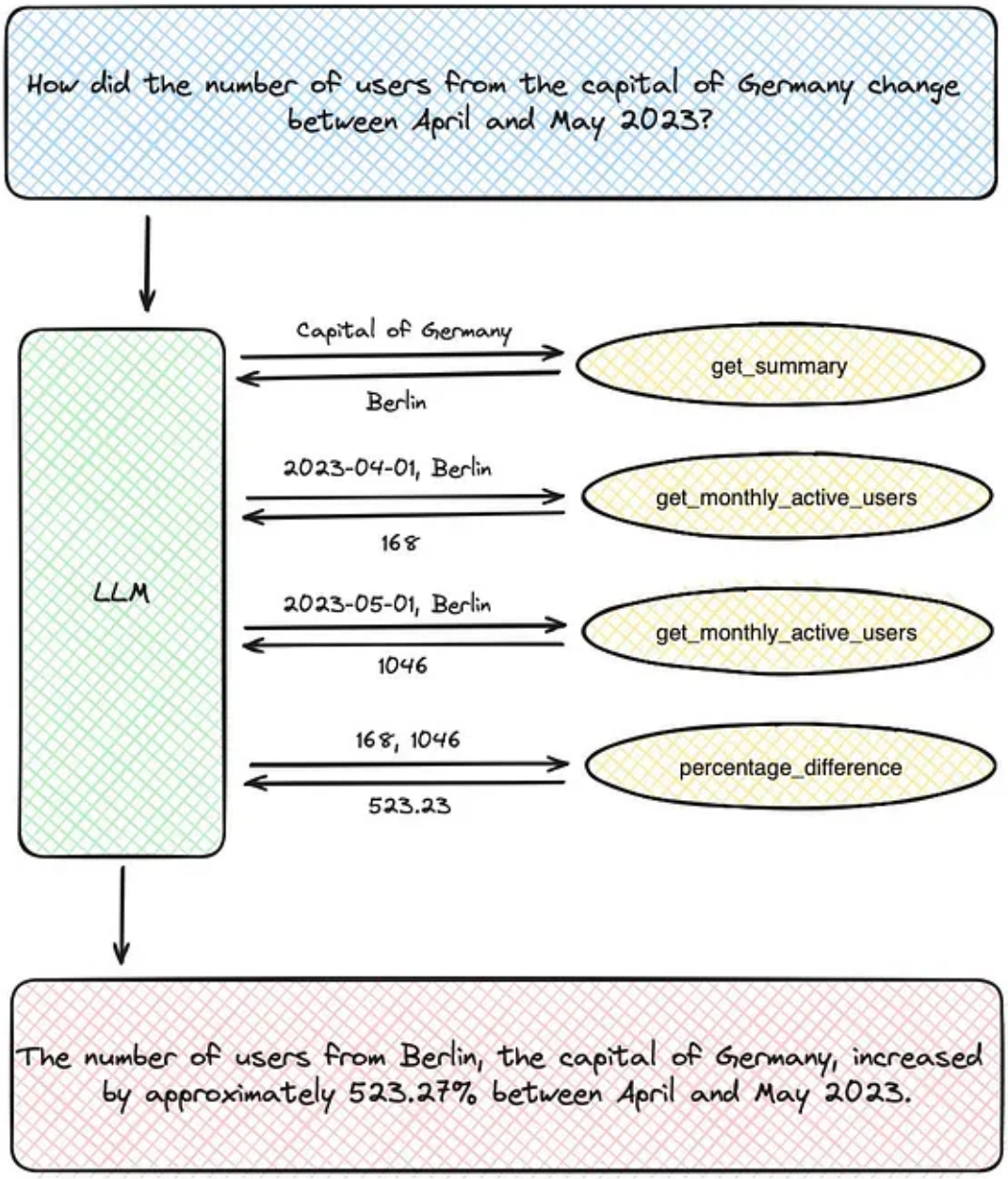

Illustration by author

In the above example, we triggered subsequent calls one by one manually, but it can be easily automated.

It's a fantastic result, and we were able to see how LLMs can do reasoning and utilize multiple tools. It took model 5 steps to achieve the result, but it followed the plan we outlined initially, so it was a pretty logical path. However, if you plan to use LLMs in production, keep in mind that it might make mistakes and introduce evaluation and quality assurance processes.

> *You can find the full code on GitHub.*

Summary

This article taught us how to empower LLMs with external tools using OpenAI functions. We've examined two use cases: extraction to get structured output and routing to use external information for questions. The final result inspires me since LLM could answer pretty complex questions using three different tools.

Let's return to the initial question of whether LLMs can replace data analysts. Our current prototype is basic and far from the junior analysts' capabilities, but it's only the beginning. Stay tuned! We will dive deeper into the different approaches to LLM agents. Next time, we will try to create an agent that can access the database and answer basic questions.

Reference

This article is inspired by the "Functions, Tools and Agents with LangChain" course from DeepLearning.AI

Llm | Data Science | Agents | Editors Pick | Artificial Intelligence

Follow

Written by Mariya Mansurova

9.1K Followers · Writer for Towards Data Science

Data & Product Analytics Lead at Wise | ClickHouse Evangelist