4
Prompt Engineering

In *Chapter 2*, we introduced the concept of prompt engineering as the process of designing and optimizing prompts – the text input that guides the behavior of a **large language model** (**LLM**) – for LLMs for a wide variety of applications and research topics. Since prompts have a massive impact on LLM performance, prompt engineering is a crucial activity while designing LLM-powered applications. In fact, there are several techniques that can be implemented not only to refine your LLM's responses but also to reduce risks associated with hallucination and bias.

In this chapter, we are going to cover the emerging techniques in the field of prompt engineering, starting from basic approaches up to advanced frameworks. By the end of this chapter, you will have the foundations to build functional and solid prompts for your LLM-powered applications, which will also be relevant in the upcoming chapters.

We will go through the following topics:

- Introduction to prompt engineering
- Basic principles of prompt engineering
- Advanced techniques of prompt engineering

Technical requirements

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

- OpenAI account and API
- Python 3.7.1 or later version

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

What is prompt engineering?

A prompt is a text input that guides the behavior of an LLM to generate a text output.

Prompt engineering is the process of designing effective prompts that elicit high-quality and relevant output from LLMs. Prompt engineering requires creativity, understanding of the LLM, and precision.

The following figure shows an example of how a well-written prompt can instruct the same model to perform three different tasks:

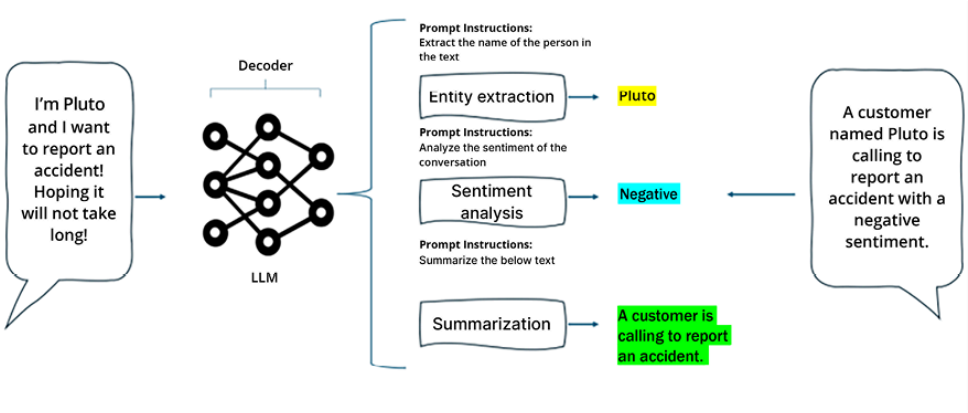

Figure 4.1: Example of prompt engineering to specialize LLMs

As you might imagine, the prompt becomes one of the key elements for an LLM-powered application's success. As such, it is pivotal to invest time and resources in this step, following some best practices and principles that we are going to cover in the next sections.

Principles of prompt engineering

Generally speaking, there are no fixed rules to obtain the "perfect" prompt since there are too many variables to be taken into account (the type of model used, the goal of the application, the supporting infrastructure, and so on). Nevertheless, there are some clear principles that have proven to produce positive effects if incorporated into the prompt. Let's examine some of them.

Clear instructions

The principle of giving clear instructions is to provide the model with enough information and guidance to perform the task correctly and efficiently. Clear instructions should include the following elements:

- The goal or objective of the task, such as "write a poem" or "summarize an article"
- The format or structure of the expected output, such as "use four lines with rhyming words" or "use bullet points with no more than 10 words each"
- The constraints or limitations of the task, such as "do not use any profanity" or "do not copy any text from the source"
- The context or background of the task, such as "the poem is about autumn" or "the article is from a scientific journal"

Let's say, for example, that we want our model to fetch any kind of instructions from text and return to us a tutorial in a bullet list. Also, if there are no instructions in the provided text, the model should inform us about that. Here are the steps:

1. First, we need to initialize our model. For this purpose, we are going to leverage OpenAI's GPT-3.5-turbo model. We first install the `openai` library:

```
$pip install openai == 0.28
```

2. To initialize the model, I used the `openai` Python library and set the OpenAI API key as the environmental variable:

```
import os
import openai
openai.api_key = os.environment.get('OPENAI_API_KEY')
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": instructions},
    ]
)
```

As you can see, the chat model comes with two variables placeholders: `system message` (or metaprompt), where we define how we want our model to behave, and `instructions` (or query), where the user will ask the model its questions.

3. Then, it takes the user's query (in this case, the text instructions). For this scenario, I set the two variables `system_message` and `instructions` as follows:

```
system_message = """
You are an AI assistant that helps humans by generating tutorials given a text.
You will be provided with a text. If the text contains any kind of istructions on how to procee
Otherwise, inform the user that the text does not contain any instructions.
Text:
"""
instructions = """
To prepare the known sauce from Genova, Italy, you can start by toasting the pine nuts to then
chop them in a kitchen mortar together with basil and garlic. Then, add half of the oil in the
Finally, transfer the pesto to a bowl and stir in the grated Parmesan cheese.
"""
```

4. Now let's test our model:

```
print(response['choices'][0]['message']['content'])
```

We get the following output:

```
Sure, I can generate a tutorial based on the given text. Here's a bullet list on how to prepare tl
- Toast the pine nuts on a pan until they are slightly browned.
- Coarsely chop the toasted pine nuts, along with basil leaves and garlic cloves, using a kitchen
- Add half of the olive oil into the kitchen mortar, and season the mixture with salt and pepper.
- Grind the mixture until it becomes a smooth paste.
- Transfer the pesto to a bowl, and stir in the grated Parmesan cheese.
- You can add more olive oil to achieve the desired consistency for the sauce.
- Your Genovese pesto sauce is now ready. It can be served with pasta, vegetables, or as a dip fo
```

5. Note that if we pass the model another text that does not contain any instructions, it will be able to respond as we instructed it:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": 'the sun is shining and dogs are running on the beach.'},
    ]
)
```

```
#print(response)
print(response['choices'][0]['message']['content'])
```

The following is the corresponding output:

```
As there are no instructions provided in the text you have given me, it is not possible to create
```

By giving clear instructions, you can help the model understand what you want it to do and how you want it to do it. This can improve the quality and relevance of the model's output and reduce the need for further revisions or corrections.

However, sometimes, there are scenarios where clarity is not enough. We might need to infer the way of thinking of our LLM to make it more robust with respect to its task. In the next section, we are going to examine one of these techniques, which will be very useful in the case of accomplishing complex tasks.

Split complex tasks into subtasks

As discussed earlier, prompt engineering is a technique that involves designing effective inputs for LLMs to perform various tasks. Sometimes, the tasks are too complex or ambiguous for a single prompt to handle, and it is better to split them into simpler subtasks that can be solved by different prompts.

Here are some examples of splitting complex tasks into subtasks:

- **Text summarization:** A complex task that involves generating a concise and accurate summary of a long text. This task can be split into subtasks such as:
  - Extracting the main points or keywords from the text
  - Rewriting the main points or keywords in a coherent and fluent way
  - Trimming the summary to fit a desired length or format
- **Machine translation:** A complex task that involves translating a text from one language to another. This task can be split into subtasks such as:
  - Detecting the source language of the text
  - Converting the text into an intermediate representation that preserves the meaning and structure of the original text
  - Generating the text in the target language from the intermediate representation
- **Poem generation**: A creative task that involves producing a poem that follows a certain style, theme, or mood. This task can be split into subtasks such as:
  - Choosing a poetic form (such as sonnet, haiku, limerick, etc.) and a rhyme scheme (such as ABAB, AABB, ABCB, etc.) for the poem
  - Generating a title and a topic for the poem based on the user's input or preference
  - Generating the lines or verses of the poem that match the chosen form, rhyme scheme, and topic
  - Refining and polishing the poem to ensure coherence, fluency, and originality
- **Code generation**: A technical task that involves producing a code snippet that performs a specific function or task. This task can be split into subtasks such as:
  - Choosing a programming language (such as Python, Java, C++, etc.) and a framework or library (such as TensorFlow, PyTorch, React, etc.) for the code
  - Generating a function name and a list of parameters and return values for the code based on the user's input or specification
  - Generating the body of the function that implements the logic and functionality of the code

- Adding comments and documentation to explain the code and its usage

Let's consider the following example in Python, where we will ask our model to generate a summary of an article:

1. We will leverage OpenAI's GPT-3.5-turbo model in a manner similar to the example discussed earlier in this chapter:

```
import os
import openai
openai.api_key = os.environ.get("OPENAI_API_KEY")
response = openai.ChatCompletion.create(
    model="gpt-35-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": article},
    ]
)
```

2. Let's set both the `system_message` and `article` variables as follows (you can find the entire scripts in the book's GitHub repository):

```
system_message = """
You are an AI assistant that summarizes articles.
To complete this task, do the following subtasks:
Read the provided article context comprehensively and identify the main topic and key points
Generate a paragraph summary of the current article context that captures the essential informa
Print each step of the process.
Article:
"""
article = """
Recurrent neural networks, long short-term memory, and gated recurrent neural networks
in particular, […]
"""
```

3. To see the output, you can run the following code:

```
print(response['choices'][0]['message']['content'])
```

Here is the obtained output:

```
Summary:
The article discusses the use of recurrent neural networks, specifically long short-term memory ar
Steps:
1. The article discusses the success and limitations of recurrent neural networks in sequence mode
2. Attention mechanisms have become popular in addressing the limitations of recurrence but are us
3. The authors propose the Transformer, a model architecture that relies solely on attention mecha
4. The Transformer can achieve state-of-the-art results in machine translation after only twelve h
```

As you can see, the model was able to produce a high-quality summary based on the key topics extracted (and displayed) from the given article. The fact that we prompted the model to split the task into subtasks "forced" it to reduce the complexity of each subtask, hence improving the quality of the final result. This approach can also lead to noticeable results when we deal

with scenarios such as mathematical problems since it enhances the analytical reasoning capabilities of the model.

Note

In a landscape of many different LLMs, it is crucial to know that the very same system message may not be as efficient in all models. A system message that perfectly works with GPT-4 might not be as efficient when applied to Llama 2, for example. Therefore, it is pivotal to design the prompt in accordance with the type of LLM you decide to pick for your application.

Splitting complex tasks into easier subtasks is a powerful technique; nevertheless, it does not address one of the main risks of LLM-generated content, that is, having a wrong output. In the next two sections, we are going to see some techniques that are mainly aimed at addressing this risk.

Ask for justification

LLMs are built in such a way that they predict the next token based on the previous ones without looking back at their generations. This might lead the model to output wrong content to the user, yet in a very convincing way. If the LLM-powered application does not provide a specific reference to that response, it might be hard to validate the ground truth behind it. Henceforth, specifying in the prompt to support the LLM's answer with some reflections and justification could prompt the model to recover from its actions. Furthermore, asking for justification might be useful also in case of answers that are right but we simply don't know the LLM's reasoning behind it. For example, let's say we want our LLM to solve riddles. To do so, we can instruct it as follows:

```
system_message = """
You are an AI assistant specialized in solving riddles.
Given a riddle, solve it the best you can.
Provide a clear justification of your answer and the reasoning behind it.
Riddle:
"""
```

As you can see, I've specified in the metaprompt to the LLM to justify its answer and also provide its reasoning. Let's see how it works:

```
riddle = """
What has a face and two hands, but no arms or legs?
"""
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": riddle},
    ]
)
#print(response)
print(response['choices'][0]['message']['content'])
```

The following is the obtained output:

```
The answer to this riddle is a clock.
A clock has a face with numbers and two hands (the hour and minute hand), which rotate around the
Hence, a clock is the correct answer to this riddle.
```

Justifications are a great tool to make your model more reliable and robust since they force it to "rethink" its output, as well as provide us with a view of how the reasoning was set to solve the problem.

With a similar approach, we could also intervene at different prompt levels to improve our LLM's performance. For example, we might discover that the model is systematically tackling a mathematical problem in the wrong way; henceforth, we might want to suggest the right approach directly at the metaprompt level. Another example might be that of asking the model to generate multiple outputs – along with their justifications – to evaluate different reasoning techniques and prompt the best one in the metaprompt.

In the next section, we are going to focus on one of these examples, more specifically, the possibility of generating multiple outputs and then picking the most likely one.

Generate many outputs, then use the model to pick the best one

As we saw in the previous section, LLMs are built in such a way that they predict the next token based on the previous ones without looking back at their generations. If this is the case, if one sampled token is the wrong one (in other words, if the model is unlucky), the LLM will keep generating wrong tokens and, henceforth, wrong content. Now, the bad news is that, unlike humans, LLMs cannot recover from errors on their own. This means that, if we ask them, they acknowledge the error, but we need to explicitly prompt them to think about that.

One way to overcome this limitation is to broaden the space of probabilities of picking the right token. Rather than generating just one response, we can prompt the model to generate multiple responses, and then pick the one that is most suitable for the user's query. This splits the job into two subtasks for our LLM:

1. Generating multiple responses to the user's query
2. Comparing those responses and picking the best one, according to some criteria we can specify in the metaprompt

Let's see an example, following up from the riddles examined in the previous section:

```
system_message = """
You are an AI assistant specialized in solving riddles.
Given a riddle, you have to generate three answers to the riddle.
For each answer, be specific about the reasoning you made.
Then, among the three answers, select the one that is most plausible given the riddle.
Riddle:
"""
riddle = """
What has a face and two hands, but no arms or legs?
"""
```

In this case, I've prompted the model to generate three answers to the riddle, then to give me the most likely, justifying why. Let's see the result:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": riddle},
    ]
)
#print(response)
print(response['choices'][0]['message']['content'])
```

We then get the following output:

```
Answer 1: A clock. A clock has a face with numbers and two hands to show minutes and hours. It doe
Answer 2: A watch. Similar to a clock, a watch also has a face with numbers and two hands to indic
Answer 3: A wall. While this answer may be less obvious, a wall does, in a way, have a face and tw
Among the three answers, the most plausible one is answer 1: a clock. The riddle distinctly mentic
```

As you can see, the model selected the most plausible answer along with a justification of its choice. It is interesting to note that "clock" and "watch" might seem similar responses; however, the model specified that "watch" is usually worn on a person's wrist and, even though it doesn't mean it has arms or legs, this element might have lowered the probability of being the correct answer.

What would you have picked?

As discussed earlier, forcing the model to tackle a problem with different approaches is a way to collect multiple samples of reasonings, which might serve as further instructions in the metaprompt. For example, if we want the model to always propose something that is not the most straightforward solution to a problem – in other words, if we want it to "think differently" – we might force it to solve a problem in N ways and then use the most creative reasoning as a framework in the metaprompt.

The last element we are going to examine is the overall structure we want to give to our metaprompt. In fact, in previous examples, we saw a sample system message with some statements and instructions. In the next section, we will see how the order and "strength" of those statements and instructions are not invariants.

Repeat instructions at the end

LLMs tend not to process the metaprompt attributing the same weight or imprortance to all the sections. In fact, in his blog post *Large Language Model Prompt Engineering for Complex Summarization*, John Stewart (a software engineer at Microsoft) found some interesting outcomes from arranging prompt sections (https://devblogs.microsoft.com/ise/gpt-summary-prompt-engineering/). More specifically, after several experimentations, he found that repeating the main instruction at the end of the prompt can help the model overcome its inner **recency bias**.

Definition

Recency bias is the tendency of LLMs to give more weight to the information that appears near the end of a prompt, and ignore or forget the information that appears earlier. This can lead to inaccurate or inconsistent responses that do not take into account the whole context of the task. For example, if the prompt is a

long conversation between two people, the model may only focus on the last few messages and disregard the previous ones.

Let's look at some ways to overcome recency bias:

- One possible way to overcome recency bias is to break down the task into smaller steps or subtasks and provide feedback or guidance along the way. This can help the model focus on each step and avoid getting lost in irrelevant details. We've covered this technique in the *Split complex tasks into subtasks* section in, which we discussed splitting complex tasks into easier subtasks.
- Another way to overcome recency bias with prompt engineering techniques is to repeat the instructions or the main goal of the task at the end of the prompt. This can help remind the model of what it is supposed to do and what kind of response it should generate.

For instance, let's say we want our model to output the sentiment of a whole chat history between an AI agent and the user. We want to make sure that the model will output the sentiment in lowercase and without punctuation.

Let's consider the following example (the conversation is truncated, but you can find the whole code in the book's GitHub repository). In this case, the key instruction is that of having as output only the sentiment in lowercase and without punctuation:

```
system_message = """
You are a sentiment analyzer. You classify conversations into three categories: positive, negative
Return only the sentiment, in lowercase and without punctuation.
Conversation:
"""
conversation = """
Customer: Hi, I need some help with my order.
AI agent: Hello, welcome to our online store. I'm an AI agent and I'm here to assist you.
Customer: I ordered a pair of shoes yesterday, but I haven't received a confirmation email yet. Ca
[…]
"""
```

In this scenario, we have key instructions before the conversation, so let's initialize our model and feed it with the two variables `system_message` and `conversation`:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": conversation},
    ]
)
#print(response)
print(response['choices'][0]['message']['content'])
```

Here is the output that we receive:

```
Neutral
```

The model didn't follow the instruction of having only lowercase letters. Let's try to repeat the instruction also at the end of the prompt:

```
system_message = f"""
You are a sentiment analyzer. You classify conversations into three categories: positive, negative
Return only the sentiment, in lowercase and without punctuation.
Conversation:
{conversation}
Remember to return only the sentiment, in lowercase and without punctuation
"""
```

Again, let's invoke our model with the updated `system_message`:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "user", "content": system_message},
    ]
)
#print(response)
print(response['choices'][0]['message']['content'])
```

Here is the corresponding output:

```
neutral
```

As you can see, now the model was able to provide exactly the output we desired. This approach is particularly useful whenever we have a conversation history to keep storing in the context window. If this is the case, having the main instructions at the beginning might induce the model not to have them in mind once it also goes through the whole history, hence reducing their strength.

Use delimiters

The last principle to be covered is related to the format we want to give to our metaprompt. This helps our LLM to better understand its intents as well as relate different sections and paragraphs to each other.

To achieve this, we can use delimiters within our prompt. A delimiter can be any sequence of characters or symbols that is clearly mapping a schema rather than a concept. For example, we can consider the following sequences to be delimiters:

- `>>>>`
- `====`
- `------`
- `####`
- `` ` ` ` ` ` ``

This leads to a series of benefits, including:

- Clear separation: Delimiters mark distinct sections within a prompt, separating instructions, examples, and desired output.
- Guidance for LLMs: Proper use of delimiters removes ambiguity, guiding the model effectively.
- Enhanced precision: Delimiters improve prompt understanding, resulting in more relevant responses.

- Improved coherence: Effective use of delimiters organizes instructions, inputs, and outputs, leading to coherent responses.

Let's consider, for example, a metaprompt that aims at instructing the model to translate user's tasks into Python code, providing an example to do so:

```
system_message = """
You are a Python expert who produces Python code as per the user's request.
===>START EXAMPLE
---User Query---
Give me a function to print a string of text.
---User Output---
Below you can find the described function:
```def my_print(text):
     return print(text)
```
<===END EXAMPLE
"""
query = "generate a Python function to calculate the nth Fibonacci number"
```

In the above example, we've used delimiters to both specify the beginning and end of an example for a one-shot learning approach and, within the example, specify the Python code snippet.

Let's see how it works:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": query},
    ]
)
#print(response)
print(response['choices'][0]['message']['content'])
```

Here is our output:

```
Sure, here's a Python function to calculate the nth Fibonacci number:
```python def fibonacci(n):
if n < 0:
return None
elif n == 0:
return 0
elif n == 1:
return 1
else:
return fibonacci(n-1) + fibonacci(n-2) ```
The function takes a non-negative integer `n` as input and returns the nth Fibonacci number. If `n
```

As you can see, it also printed the code with backticks, as shown within the system message.

All the principles examined up to this point are general rules that can make your LLM-powered application more robust. Those techniques should be kept in mind regardless of the type of application you are developing since they are general best practices that improve your LLM performance. In the following section, we are going to see some advanced techniques for prompt engineering.
```

Advanced techniques

Advanced techniques might be implemented for specific scenarios and address the way the model reasons and thinks about the answer before providing it to the final user. Let's look at some of these in the upcoming sections.

Few-shot approach

In their paper *Language Models are Few-Shot Learners*, Tom Brown et al. demonstrate that GPT-3 can achieve strong performance on many NLP tasks in a few-shot setting. This means that for all tasks, GPT-3 is applied without any fine-tuning, with tasks and few-shot demonstrations specified purely via text interaction with the model.

This is an example and evidence of how the concept of few-shot learning – which means providing the model with examples of how we would like it to respond – is a powerful technique that enables model customization without interfering with the overall architecture.

For example, let's say we want our model to generate a tagline for a new product line of climbing shoes we've just coined – Elevation Embrace. We have an idea of what the tagline should be like – concise and direct. We could explain it to the model in plain text; however, it might be more effective simply to provide it with some examples of similar projects.

Let's see an implementation with code:

```
system_message = """
You are an AI marketing assistant. You help users to create taglines for new product names.
Given a product name, produce a tagline similar to the following examples:
Peak Pursuit - Conquer Heights with Comfort
Summit Steps - Your Partner for Every Ascent
Crag Conquerors - Step Up, Stand Tall
Product name:
"""
product_name = 'Elevation Embrace'
```

Let's see how our model will handle this request:

```
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": product_name},
    ]
)
#print(response)
print(response['choices'][0]['message']['content'])
```

The following is our output:

```
Tagline idea: Embrace the Heights with Confidence.
```

As you can see, it maintained the style, length, and also writing convention of the provided taglines. This is extremely useful when you want your model to follow examples you already have, such as fixed templates.

Note that, most of the time, few-shot learning is powerful enough to customize a model even in extremely specialized scenarios, where we could think about fine-tuning as the proper tool. In fact, proper few-shot learning could be as effective as a fine-tuning process.

Let's look at another example. Let's say we want to develop a model that specializes in sentiment analysis. To do so, we provide it with a series of examples of texts with different sentiments, alongside the output we would like – positive or negative. Note that this set of examples is nothing but a small training set for supervised learning tasks; the only difference from fine-tuning is that we are not updating the model's parameters.

To provide you with a concrete representation of what was said above, let's provide our model with just two examples for each label:

```
system_message = """
You are a binary classifier for sentiment analysis.
Given a text, based on its sentiment, you classify it into one of two categories: positive or nega
You can use the following texts as examples:
Text: "I love this product! It's fantastic and works perfectly."
Positive
Text: "I'm really disappointed with the quality of the food."
Negative
Text: "This is the best day of my life!"
Positive
Text: "I can't stand the noise in this restaurant."
Negative
ONLY return the sentiment as output (without punctuation).
Text:
"""
```

To test our classifier, I've used the IMDb database of movie reviews available on Kaggle at https://www.kaggle.com/datasets/yasserh/imdb-movie-ratings-sentiment-analysis/data. As you can see, the dataset contains many movie reviews along with their associated sentiment – positive or negative. Let's substitute the binary label of 0–1 with a verbose label of Negative–Positive:

```
import numpy as np
import pandas as pd
df = pd .read_csv('movie.csv', encoding='utf-8')
df['label'] = df['label'].replace({0: 'Negative', 1: 'Positive'})
df.head()
```

This gives us the first few records of the dataset, which are as follows:

	text	label
0	I grew up (b. 1965) watching and loving the Th...	Negative
1	When I put this movie in my DVD player, and sa...	Negative
2	Why do people who do not know what a particula...	Negative
3	Even though I have great interest in Biblical...	Negative
4	Im a die hard Dads Army fan and nothing will e...	Positive

Figure 4.2: First observations of the movie dataset

Now, we want to test the performance of our model over a sample of 10 observations of this dataset:

```
df = df.sample(n=10, random_state=42)
def process_text(text):
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": system_message},
            {"role": "user", "content": text},
        ]
    )
    return response['choices'][0]['message']['content']
df['predicted'] = df['text'].apply(process_text)
print(df)
```

The following is our output:

```
                                                    text     label predicted
32823  The central theme in this movie seems to be co...  Negative  Negative
16298  An excellent example of "cowboy noir", as it's...  Positive  Positive
28505  The ending made my heart jump up into my throa...  Negative  Positive
6689   Only the chosen ones will appreciate the quali...  Positive  Positive
26893  This is a really funny film, especially the se...  Positive  Positive
36572  Sure, we all like bad movies at one time or an...  Negative  Negative
12335  Why?!! This was an insipid, uninspired and emb...  Negative  Negative
29591  This is one of those movies that has everythin...  Positive  Positive
18948  i saw this film over 20 years ago and still re...  Positive  Positive
31067  This true story of Carlson's Raiders is more o...  Negative  Negative
```

Figure 4.3: Output of a GPT-3.5 model with few-shot examples

As you can see, by comparing the `label` and `predicted` columns, the model was able to correctly classify all the reviews, without even fine-tuning! This is just an example of what you can achieve – in terms of model specialization – with the technique of few-shot learning.

Chain of thought

Introduced in the paper *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models* by Wei et al., **chain of thought** (**CoT**) is a technique that enables complex reasoning capabilities through intermediate reasoning steps. It also encourages the model to explain its reasoning, "forcing" it not to be too fast and risking giving the wrong response (as we saw in previous sections).

Let's say that we want to prompt our LLM to solve generic first-degree equations. To do so, we are going to provide it with a basic reasoning list that it might want to follow:

```
system_message = """
To solve a generic first-degree equation, follow these steps:
1. **Identify the Equation:** Start by identifying the equation you want to solve. It should be i
2. **Isolate the Variable:** Your goal is to isolate the variable 'x' on one side of the equation

   a. **Add or Subtract Constants:** Add or subtract 'b' from both sides of the equation to move
```

```
   b. **Divide by the Coefficient:** Divide both sides by 'a' to isolate 'x'. If 'a' is zero, the
3. **Simplify:** Simplify both sides of the equation as much as possible.
4. **Solve for 'x':** Once 'x' is isolated on one side, you have the solution. It will be in the f
5. **Check Your Solution:** Plug the found value of 'x' back into the original equation to ensure
6. **Express the Solution:** Write down the solution in a clear and concise form.
7. **Consider Special Cases:** Be aware of special cases where there may be no solution or infinit
Equation:
"""
```

Let's see how it can be implemented:

```
equation = "3x + 5 = 11"
response = openai.ChatCompletion.create(
    model="gpt-3.5-turbo", # engine = "deployment_name".
    messages=[
        {"role": "system", "content": system_message},
        {"role": "user", "content": equation},
    ]
)
#print(response)
print(response['choices'][0]['message']['content'])
```

The following is our output:

```
1. Identify the Equation: The equation is 3x + 5 = 11.
2. Isolate the Variable:

   a. Subtract Constants: Subtract 5 from both sides to move constants to one side:

      3x + 5 - 5 = 11 - 5

   Simplifying, we get:

      3x = 6

   b. Divide by the Coefficient: Divide both sides by 3 to isolate 'x':

      3x/3 = 6/3

   Simplifying, we get:

      x = 2

3. Simplify: Both sides are already simplified.
4. Solve for 'x': x = 2.
5. Check Your Solution: Plug x = 2 back into the original equation:
   3(2) + 5 = 11

   Simplifying, we get:

   6 + 5 = 11

   The left side equals the right side, so the solution is correct.
6. Express the Solution: The solution to the equation 3x + 5 = 11 is x = 2.
7. Consider Special Cases: Since the coefficient 3 is not zero, there are no special cases to cons
```

As you can see, the model clearly followed the seven steps specified in the metaprompt, which also allows the model to "take its time" to perform this task. Note that you can also combine it

with few-shot prompting to get better results on more complex tasks that require reasoning before responding.

With CoT, we are prompting the model to generate intermediate reasoning steps. This is also a component of another reasoning technique, which we are going to examine in the next section.

ReAct

Introduced in the paper *ReAct: Synergizing Reasoning and Acting in Language Models* by Yao et al., **ReAct** (**Reason and Act**) is a general paradigm that combines reasoning and acting with LLMs. ReAct prompts the language model to generate verbal reasoning traces and actions for a task, and also receives observations from external sources such as web searches or databases. This allows the language model to perform dynamic reasoning and quickly adapt its action plan based on external information. For example, you can prompt the language model to answer a question by first reasoning about the question, then performing an action to send a query to the web, then receiving an observation from the search results, and then continuing with this thought, action, observation loop until it reaches a conclusion.

The difference between CoT and ReAct approaches is that CoT prompts the language model to generate intermediate reasoning steps for a task, while ReAct prompts the language model to generate intermediate reasoning steps, actions, and observations for a task.

Note that the "action" phase is generally related to the possibility for our LLM to interact with external tools, such as a web search.

For example, let's say we want to ask our model for some up-to-date information about the upcoming Olympic games. To do so, we are going to build a smart LangChain agent (as described in *Chapter 2*) leveraging `SerpAPIWrapperWrapper` (to wrap the `SerpApi` to navigate the web), the `AgentType` tool (to decide which type of agent to use for our goal), and other prompt-related modules (to make it easier to "templatize" our instructions). Let's see how we can do this (I won't dive deeper into each component of the following code since the next chapter will be entirely focused on LangChain and its main components):

```
import os
from dotenv import load_dotenv
from langchain import SerpAPIWrapper
from langchain.agents import AgentType, initialize_agent
from langchain.chat_models import ChatOpenAI
from langchain.tools import BaseTool, StructuredTool, Tool, tool
from langchain.schema import HumanMessage
model = ChatOpenAI(
    model_name='gpt-35-turbo'
)
load_dotenv()
key = os.environ["SERPAPI_API_KEY"]
search = SerpAPIWrapper()
tools = [
    Tool.from_function(
        func=search.run,
        name="Search",
        description="useful for when you need to answer questions about current events"
    )
    ]
agent_executor = initialize_agent(tools, model, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbo
```

As you can see, for this purpose, I've used a pre-built agent type available in LangChain called `ZERO_SHOT_REACT_DESCRIPTION`. It comes with a precompiled prompt that follows the ReAct approach. Let's inspect that prompt:

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

Here is the corresponding output:

```
Answer the following questions as best you can. You have access to the following tools:
Search: useful for when you need to answer questions about current events
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 [Search]
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}
```

Let's now test our agent by asking something about the upcoming Olympic games and zooming in on the intermediate steps:

```
agent_executor('who are going to be the italian male athletes for climbing at the Paris 2024 Olym
```

This is the output with intermediate steps:

```
> Entering new AgentExecutor chain... I should search for recent news or updates about the Italia
>Observation: A select group of climbers ensured their participation in the 2024 Paris Olympics. A
>Input: "List of Italian male climbers Paris 2024 Olympics" Observation: Italy fielded a squad of
>Thought:This information does not seem to be readily available, I should try contacting the Itali
>Action: Contact Action Input: Email or phone call to the Italian climbing federation requesting i
>Action: Search Action Input: "Latest updates on Italian male climbers for Paris 2024 Olympics" Ob
>Thought: Based on the information I have gathered, it seems that there is no comprehensive list o
>Final Answer: Matteo Zurloni is one of the Italian male climbers who has secured a spot at the Pa
```

Here is the obtained output:

```
'Matteo Zurloni is one of the Italian male climbers who has secured a spot at the Paris 2024 Olym
```

At the time of this question (7th of October 2023), the answer is definitely correct. Note how the model went through several iterations of `Observation`/`Thought`/`Action` until it reached the conclusion. This is a great example of how prompting a model to think step by step and explicitly define each step of the reasoning makes it "wiser" and more cautious before answering. It is also a great technique to prevent hallucination.

Overall, prompt engineering is a powerful discipline, still in its emerging phase yet already widely adopted within LLM-powered applications. In the following chapters, we are going to see concrete applications of this technique.

Summary

In this chapter, we covered many aspects of the activity of prompt engineering, a core step in the context of improving the performance of LLMs within your application, as well as customizing it depending on the scenario. Prompt engineering is an emerging discipline that is paving the way for a new category of applications, infused with LLMs.

We started with an introduction to the concept of prompt engineering and why it is important, and then moved toward the basic principles – including clear instructions, asking for justification, etc. Then, we moved on to more advanced techniques that are meant to shape the reasoning approach of our LLM: few-shot learning, CoT, and ReAct.

In the next chapters, we will see those techniques in action by building real-world applications using LLMs.

References

- ReAct approach: https://arxiv.org/abs/2210.03629
- What is prompt engineering?: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-prompt-engineering
- Prompt engineering techniques: https://blog.mrsharm.com/prompt-engineering-guide/
- Prompt engineering principles: https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivots=programming-language-chat-completions
- Recency bias: https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/advanced-prompt-engineering?pivots=programming-language-chat-completions#repeat-instructions-at-the-end
- Large Language Model Prompt Engineering for Complex Summarization: https://devblogs.microsoft.com/ise/2023/06/27/gpt-summary-prompt-engineering/
- Language Models are Few-Shot Learners: https://arxiv.org/pdf/2005.14165.pdf
- IMDb dataset: https://www.kaggle.com/datasets/yasserh/imdb-movie-ratings-sentiment-analysis/code
- ReAct: https://arxiv.org/abs/2210.03629
- Chain of Thought Prompting Elicits Reasoning in Large Language Models: https://arxiv.org/abs/2201.11903

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

https://packt.link/llm