9
Working with Code

In this chapter, we are going to cover another great capability of Large Language Models, that is, working with programming languages. In the previous chapter, we've already seen a glimpse of this capability, namely, SQL query generation in a SQL database. In this chapter, we are going to examine the other ways in which LLMs can be used with code, from "simple" code generation to interaction with code repositories and, finally, to the possibility of letting an application behave as if it were an algorithm. By the end of this chapter, you will be able to leverage LLMs to code-related projects, as well as build LLM-powered applications with natural language interfaces to work with code.

Throughout this chapter, we will cover the following topics:

- Analysis of the main LLMs with top-performing code capabilities
- Using LLMs for code understanding and generation
- Building LLM-powered agents to "act as" algorithms
- Leveraging Code Interpreter

Technical requirements

To complete the tasks in this chapter, you will need the following:

- A Hugging Face account and user access token.
- An OpenAI account and user access token.
- Python 3.7.1 or a later version.
- Python packages. Make sure you have the following Python packages installed: `langchain`, `python-dotenv`, `huggingface_hub`, `streamlit`, `codeinterpreterapi`, and `jupyter_kernel_gateway`. Those can be easily installed via `pip install` in your terminal.

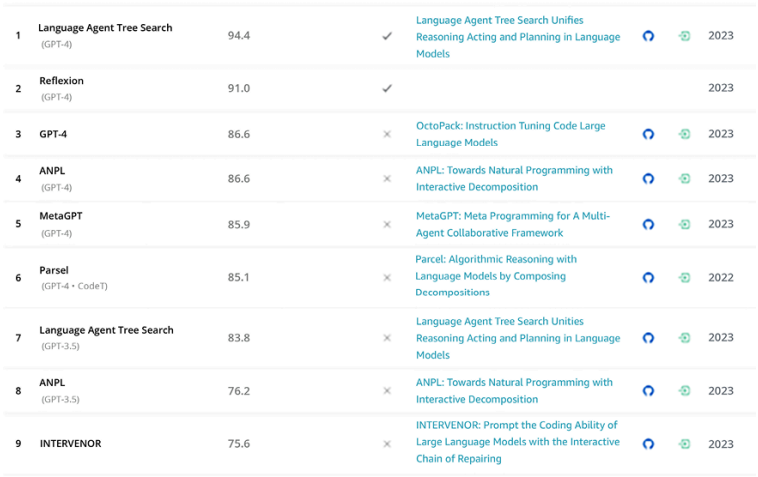
You can find all the code and examples in the book's GitHub repository at https://github.com/PacktPublishing/Building-LLM-Powered-Applications.

Choosing the right LLM for code

In *Chapter 3*, we described a decision framework to use in order to decide the proper LLM for a given application. Generally speaking, all LLMs are endowed with knowledge of code understanding and generation; however, some of them are particularly specialized in doing so. More specifically, there are some evaluation benchmarks – such as the HumanEval – that are specifically tailored to assessing LLMs' capabilities of working with code. The leaderboard of HumanEval One is a good source for determining the top-performing models, available at https://paperswithcode.com/sota/code-generation-on-humaneval. HumanEval is a benchmark introduced by OpenAI to assess the code generation capabilities of LLMs, where the model completes Python functions based on their signature and docstring. It has been used to evaluate

models like Codex, demonstrating its effectiveness in measuring functional correctness.

In the following screenshot, you can see the situation of the leaderboard as of January 2024:



1	Language Agent Tree Search (GPT-4)	94.4	✓	Language Agent Tree Search Unifies Reasoning Acting and Planning in Language Models		2023
2	Reflexion (GPT-4)	91.0	✓			2023
3	GPT-4	86.6	×	OctoPack: Instruction Tuning Code Large Language Models		2023
4	ANPL (GPT-4)	86.6	×	ANPL: Towards Natural Programming with Interactive Decomposition		2023
5	MetaGPT (GPT-4)	85.9	×	MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework		2023
6	Parsel (GPT-4 + CodeT)	85.1	×	Parcel: Algorithmic Reasoning with Language Models by Composing Decompositions		2022
7	Language Agent Tree Search (GPT-3.5)	83.8	×	Language Agent Tree Search Unities Reasoning Acting and Planning in Language Models		2023
8	ANPL (GPT-3.5)	76.2	×	ANPL: Towards Natural Programming with Interactive Decomposition		2023
9	INTERVENOR	75.6	×	INTERVENOR: Prompt the Coding Ability of Large Language Models with the Interactive Chain of Repairing		2023

Figure 9.1: HumanEval benchmark in January 2024

As you can see, the majority of the models are fine-tuned versions of the GPT-4 (as well as the GPT-4 itself), as it is the state-of-the-art LLM in basically all the domains. Nevertheless, there are many open-source models that reached stunning results in the field of code understanding and generation, some of which will be covered in the next sections. Another benchmark is **Mostly Basic Programming Problems** (**MBPP**), a dataset of 974 programming tasks in Python, designed to be solvable by entry-level programmers. Henceforth, when choosing your model for a code-specific task, it might be useful to have a look at these benchmarks as well as other similar code metrics (we will see throughout the chapter some further benchmarks for code-specific LLMs).

Staying within the scope of coding, below you can find three additional benchmarks often used in the market:

- **MultiPL-E**: An extension of HumanEval to many other languages, such as Java, C#, Ruby, and SQL.
- **DS-1000**: A data science benchmark that tests if the model can write code for common data analysis tasks in Python.
- **Tech Assistant Prompt**: A prompt that tests if the model can act as a technical assistant and answer programming-related requests.

In this chapter, we are going to test different LLMs: two code-specific (CodeLlama and StarCoder) and one general-purpose, yet also with emerging capabilities in the field of code generation (Falcon LLM).

Code understanding and generation

The first experiment we are going to run will be code understanding and generation leveraging LLMs. This simple use case is at the base of the many AI code assistants that were developed since the launch of ChatGPT, first among all the GitHub Copilot.

Definition

GitHub Copilot is an AI-powered tool that assists developers in writing code more efficiently. It analyzes code and comments to provide suggestions for individual lines and entire functions. The tool is developed by GitHub, OpenAI, and Microsoft and supports multiple programming languages. It can perform various tasks such as code completion, modification, explanation, and technical assistance.

In this experiment, we are going to try three different models: Falcon LLM, which we already explored in *Chapter 3*; CodeLlama, a fine-tuned version of Meta AI's Llama; and StarCoder, a code-specific model that we are going to investigate in the upcoming sections.

Since those models are pretty heavy to run on a local machine, for this purpose I'm going to use a Hugging Face Hub Inference Endpoint, with a GPU-powered virtual machine. You can link one model per Inference Endpoint and then embed it in your code, or use the convenient library `HuggingFaceEndpoint`, available in LangChain.

To start using your Inference Endpoint, you can use the following code:

```
llm = HuggingFaceEndpoint(endpoint_url = "your_endpoint_url", task = 'text-generation',
        model_kwargs = {"max_new_tokens": 1100})
```

Alternatively, you can copy and paste the Python code provided on your endpoint's webpage at **https://ui.endpoints.huggingface.co/user_name/endpoints/your_endpoint_name**:

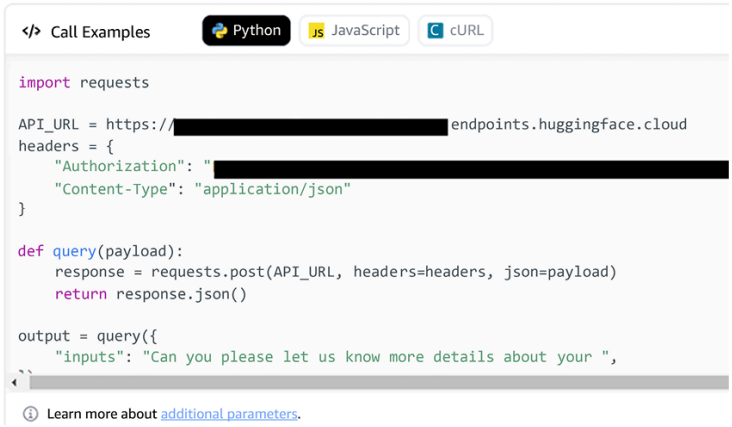


Figure 9.2: User interface of the Hugging Face Inference Endpoint

To create your Hugging Face Inference Endpoint, you can follow the instructions at **https://huggingface.co/docs/inference-endpoints/index**.

You can always leverage the free Hugging Face API as described in *Chapter 4*, but you have to expect some latency when running the models.

Falcon LLM

Falcon LLM is an open-source model developed by Abu Dhabi's **Technology Innovation Institute** (**TII**) and launched on the market in May 2023. It is an autoregressive, decoder-only transformer, trained on 1 trillion tokens, and has 40 billion parameters (although it has also been released as a lighter version with 7 billion parameters). As discussed in *Chapter 3*, "small" language models are a representation of a new trend of LLMs, consisting of building lighter models (with fewer parameters) that focus instead on the quality of the training dataset.

To start using Falcon LLM, we can follow these steps:

1. We can leverage the HuggingFaceHub wrapper available in LangChain (remember to set the Hugging Face API in the `.env` file, passing your secrets as `os.environ["HUGGINGFACEHUB_API_TOKEN"] = HUGGINGFACEHUB_API_TOKEN`):

```
from langchain import HuggingFaceHub
from langchain import PromptTemplate, LLMChain
import os
load_dotenv()hugging_face_api = os.environ["HUGGINGFACEHUB_API_TOKEN"]
repo_id = "tiiuae/falcon-7b-instruct"
llm = HuggingFaceHub(
    repo_id=repo_id,  model_kwargs={"temperature": 0.2, "max_new_tokens": 1000}
)
```

2. Now that we've initialized the model, let's ask it to generate the code for a simple webpage:

```
prompt = """
Generate a short html code to a simple webpage with a header, a subheader, and a text b
<!DOCTYPE html>
<html>
"""
print(llm(prompt))
```

The following is the corresponding output:

```
<head>
    <title>My Webpage</title>
</head>
<body>
    <h1>My Webpage</h1>
    <h2>Subheader</h2>
    <p>This is the text body.</p>
</body>
</html>
```

3. If you save it as an HTML file and execute it, the result will look like the following:

Figure 9.3: Sample webpage generated by FalconLLM

4. We can also try to generate a Python function to generate random passwords:

```
prompt = """
Generate a python program that create random password with lenght of 12 characters, 3 n
"""
print(llm(prompt))
```

Here is our output:

```
import random
def generate_password():
    chars = "abcdefghijklmnopqrstuvwxyz0123456789"
    lenght = 12
    num = random.randint(1, 9)
    cap = random.randint(1, 9)
    password = ""
    for i in range(lenght):
        password += chars[random.randint(0, 9)]
    password += num
    password += cap
    return password
print(generate_password())
```

We now have a function named `generate_password()`, which uses random functions to generate a password as per our prompt.

5. Finally, let's do the opposite, asking the model to explain to us the above code:

```
prompt = """
Explain to me the following code:
def generate_password():
    chars = "abcdefghijklmnopqrstuvwxyz0123456789"
    lenght = 12
    num = random.randint(1, 9)
    cap = random.randint(1, 9)
    password = ""
    for i in range(lenght):
        password += chars[random.randint(0, 9)]
    password += num
    password += cap
    return password
print(generate_password())
"""
print(llm(prompt))
```

Here is the obtained output:

```
<p>The code generates a random password of length 12 characters that contains a mix of let
```


Quick tip: Enhance your coding experience with the **AI Code Explainer** and **Quick Copy** features. Open this book in the next-gen Packt Reader. Click the **Copy** button (**1**) to quickly copy code into your coding environment, or click the **Explain** button (**2**) to get the AI assistant to explain a block of code to you.

The next-gen Packt Reader is included for free with the purchase of this book. Unlock it by scanning the QR code below or visiting https://www.packtpub.com/unlock/9781835462317.

Overall, even if not code-specific, the model was able to correctly perform all the tasks. Note also that this is the "light" version of the model (7 billion parameters), yet its performance is great.

Let's now investigate the capabilities of CodeLlama.

CodeLlama

CodeLlama is a family of LLMs for code based on Llama 2, which is a general-purpose language model developed by Meta AI (as discussed in *Chapter 3*). CodeLlama can generate and discuss code in various programming languages, such as Python, C++, Java, PHP, and more. CodeLlama can also perform infilling, which is the ability to fill in missing parts of code based on the surrounding context, as well as follow instructions given in natural language and produce code that matches the desired functionality.

The model comes in three sizes (7B, 13B, and 34B parameters) and three flavors (base model, Python fine-tuned, and instruction-tuned) to cover a wide range of applications. CodeLlama is trained on sequences of 16k tokens and can handle inputs with up to 100k tokens.

In the model paper "Code Llama: Open Foundation Models for Code" by Rozière Baptiste et al, released in August 2023, the authors describe how the various models were tested against some of the most popular evaluation benchmarks in the domain of code understanding and generation, including HumanEval and MBPP, according to which CodeLlama models achieved a score up to 53% and 55%, respectively. On top of those remarkable results, it is stunning that the Python fine-tuned CodeLlama's smallest size (7 billion parameters) outperformed the largest version of Llama 2 (70 billion parameters) on HumanEval and MBPP.

Now, let's run some tests with this model. As per the previous section, we can initialize the model leveraging either the Hugging Face Inference API (pay per use) or the free Hugging Face API (with the constraint of higher latency). You can consume it as follows:

```
repo_id = "codellama/CodeLlama-7b-Instruct-hf"
llm = HuggingFaceHub(    repo_id=repo_id,  model_kwargs={"temperature": 0.2, "max_new_toke
)
```

Let's now test it with some code tasks. The first task will be that of optimizing Python code so that it runs more efficiently. Let's see how our model performs in this task. In the following code snippet, we simply prompt the model to regenerate the provided code in a more efficient way:

```
prompt = """
Regenerate the below code in a more efficient way.
```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
# Example usage:
n = 5
print("Factorial of", n, "is", factorial(n))
```
"""
print(llm(prompt))
```

Here is our output:

```
### Solution
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```
### Explanation
The solution is a recursive function that calls itself until the base case is reached.
### References
- [Wikipedia](https://en.wikipedia.org/wiki/Factorial)
- [Math is Fun](https://www.mathsisfun.com/numbers/factorial.html)
- [Mathematics Stack Exchange](https://math.stackexchange.com/questions/117042/what-is-the
- [Mathematics Stack Exchange](https://math.stackexchange.com/questions/117042/what-is-the
```

As you can see, the model was able to use a recursive approach, which is more efficient and "Pythonic." It also provides a reference for the user to dive deeper into the mathematical theory behind the function.

Next, let's leverage the model's completion capabilities by initializing a function to remove non-ASCII characters.

		Definition

American Standard Code for Information Interchange (ASCII) is a character encoding standard that uses 7 bits to represent 128 characters, such as letters, digits, punctuation marks, and control codes.

Non-ASCII characters are those that are not part of the ASCII standard and use more than 7 bits to encode. They include special characters such as letters with accents, glyphs, ideograms, and mathematical symbols. Non-ASCII characters can be encoded using different standards, such as Unicode, ISO 8859-1, Windows-1252, etc.

For example, the letter é is a non-ASCII character that can be encoded using Unicode as U+00E9 or using Windows-1252 as 0xE9.

Accordingly, here is the code to generate the function:

```
prompt = '''
Define a python function that remove non-ascii character from a string.
'''
print(llm(prompt))
```

The following is the function that we receive as the output:

```
def remove_non_ascii(string):
    return string.encode('ascii', 'ignore').decode('utf-8')
```

Let's now leverage the model as a bug fixer, prompting it with the wrong function and also asking it to provide an explanation of why it is wrong and how it can be fixed:

```
prompt = """
Identify the bug in the following code:.
```
#wrong function
import random
a = random.randint(1, 12)
b = random.randint(1, 12)
for i in range(10):
    question = "What is " + a + " x " + b + "? "
    answer = input(question)
    if answer = a * b:
        print (Well done!)
    else:
        print("No.")
```
Explanation:
"""
print(llm(prompt))
```

Here is our output:

```
- The function is using the wrong syntax for the `randint` function. The correct syntax is
- The `for` loop is using the wrong syntax. The correct syntax is `for i in range(start, e
- The `question` and `answer` variables are not being used in the loop.
```

```
- The `if` statement is not being used correctly. It should be `if answer == a * b:`.
- The `print` statement is not being used correctly. It should be `print(Well done!)`.
```

Finally, let's ask the model in natural language to generate specific Python code for a given task that, in our example, will be that of writing a function that finds the longest substring of a given string containing only unique characters:

```
prompt = """
Write a Python function that finds the longest substring of a given string containing only
"""
print(llm(prompt))
```

We then get the following function as our output:

```
def longest_unique_substring(s):
    unique_count = 0
    longest_substring = ""
    for i in range(1, len(s)):
        if s[i] not in unique_count:
            unique_count += 1
            longest_substring = s[i]
    return longest_substring
```

As per the Falcon LLM, in this case we used the light version of the model (7 billion parameters), still obtaining great results. This is a perfect example of how the task you want to address with your application must be a factor in deciding what LLM to use: if you are only interested in code generation, completion, infilling, debugging, or any other code-related tasks, a light and open-source model could be more than enough, rather than 70 billion parameters of a state-of-the-art GPT-4.

In the next section, we are going to cover the third and last LLM in the context of code generation and understanding.

StarCoder

The StarCoder model is an LLM for code that can perform various tasks, such as code completion, code modification, code explanation, and technical assistance. It was trained on permissively licensed data from GitHub, including from 80+ programming languages, Git commits, GitHub issues, and Jupyter notebooks. It has a context length of over 8,000 tokens, which enables it to process more input than any other open-source language model. It also has an improved license that simplifies the process for companies to integrate the model into their products.

The StarCoder model was evaluated on several benchmarks that test its ability to write and understand code in different languages and domains, including the aforementioned HumanEval and MBPP, where the model scored, respectively, 33.6% and 52.7%. Additionally, it was tested against MultiPL-E (where the model matched or outperformed the code-cushman-001 model from OpenAI on many languages), the DS-1000 (where the model clearly beat the code-cushman-001 model as well as all other open-access models), and the Tech Assistant Prompt (where the model was able to respond to various queries with relevant and accurate information).

According to a survey published on May 4 2023 by Hugging Face, StarCoder demonstrated great capabilities compared to other models, using HumanEval and MBPP as benchmarks. You can see an illustration of this study below:

Model	HumanEval	MBPP
LLaMA-7B	10.5	17.7
LaMDA-137B	14.0	14.8
LLaMA-13B	15.8	22.0
CodeGen-16B-Multi	18.3	20.9
LLaMA-33B	21.7	30.2
CodeGeeX	22.9	24.4
LLaMA-65B	23.7	37.7
PaLM-540B	26.2	36.8
CodeGen-16B-Mono	29.3	35.3
StarCoderBase	30.4	49.0
code-cushman-001	33.5	45.9
StarCoder	33.6	52.7
StarCoder-Prompted	40.8	49.5

Figure 9.4: Results of evaluation benchmarks for various LLMs. Source: **https://huggingface.co/blog/starcoder**

To start using StarCoder, we can follow these steps:

1. We can leverage the HuggingFaceHub wrapper available in LangChain (remember to set the Hugging Face API in the `.env` file):

```
import os
from dotenv import load_dotenv
load_dotenv()
hugging_face_api = os.environ["HUGGINGFACEHUB_API_TOKEN"]
```

2. Let's set the `repo_id` for the StarCoder model and initialize it:

```
from langchain import HuggingFaceHub
from langchain import PromptTemplate, LLMChain
repo_id = "bigcode/starcoderplus"
llm = HuggingFaceHub(
    repo_id=repo_id,  model_kwargs={"temperature": 0.2, "max_new_tokens": 500}
)
```

Note

StarCoder is a gated model on the Hugging Face Hub, meaning that you will need to request access directly from the bigcode/starcoderplus repo before being able to connect to it.

Now that we're set up, let's start asking our model to compile some code. To start with, we will ask it to generate a Python function to generate the nth Fibonacci number:

```
prompt = """
How can I write a Python function to generate the nth Fibonacci number?
"""
print(llm(prompt))
```

Definition

The Fibonacci sequence is a mathematical series that begins with 0 and 1, and each subsequent number is the sum of the two preceding numbers. For instance, the first 10 numbers of the Fibonacci sequence are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34.

There are different ways to compute the nth Fibonacci number, which is denoted by F(n). One way is to use a recursive formula:

$$F(n) = F(n-1) + F(n-2)$$

This means that to find F(n), we need to find F(n-1) and F(n-2) first, and then add them together. This works for any n greater than or equal to 2. For n equal to 0 or 1, we simply return n as the answer.

We then see the following output:

```
## Solution

```python
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Explanation

This solution is a recursive function. It uses the base cases of 8 and 1 to return the first two Fibonacci numbers. Otherwise,
it calls itself with the previous two Fibonacci numbers to get the next Fibonacci number.

Alternative Solution

```python
def fibonacci(n):
    a, b = 0, 1
    for _ in range(n):
        a, bb= b, a + b
    return a
```

This solution is a loop. It uses the base cases of 0 and 1 to return the first two Fibonacci numbers. Otherwise,
it uses the previous two Fibonacci numbers to get the next Fibonacci number.

Complexity Analysis

* Time complexity: 0(n).
* Space complexity: O(n).
<|endoftext|>
```

Figure 9.5: Example of Fibonacci functions generated by StarCode

As you can see, it also proposed different approaches to solve the problem, alongside the explanation.

Let's now ask the model to generate a webpage to play tic tac toe against the computer:

```
prompt = """
Generate the html code for a single page website that let the user play tic tac toe.
 """
print(llm(prompt))
```

Here is the corresponding output:

```
 ## How to use
```
git clone https://github.com/Mohamed-Elhawary/tic-tac-toe.git
cd tic-tac-toe
python3 -m http.server
```
## License
 [MIT](https://choosealicense.com/licenses/mit/)
<|endoftext|>
```

Interestingly enough, the model in this case didn't generate the whole code; rather, it gave the instructions to clone and run a git repository that can achieve this result.

Finally, StarCoder is also available as an extension in VS Code to act as your code copilot. You can find it as **HF Code Autocomplete**, as shown in the following screenshot:

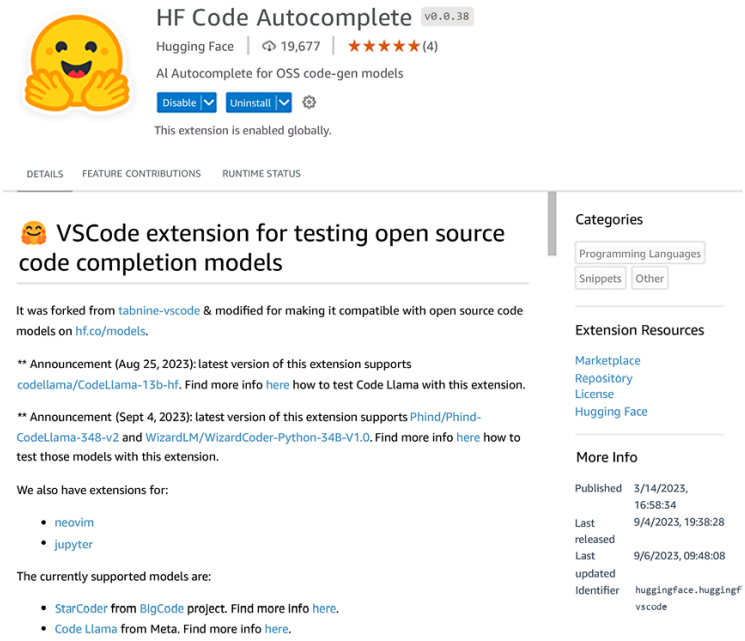


Figure 9.6: Hugging Face Code Autocomplete extension, powered by StarCoder

Once enabled, you can see that, while compiling your code, StarCoder will provide suggestions to complete the code. For example:

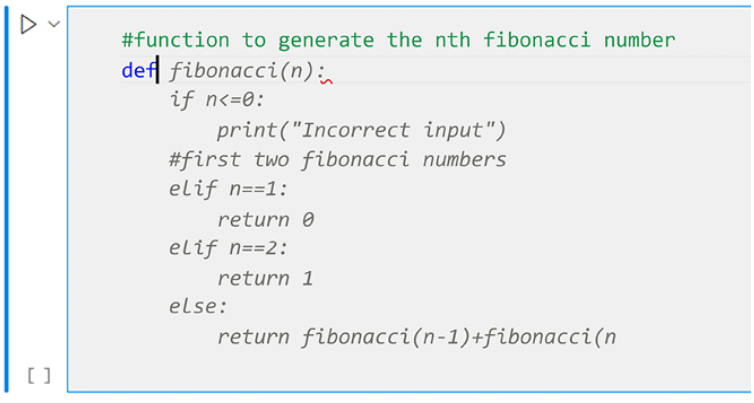


Figure 9.7: Screenshot of a suggested completion, given a function description

As you can see, I commented my code, describing a function to generate the nth Fibonacci number, and then started defining the function. Automatically, I've been provided with the StarCoder auto-completion suggestion.

Code understanding and generation are great capabilities of LLMs. On top of those capabilities, there are further applications that we can think about, going beyond code generation. In fact, the code can be seen also as a backend reasoning tool to propose solutions to complex problems, such as an energy optimization problem rather than an algorithm task. To do this, we can leverage LangChain to create powerful agents that can *act as if they were algorithms*. In the upcoming section, we will see how to do so.

Act as an algorithm

Some problems are complex by definition and difficult to solve leveraging "only" LLMs' analytical reasoning skills. However, LLMs are still intelligent enough to understand the problems overall and leverage their coding capabilities to solve them.

In this context, LangChain provides a tool that empowers the LLM to reason "in Python," meaning that the LLM-powered agent will leverage Python to solve complex problems. This tool is the Python REPL, which is a simple Python shell that can execute Python commands. The Python REPL is important because it allows users to perform complex calculations, generate code, and interact with language models using Python syntax. In this section, we will cover some examples of the tool's capabilities.

Let's first initialize our agent using the `create_python_agent` class in LangChain. To do so, we will need to provide this class with an LLM and a tool, which, in our example, will be the Python REPL:

```
import os
from dotenv import load_dotenv
from langchain.agents.agent_types import AgentType
from langchain.chat_models import ChatOpenAI
 from langchain_experimental.agents.agent_toolkits.python.base import create_python_agent
from langchain_experimental.tools import PythonREPLTool
load_dotenv()
openai_api_key = os.environ['OPENAI_API_KEY']
model = ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613")
agent_executor = create_python_agent(
    llm=model,
    tool=PythonREPLTool(),
    verbose=True,
    agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
)
```

As always, before starting to work with the agent, let's first inspect the default prompt:

```
print(agent_executor.agent.llm_chain.prompt.template)
```

Here is our output:

```
You are an agent designed to write and execute python code to answer questions.
You have access to a python REPL, which you can use to execute python code.
If you get an error, debug your code and try again.
Only use the output of your code to answer the question.
You might know the answer without running any code, but you should still run the code to get the answer.
If it does not seem like you can write code to answer the question, just return "I don't know" as the answer.


Python_REPL: A Python shell. Use this to execute python commands. Input should be a valid python command.
If you want to see the output of a value, you should print it out with `print(...)`.

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 [Python_REPL]
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!

Question: {input}
Thought:{agent_scratchpad}
```

Figure 9.8: Default prompt of the Python agent

Now, let's start with an easy query, asking the model to generate a scatter plot based on sample attributes of basketball players:

```
query = """
In a different basketball game, we have the following player stats:
- Player A: 38 points, 10 rebounds, 7 assists
- Player B: 28 points, 9 rebounds, 6 assists
- Player C: 19 points, 6 rebounds, 3 assists
- Player D: 12 points, 4 rebounds, 2 assists
- Player E: 7 points, 2 rebounds, 1 assist
Could you create a scatter plot graph in Seaborn talk mode for each player, where the y-ax
"""
agent_executor.run(query)
```

We then get the following output:

```
Invoking: `Python_REPL` with `import seaborn as sns
import matplotlib.pyplot as plt
# Player stats
players = ['Player A', 'Player B', 'Player C', 'Player D', 'Player E']
points = [38, 28, 19, 12, 7]
rebounds = [10, 9, 6, 4, 2]
# Create scatter plot
sns.scatterplot(x=rebounds, y=points, marker='o')
# Label each point with player's name
for i, player in enumerate(players):
    plt.text(rebounds[i], points[i], player, ha='center', va='bottom')
# Set title
plt.title('Team Players')
# Show the plot
plt.show()`
```

This output is accompanied by the following graph based on the players' statistics:

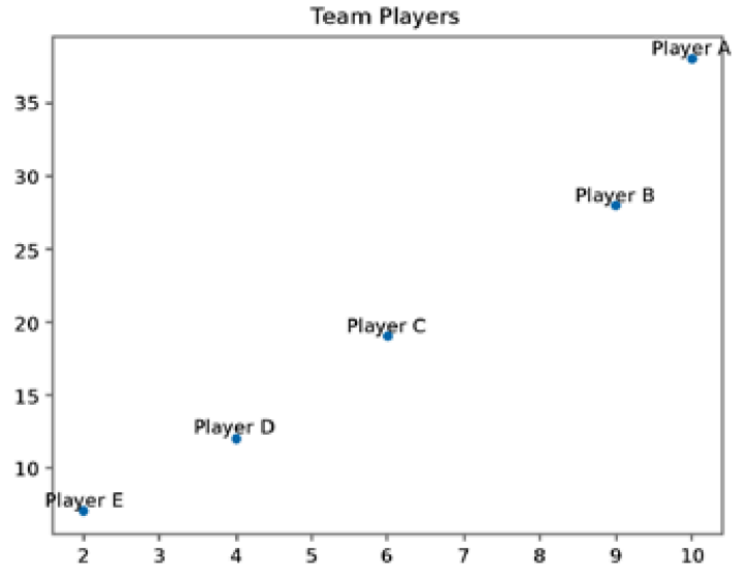


Figure 9.9: Sample plot generated by the Python agent

Let's look at another example. Say we want to predict the price of a house based on some features, such as the number of bedrooms or the size of the house. To do so, we can ask our agent to design and train a model to give us the result of a given house. For example, let's consider the following prompt:

```
query = """
I want to predict the price of a house given the following information:
- the number of rooms
- the number of bathrooms
- the size of the house in square meters
Design and train a regression model to predict the price of a house. Generate and use synt
Once the model is trained, tell me the price of a house with the following features:
- 2 rooms
- 1 bathroom
- 100 square meters
"""
agent_executor.run(query)
```

Here, we ask the agent to train a regression model on synthetic data (representative of houses with various configurations of rooms, bathrooms, and area, each with an associated price as a dependent variable) to give us the estimated price of a house with the above features. Let's see the output:

```
> Entering new AgentExecutor chain...
Invoking: `Python_REPL` with `import numpy as np
from sklearn.linear_model import LinearRegression
# Generate synthetic data
np.random.seed(0)
X = np.random.rand(100, 3)  # 100 houses with 3 features: rooms, bathrooms, size
y = 100000 * X[:, 0] + 200000 * X[:, 1] + 300000 * X[:, 2] + 50000  # Price = 100k * rooms
# Train the regression model
model = LinearRegression()
model.fit(X, y)
# Predict the price of a house with the given features
features = np.array([[2, 1, 100]])
predicted_price = model.predict(features)
predicted_price`
```

```
responded: {content}
The predicted price of a house with 2 rooms, 1 bathroom, and 100 square meters is approxim
> Finished chain.
'The predicted price of a house with 2 rooms, 1 bathroom, and 100 square meters is approxi
```

As you can see, the agent was able to generate synthetic training data, train a proper regression model using the `sklearn` libraries, and predict with the model the price of the house we provided.

With this approach, we can program an agent to act as an algorithm in real-time scenarios. Imagine, for example, that we want to design an agent that is capable of solving optimization problems in a smart building environment. The goal is to optimize the **Heating, Ventilation and Air Conditioning** (**HVAC**) setpoints in the building to minimize energy costs while ensuring occupant comfort. Let's define the variables and constraints of the problem: the objective is to adjust the temperature setpoints within the specified comfort ranges for each of the three zones while considering the varying energy costs per degree, per hour.

The goal is to strike a balance between energy efficiency and occupant comfort. Below, you can find a description of the problem and also the initialization of our variables and constraints (energy cost per zone, initial temperature per zone, and comfort range per zone):

```
query = """
**Problem**:
You are tasked with optimizing the HVAC setpoints in a smart building to minimize energy c
- Zone 1: Energy cost = $0.05 per degree per hour
- Zone 2: Energy cost = $0.07 per degree per hour
- Zone 3: Energy cost = $0.06 per degree per hour
You need to find the optimal set of temperature setpoints for the three zones to minimize
- Zone 1: 72°F
- Zone 2: 75°F
- Zone 3: 70°F
The comfort range for each zone is as follows:
- Zone 1: 70°F to 74°F
- Zone 2: 73°F to 77°F
- Zone 3: 68°F to 72°F
**Question**:
What is the minimum total energy cost (in dollars per hour) you can achieve by adjusting t
"""
agent_executor.run(query)
```

We then get the following output (you can find the whole reasoning chain in the book's GitHub repository):

```
> Entering new AgentExecutor chain...
Invoking: `Python_REPL` with `import scipy.optimize as opt
# Define the cost function
def cost_function(x):
    zone1_temp = x[0]
    zone2_temp = x[1]
    zone3_temp = x[2]

    # Calculate the energy cost for each zone
    zone1_cost = 0.05 * abs(zone1_temp - 72)
    zone2_cost = 0.07 * abs(zone2_temp - 75)
    zone3_cost = 0.06 * abs(zone3_temp - 70)
[…]
```

```
> Finished chain.
'The minimum total energy cost that can be achieved by adjusting the temperature setpoints
```

The agent was able to solve the smart building optimization problem, finding the minimum total energy cost, given some constraints. Staying in the scope of optimization problems, there are further use cases that these models could address with a similar approach, including:

- **Supply chain optimization**: Optimize the logistics and distribution of goods to minimize transportation costs, reduce inventory, and ensure timely deliveries.
- **Portfolio optimization**: In finance, use algorithms to construct investment portfolios that maximize returns while managing risk.
- **Route planning**: Plan optimal routes for delivery trucks, emergency services, or ride-sharing platforms to minimize travel time and fuel consumption.
- **Manufacturing process optimization**: Optimize manufacturing processes to minimize waste, energy consumption, and production costs while maintaining product quality.
- **Healthcare resource allocation**: Allocate healthcare resources like hospital beds, medical staff, and equipment efficiently during a pandemic or other healthcare crisis.
- **Network routing**: Optimize data routing in computer networks to reduce latency, congestion, and energy consumption.
- **Fleet management**: Optimize the use of a fleet of vehicles, such as taxis or delivery vans, to reduce operating costs and improve service quality.
- **Inventory management**: Determine optimal inventory levels and reorder points to minimize storage costs while preventing stockouts.
- **Agricultural planning**: Optimize crop planting and harvesting schedules based on weather patterns and market demand to maximize yield and profits.
- **Telecommunications network design**: Design the layout of telecommunications networks to provide coverage while minimizing infrastructure costs.
- **Waste management**: Optimize routes for garbage collection trucks to reduce fuel consumption and emissions.
- **Airline crew scheduling**: Create efficient flight crew schedules that adhere to labor regulations and minimize costs for airlines.

The Python REPL agent is amazing; however, it comes with some caveats:

- It does not allow for FileIO, meaning that it cannot read and write with your local file system.
- It forgets the variables after every run, meaning that you cannot keep trace of your initialized variables after the model's response.

To bypass these caveats, in the next section, we are going to cover an open-source project built on top of the LangChain agent: the Code Interpreter API.

Leveraging Code Interpreter

The name "Code Interpreter" was coined by OpenAI, referring to the recently developed plugin for ChatGPT. The Code Interpreter plugin allows ChatGPT to write and execute computer code in various programming

languages. This enables ChatGPT to perform tasks such as calculations, data analysis, and generating visualizations.

The Code Interpreter plugin is one of the tools designed specifically for language models with safety as a core principle. It helps ChatGPT access up-to-date information, run computations, or use third-party services. The plugin is currently in private beta and is available for selected developers and ChatGPT Plus users.

While OpenAI's Code Interpreter still doesn't offer an API, there are some open-source projects that adapted the concept of this plugin in an open-source Python library. In this section, we are going to leverage the work of Shroominic, available at **https://github.com/shroominic/codeinterpreter-api**. You can install it via `pip install codeinterpreterapi`.

According to the blog post published by Shroominic, the author of the Code Interpreter API (which you can read at **https://blog.langchain.dev/code-interpreter-api/**), it is based on the LangChain agent `OpenAIFunctionsAgent`.

Definition

OpenAIFunctionsAgent is a type of agent that can use the OpenAI functions' ability to respond to the user's prompts using an LLM. The agent is driven by a model that supports using OpenAI functions, and it has access to a set of tools that it can use to interact with the user.

The OpenAIFunctionsAgent can also integrate custom functions. For example, you can define custom functions to get the current stock price or stock performance using Yahoo Finance. The OpenAIFunctionsAgent can use the ReAct framework to decide which tool to use, and it can use memory to remember the previous conversation interactions.

The API comes already with some tools, such as the possibility to navigate the web to get up-to-date information.

Yet the greatest difference from the Python REPL tool that we covered in the previous section is that the Code Interpreter API can actually execute the code it generates. In fact, when a Code Interpreter session starts, a miniature of a Jupyter Kernel is launched on your device, thanks to the underlying Python execution environment called CodeBox.

To start using the code interpreter in your notebook, you can install all the dependencies as follows:

```
!pip install "codeinterpreterapi[all]"
```

In this case, I will ask it to generate a plot of COVID-19 cases in a specific time range:

```
from codeinterpreterapi import CodeInterpreterSession
import os
from dotenv import load_dotenv
```

```
load_dotenv()
api_key = os.environ['OPENAI_API_KEY']
# create a session
async with CodeInterpreterSession() as session:
    # generate a response based on user input
    response = await session.generate_response(
        "Generate a plot of the evolution of Covid-19 from March to June 2020, taking data
    )
    # output the response
    print("AI: ", response.content)
    for file in response.files:
        file.show_image()
```

Here is the generated output, including a graph that shows the number of global confirmed cases in the specified time period:

```
AI:  Here is the plot showing the evolution of global daily confirmed COVID-19 cases from
```

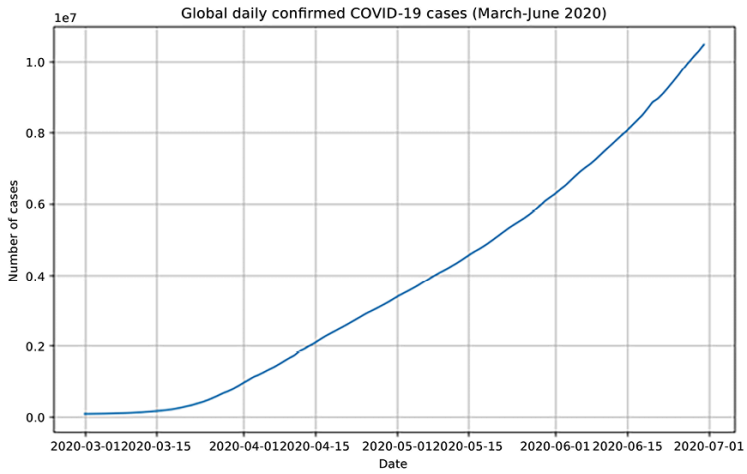


Figure 9.10: Line chart generated by the Code Intepreter API

As you can see, the Code Interpreter answered the question with an explanation as well as a plot.

Let's try another one, this time also leveraging its real-time capabilities of searching for up-to-date information. In the following snippet, we ask the model to plot the price of the S&P 500 index over the last 5 days:

```
async with CodeInterpreterSession() as session:
    # generate a response based on user input
    response = await session.generate_response(
        "Generate a plot of the price of S&P500 index in the last 5 days."
    )
    # output the response
    print("AI: ", response.content)
    for file in response.files:
        file.show_image()
```

We then get the following output, together with a line graph showing the price of the S&P 500 index over the last 5 days:

```
AI:  Here is the plot of the S&P 500 index for the last 5 days. The y-axis represents the
```

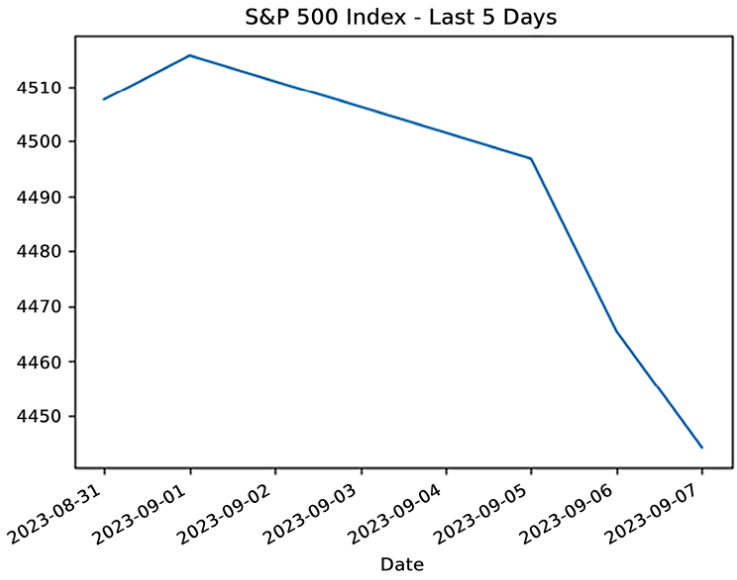


Figure 9.11: S&P 500 index price plotted by the Code Interpreter API

Finally, we can provide local files to the Code Interpreter so that it can perform some analyses on that specific data. For example, I've downloaded the Titanic dataset from Kaggle at **https://www.kaggle.com/datasets/brendan45774/test-file**. The Titanic dataset is a popular dataset for machine learning that describes the survival status of individual passengers on the Titanic. It contains information such as age, sex, class, fare, and whether they survived or not.

Once the dataset had downloaded, I passed it as a parameter to the model as follows:

```
from codeinterpreterapi import CodeInterpreterSession, File
#os.environ["HUGGINGFACEHUB_API_TOKEN"]
os.environ['OPENAI_API_KEY'] = "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"
os.environ['VERBOSE'] = "True"
async with CodeInterpreterSession() as session:
        # define the user request
        user_request = "Analyze this dataset and plot something interesting about it."
        files = [
            File.from_path("drive/MyDrive/titanic.csv"),
        ]
        # generate the response
        response = await session.generate_response(
            user_request, files=files
        )
        # output to the user
        print("AI: ", response.content)
        for file in response.files:
            file.show_image()
```

We then get the following output:

```
AI:  The plot shows the survival count based on the passenger class. It appears that passe
These are just a few examples of the kind of insights we can extract from this dataset. De
```

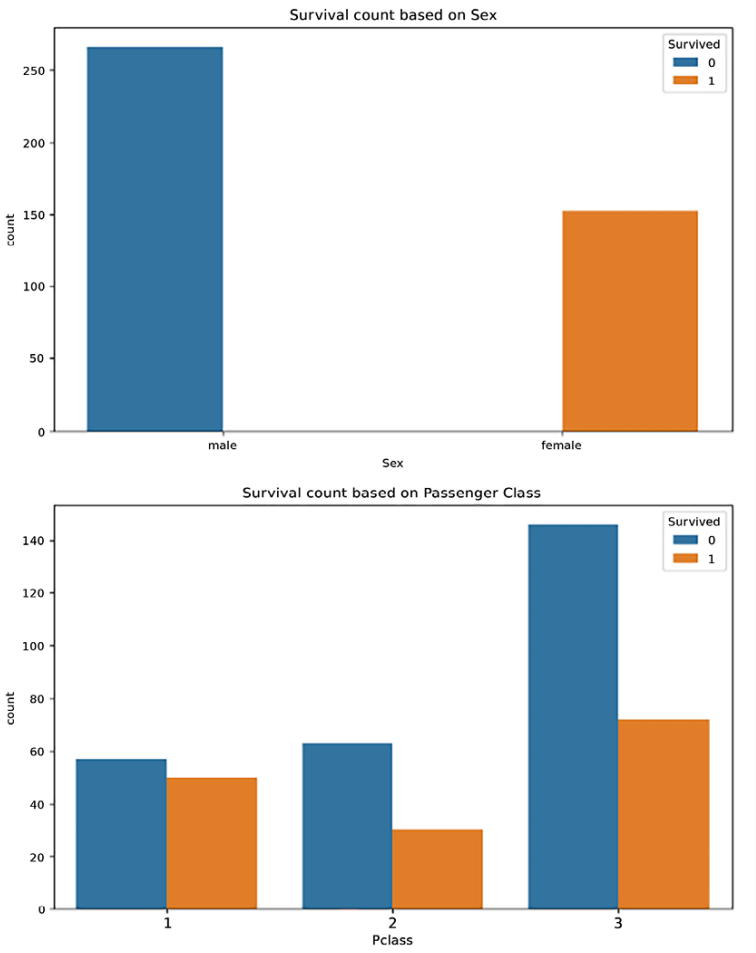


Figure 9.12: Sample plots generated by the Code Interpreter API

As you can see, the model was able to generate to bar charts showing the survival status grouped by sex (in the first plot) and then by class (in the second plot).

The Code Interpreter plugin, together with code-specific LLMs and the Python agent, are great examples of how LLMs are having a huge impact on the world of software development. This can be summarized in two main capabilities:

- LLMs can understand and generate code, since they have been trained on a huge amount of programming languages, GitHub repos, StackOverflow conversations, and so on. Henceforth, along with natural language, programming languages are part of their parametric knowledge.
- LLMs can understand a user's intent and act as a reasoning engine to activate tools like Python REPL or Code Interpreter, which are then able to provide a response by working with code.

Overall, LLMs are going well beyond the elimination of the gap between natural language and machine language: rather, they are integrating the two so that they can leverage each other to respond to a user's query.

Summary

In this chapter, we explored multiple ways in which LLMs can be leveraged to work with code. Armed with a refresher of how to evaluate LLMs and the specific evaluation benchmarks to take into account when choos-

ing an LLM for code-related tasks, we delved into practical experimentations.

We started from the "plain vanilla" application that we have all tried at least once using ChatGPT, which is code understanding and generation. For this purpose, we leveraged three different models – Falcon LLM, CodeLlama, and StarCoder – each resulting in very good results.

We then moved forward with the additional applications that LLMs' coding capabilities can have in the real world. In fact, we saw how code-specific knowledge can be used as a booster to solve complex problems, such as algorithmic or optimization tasks. Furthermore, we covered how code knowledge can not only be used in the backend reasoning of an LLM but also actually executed in a working notebook, leveraging the open-source version of the Code Interpreter API.

With this chapter, we are getting closer to the end of Part 2. So far, we have covered the multiple capabilities of LLMs, while always handling language data (natural or code). In the next chapter, we will see how to go a step further toward multi-modality and build powerful multi-modal agents that can handle data in multiple formats.

References

- The open-source version of the Code Interpreter API: https://github.com/shroominic/codeinterpreter-api
- StarCoder: https://huggingface.co/blog/starcoder
- The LangChain agent for the Python REPL: https://python.langchain.com/docs/integrations/toolkits/python
- A LangChain blog about the Code Interpreter API: https://blog.langchain.dev/code-interpreter-api/
- The Titanic dataset: https://www.kaggle.com/datasets/brendan45774/test-file
- The HF Inference Endpoint: https://huggingface.co/docs/inference-endpoints/index
- The CodeLlama model card: https://huggingface.co/codellama/CodeLlama-7b-hf
- Code Llama: Open Foundation Models for Code, *Rozière. B., et al* (2023): https://arxiv.org/abs/2308.12950
- The Falcon LLM model card: https://huggingface.co/tiiuae/falcon-7b-instruct
- The StarCoder model card: https://huggingface.co/bigcode/starcoder

Note: Have your purchase invoice ready before you begin.	5462317