Chapter 6 - Prompt Engineering

As discussed in previous chapters, one of the ways generative models differ from embedding/representation model is that generative models need instructions to know what to do. In this context, **these instructions are known as *prompts***.

> In this chapter, we will have a deeper look into generative models, specifically into the realm of prompt engineering.

Using Text Generation Models

Choosing a Text Generation Model

Although proprietary (e.g., ChatGPT, Sonnet, Gemini) are generally better than open-source models, we focus on open-source models in this books as they offer flexibility and are free to use.

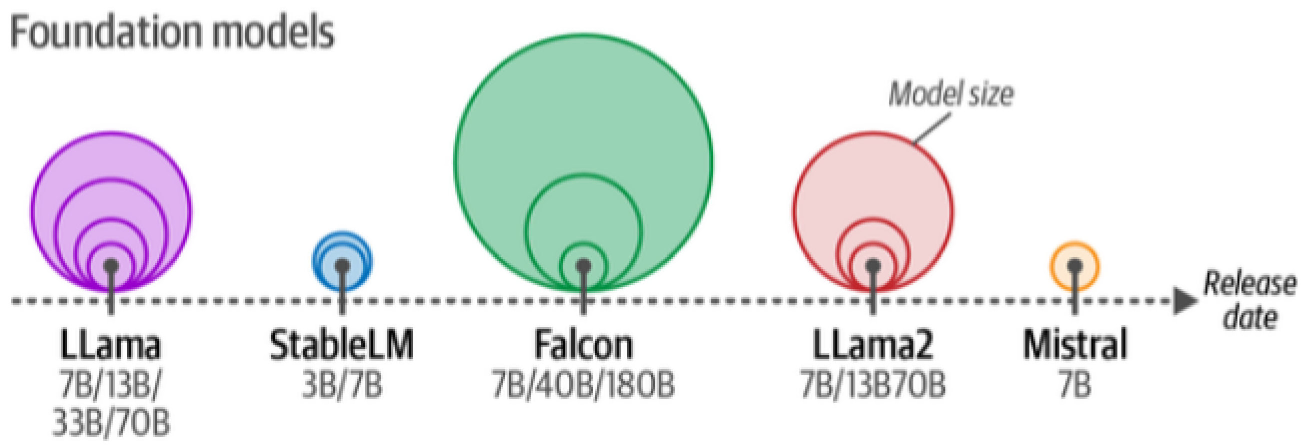


Figure 6-1. Foundation models are often released in several different sizes.

In the above image you can see some of the most popular open-source models and their sizes. As discussed in the previous chapter, models can be fine-tuned. These models and other open-source models have been fine-tuned for specific uses.

In this chapter, we will continue using `Phi-3-mini`, a model of 3.8 billion parameters.

Loading a Text Generation Model

As we have done in the earlier chapters, we will start experimenting with models by loading them. But now, we will look closer into and focus on using and developing the prompt template.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline
```

```
# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code= True,
    )

tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-
instruct")

# Create a pipeline
pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=500,
    do_sample=False,
    )
```

Let's take a look into the prompt template using the chicken prompt from Chapter 1

```
# Prompt
messages = [
    {"role": "user", "content": "Create a funny joke about chickens."}
    ]

# Generate the output
output = pipe(messages)
print(output[0]["generated_text"])
```

This outputs:

```
Why don't chickens like to go to the gym? Because they can't
crack the egg-sistence of it!
```

Under the hood, `transformers.pipeline` converts the messages into a specific prompt template. Let's see how this actually works by accessing the tokenizer:

```
# Apply prompt template
prompt = pipe.tokenizer.apply_chat_template(messages, tokenize= False)
```

```
print(prompt)
```

This outputs:

```
<s><|user|>
Create a funny joke about chickens.<|end|>
<|assistant|>
```

From the Chapter 2, you probably recognize these *special tokens*: `<|user|>` and `<|assistant|>` and `<|s|>`. Remember that these tokens provide information such as the starting point, end point, etc. But in this case, they provide information on who said what.

This prompt template was used during the training of the model. That's why each model has their own specific prompt template.

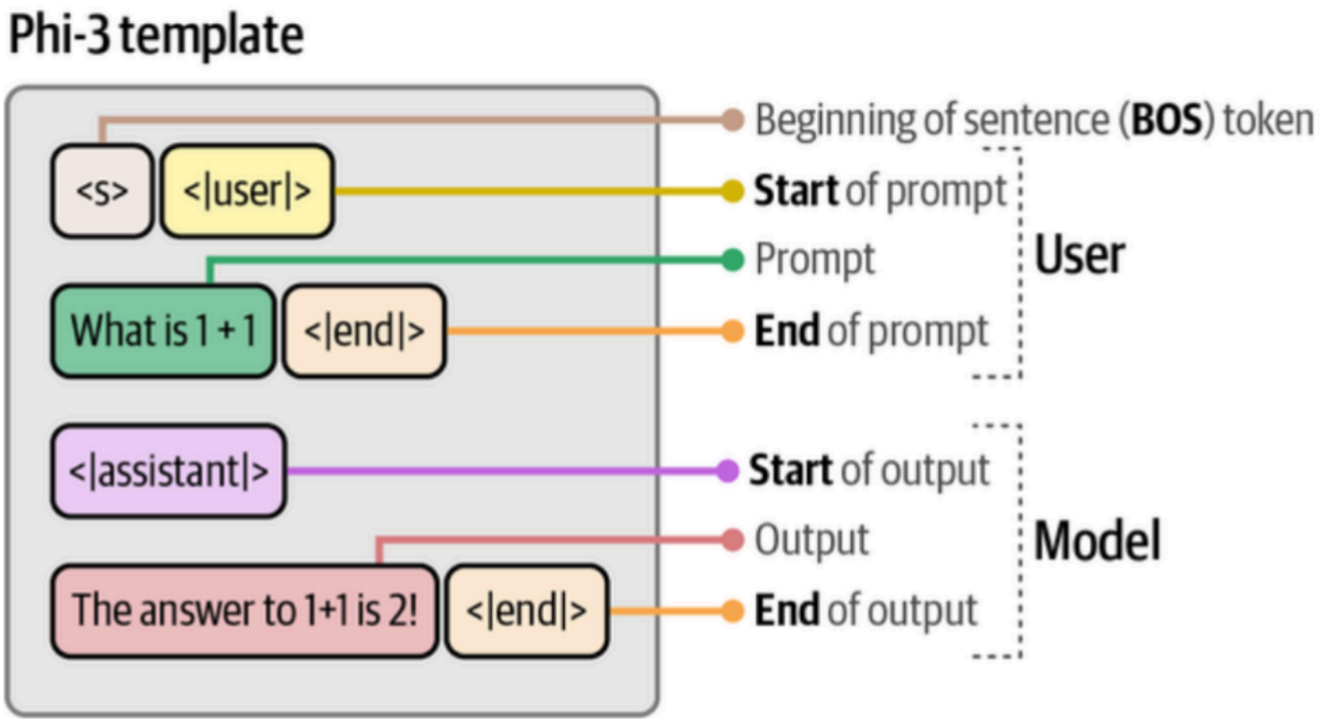


Figure 6-2. The template Phi-3 expects when interacting with the model.

Controlling Model Output

Besides prompt engineering, another way we can influence the output of the model is by changing the configurations, better known hyperparameters. Examples are `temperature` and `top_p` in `pipe`. These hyperparameters have been heavily discussed and experimented with in previous lab notebooks.

We know that `temperature` affects the *creativity* of the model in generating the next words. A good illustration is provided below:

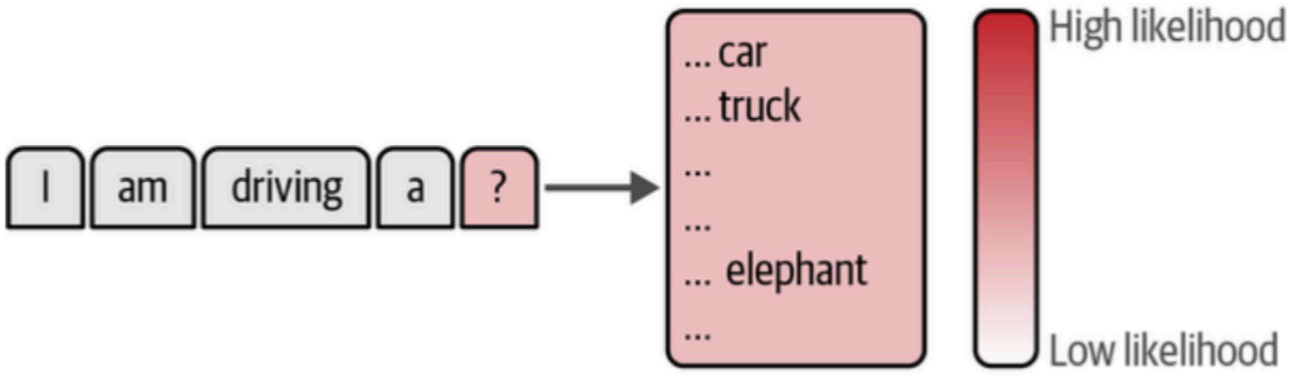


Figure 6-3. The model chooses the next token to generate based on their likelihood scores.

Another hyperparameter is `do_sample`, we set it to `False` when we want the model to pick the next word with the highest probability. To make use of `temperature` and `top_p`, we set it to `True`.

`top_p`, `top_k`, `temperature`, `do_sample`

`top_k`: Limits selection to the **k most likely tokens**. If `top_k=50`, only the 50 tokens with highest probabilities are considered.
`top_p` (nucleus sampling): Limits selection to tokens whose **cumulative probability ≤ p**. Selects the smallest set of tokens that together have p% probability mass.
If `top_p=1.0`, all tokens are considered (no filtering)
`temperature`: Controls randomness/creativity. Higher temperature → more random (flatter distribution), lower temperature → more focused (sharper distribution).
`temperature=0` is a special case: always picks the highest probability token (deterministic)
`do_sample`: When set to `False`, the model uses greedy decoding (always picks highest probability token). When `True`, samples from the distribution according to temperature/top_p/top_k.

`temperature`

As discussed many times in previous chapters, **temperature** controls the *randomness* or *creativity* of the text generated.
A temperature of 0 generates the same response every time because it always chooses the most likely word.

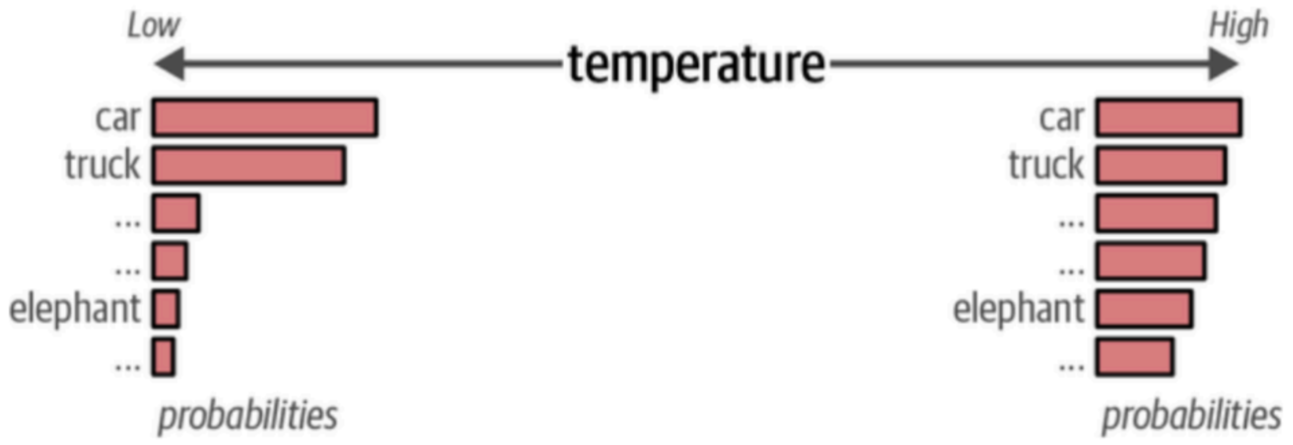


Figure 6-4. A higher **`temperature`** *increases the likelihood that less probable tokens are generated and vice versa.*

You can use `temperature` in your pipeline as follows:

```
# Using a high temperature
output = pipe(messages, do_sample=True, temperature=1)

print(output[0]["generated_text"])
```

This outputs:

```
Why don't chickens like to go on a rollercoaster? Because they're afraid
they might suddenly become chicken-soup!
```

> Because `temperature=1`, every time you run this code, the output will change.

`top_p`

`top_p`, also known as *nucleus sampling*, is a sampling technique that controls which subset of tokens (*the nucleus*) the LLM can consider.

If `top_p`=1, it will consider tokens until it reaches that value.

Example:

```
Sentence: "Have a nice..."

Next word probabilities:
"Day": 25%
"Night": 20%
"Weekend": 15%
"Trip": 10%
"Meal": 10%
"Time": 5%
```

```
"Rest": 5%
-------------
Total: 100%

If top_p=0.45, it will consider only "Day" and "Night", (0.25 + 0.20 = 0.45,
more deterministic)
```

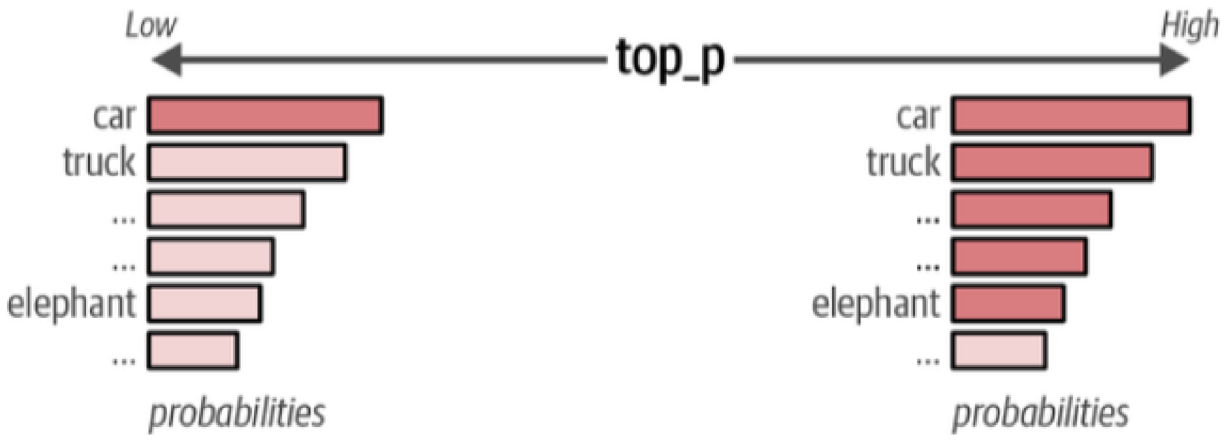


Figure 6-5. A higher `top_p` increases the number of tokens that can be selected to generate and vice versa.

Similarly, the **`top_k`** parameter controls exactly how many tokens are considered.

You can use `top_p` in your pipeline as follows:

```
# Using a high top_p
output = pipe(messages, do_sample=True , top_p=1)

print(output[0]["generated_text"])
```

This outputs:

```
Why don't chickens make good comedians? Because their 'jokes' always
'feather' the truth!
```

`top_p` and `temperature` use cases

Example Use Case	Temperature	top_p	Description
Brainstorming session	High	High	High randomness with large pool of potential tokens. The results will be highly diverse, often leading to very creative and unexpected results.

Example Use Case	Temperature	top_p	Description
Email generation	Low	Low	Deterministic output with high probable predicted tokens. This results in predictable, focused, and conservative outputs.
Creative writing	High	Low	High randomness with a small pool of potential tokens. This combination produces creative outputs but still remains coherent.
Translation	Low	High	Deterministic output with high probable predicted tokens. Produces coherent output with a wider range of vocabulary, leading to outputs with linguistic variety.

Intro to Prompt Engineering

An essential part of working with text-generative LLMs is prompt engineering.

The Basic Ingredients of a Prompt

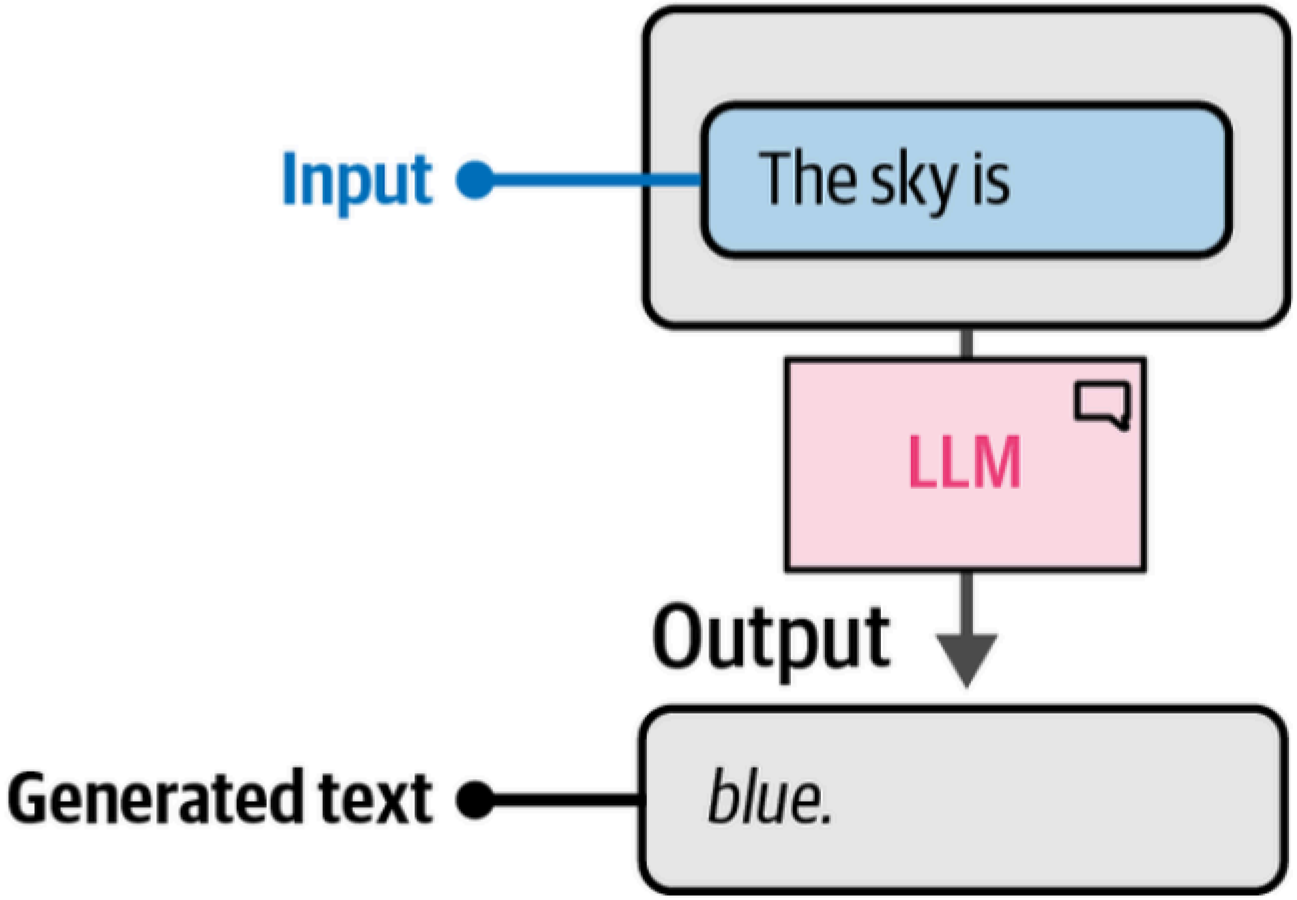


Figure 6-6. A basic example of a prompt. No instruction is given so the LLM will simply try to complete the sentence.

A basic prompt consists of two components—the instruction itself and the data that relates to the instruction.

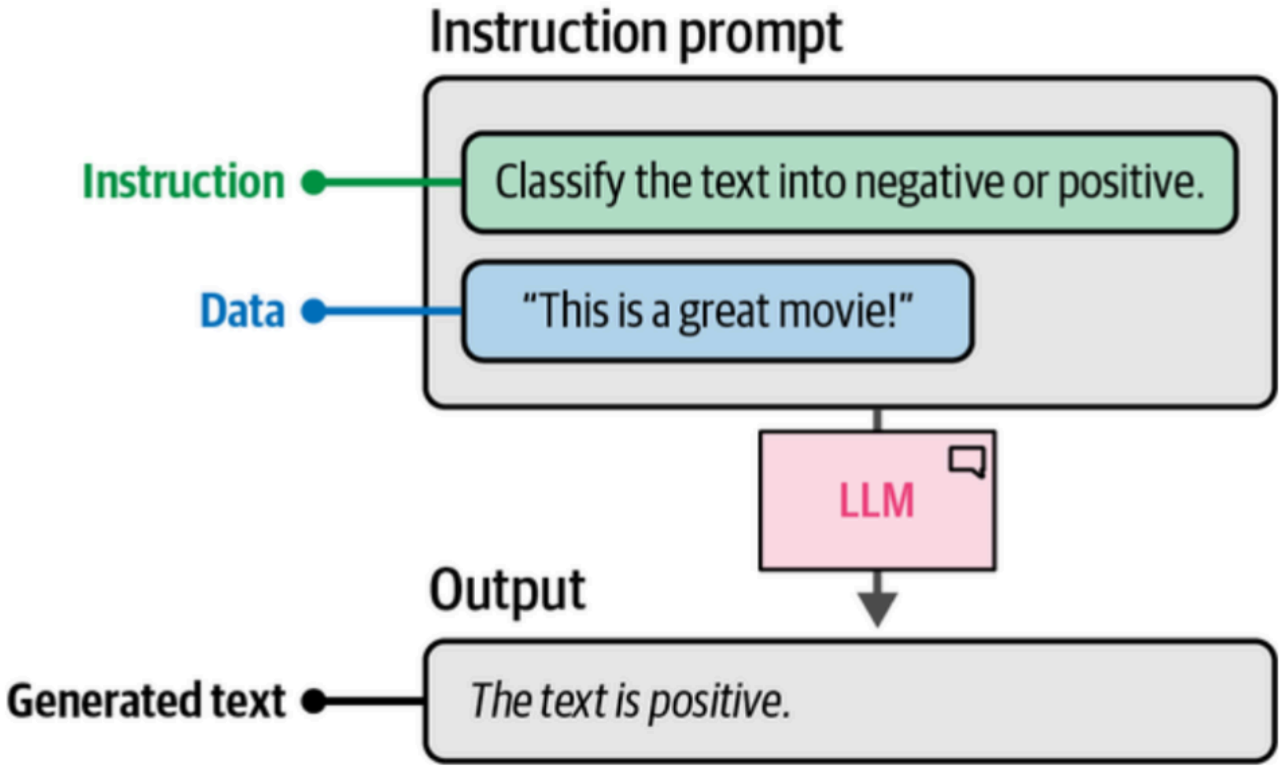


Figure 6-7. Two components of a basic instruction prompt: the instruction itself and the data it refers to.

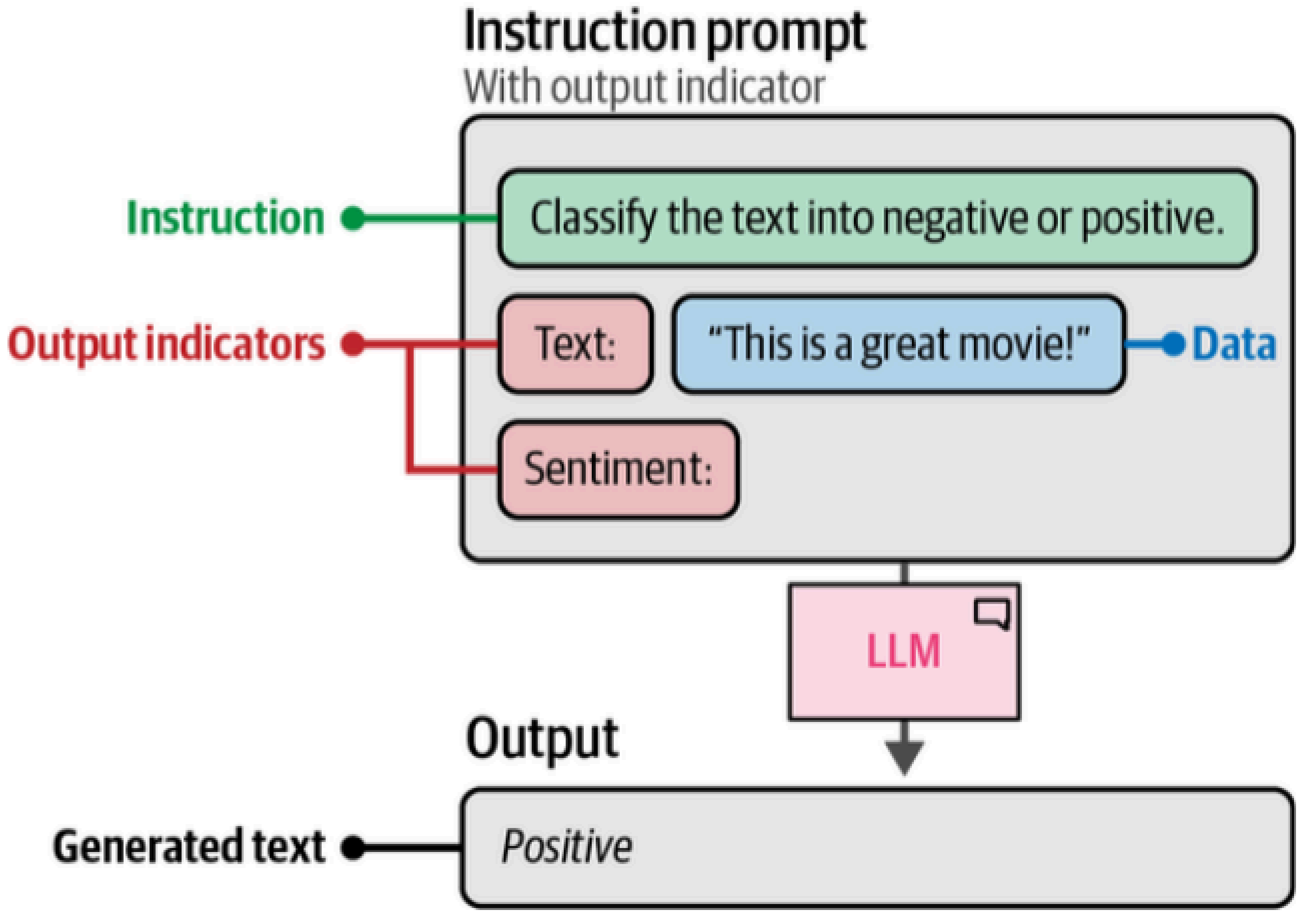


Figure 6-8. Extending the prompt with an output indicator that allows for a specific output.

Instruction-Based Prompting

This is perhaps the most common, and the only type of prompting most people know.

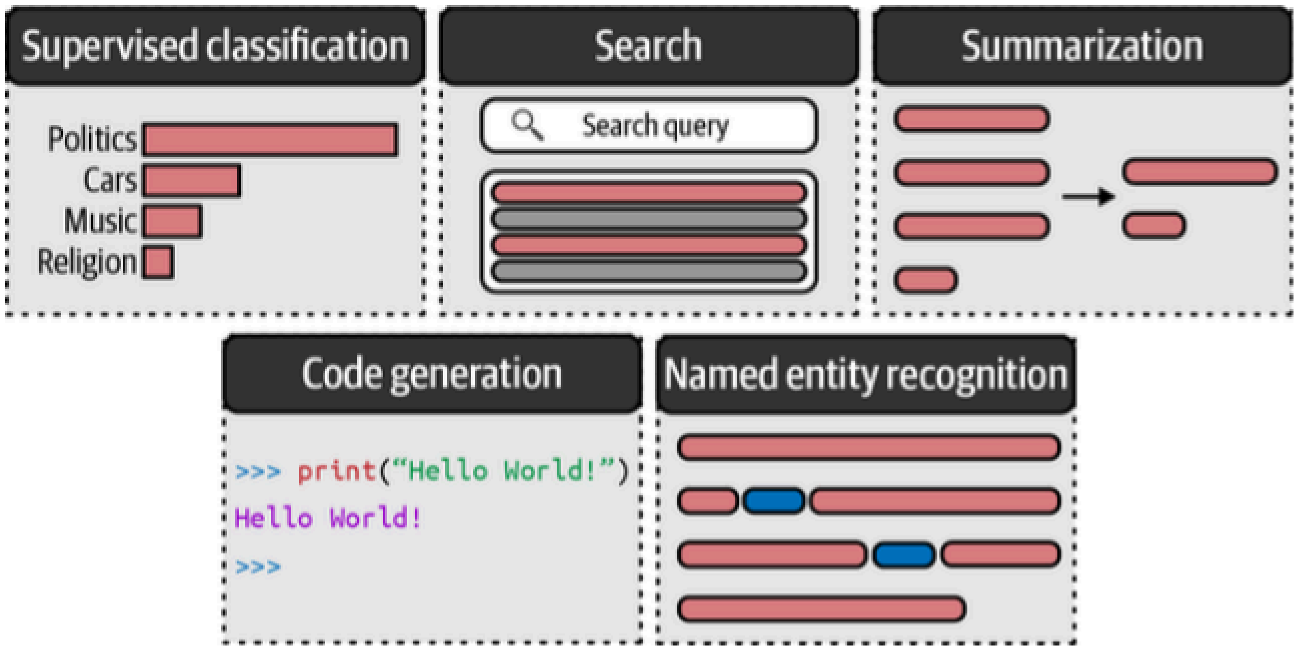
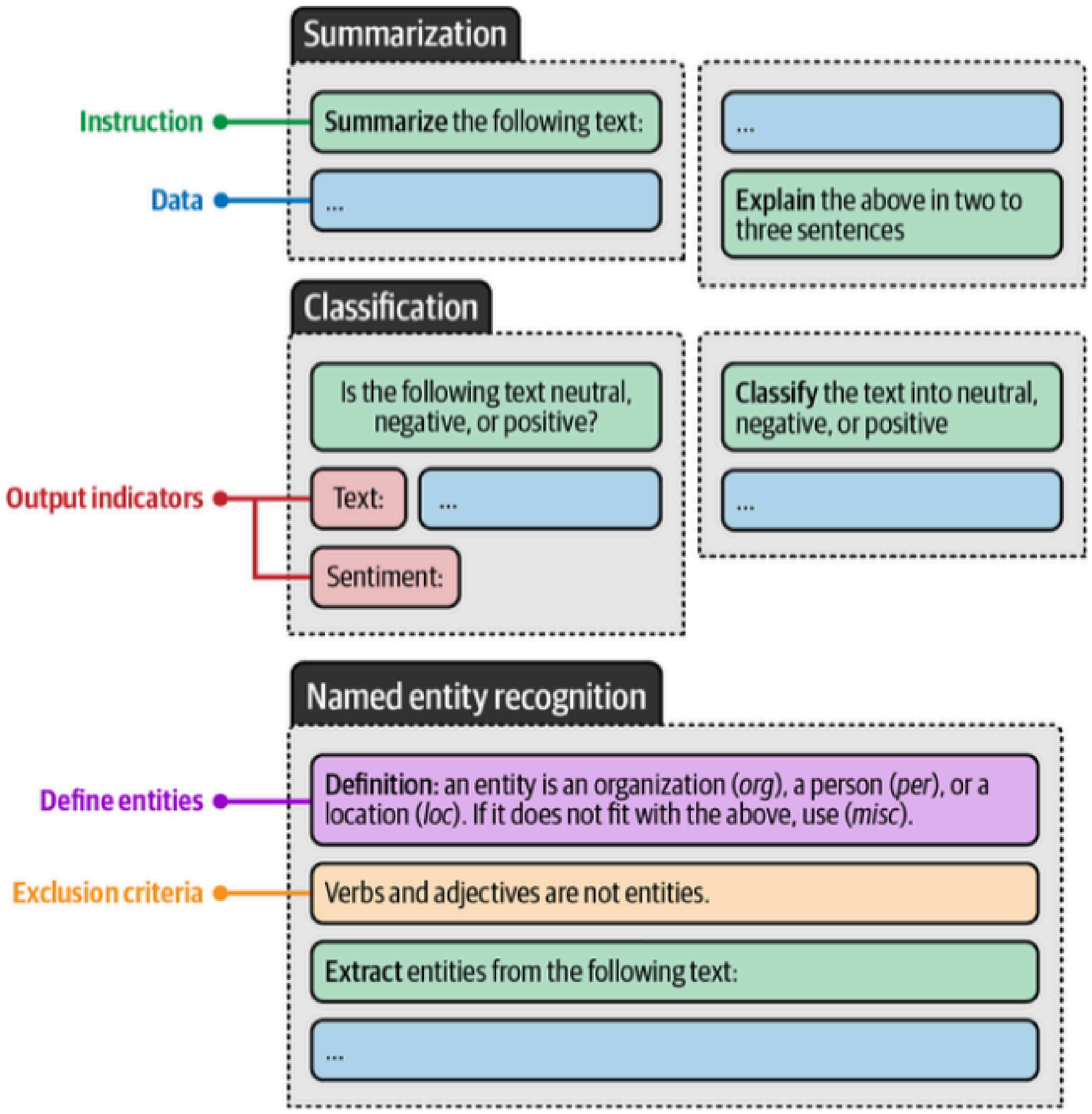


Figure 6-9. Use cases for instruction-based prompting.



Summarization
Instruction
Summarize the following text:
Data
...
...
Explain the above in two to three sentences
Classification
Is the following text neutral, negative, or positive?
Output indicators
Text:
...
Sentiment:
Classify the text into neutral, negative, or positive
...
Named entity recognition
Define entities
Definition: an entity is an organization (org), a person (per), or a location (loc). If it does not fit with the above, use (misc).
Exclusion criteria
Verbs and adjectives are not entities.
Extract entities from the following text:
...

Figure 6-10. Prompt examples of common use cases. Notice how within a use case, the structure and location of the instruction can be changed.

Although each task requires different specific instructions, there is a lot of overlap in prompting techniques to improve the quality of the output. The following are some these techniques:

- **Specificity**
  Accurately describe what you want to achieve. Instead of writing "Write a description for a product.", instead write "Write a description for a product in less than two sentences and use a formal tone." As you can see, the 2nd prompt adds more information about the request - tone and length.
- **Hallucination**
  LLMs may sometimes generate incorrect information, this is known as *hallucination*. To reduce its impact, we can ask the LLM to only generate an answer if it actually knows the answer.

- **Order**
  Either begin or end your prompt with the instruction. ***Information in the middle is often forgotten.*** LLMs to focus on information at the beginning of a prompt (*primacy effect*) or at the end of a prompt (*recency effect*).

Advanced Prompt Engineering

The Potential Complexity of a Prompt

In addition to the previously mentioned and used components such as *instructions*, *data*, and *output indicators*, there are advanced components for more complex prompts:

- **Persona**
  Describe what role the LLM should take on. It's putting the LLM in a specific character's shoes. Example: "You are a children's storyteller." If you want the LLM to use specific tone and way of writing.
- **Instruction**
  This is the task itself. This needs to be as specific as possible. The LLM should not have to interpret the instructions.
- **Context**
  These are additional information describing the context of the problem or task. It answers questions: "What is the reason for the instruction?"
- **Format**
  The format the LLM should use to output the generated text. Without it, the LLM would come up with a format itself.
- **Audience**
  The target of the generated text. This describes the level of the generated output. For example, for education purposes, it is helpful to use ELI5 ("Explain it like I'm 5")
- **Tone**
  The tone of voice the LLM should use in the generated text. If you are writing a formal email to your boss, you might not want to use an informal tone of voice.
- **Data**
  The main data related to the task itself. (e.g., dataset, QA list)

This is illustrated below.

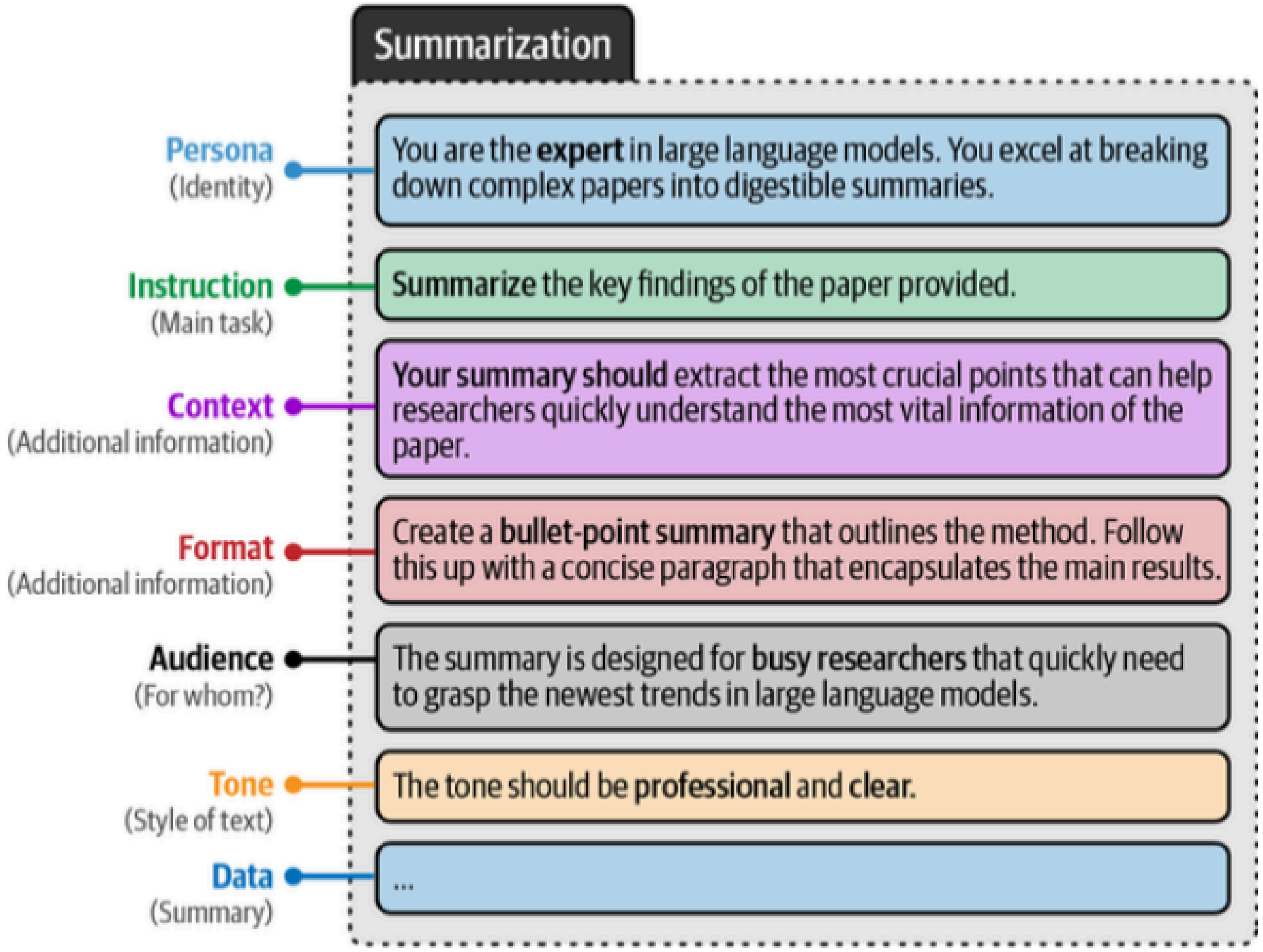

Summarization
Persona
(Identity)
You are the expert in large language models. You excel at breaking down complex papers into digestible summaries.
Instruction
(Main task)
Summarize the key findings of the paper provided.
Context
(Additional information)
Your summary should extract the most crucial points that can help researchers quickly understand the most vital information of the paper.
Format
(Additional information)
Create a bullet-point summary that outlines the method. Follow this up with a concise paragraph that encapsulates the main results.
Audience
(For whom?)
The summary is designed for busy researchers that quickly need to grasp the newest trends in large language models.
Tone
(Style of text)
The tone should be professional and clear.
Data
(Summary)
...

Figure 6-11. An example of a complex prompt with many components.

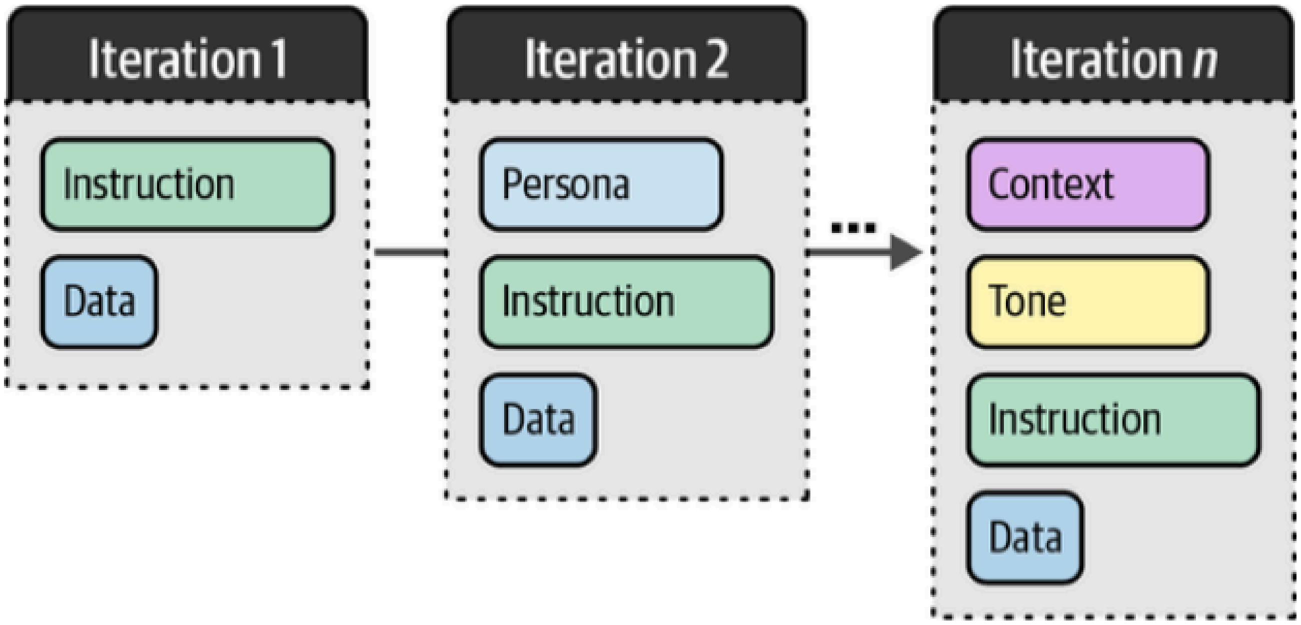


Figure 6-12. Iterating over modular components is a vital part of prompt engineering.

You can use your own data by adding it to the `data` variable:

```
# Prompt components
persona = "You are an expert in Large Language models. You excel at breaking
down complex papers into digestible summaries. "
instruction = "Summarize the key findings of the paper provided. "
context = "Your summary should extract the most crucial points that can help
researchers quickly understand the most vital information of the paper. "
data_format = "Create a bullet-point summary that outlines the method.
Follow this up with a concise paragraph that encapsulates the main results.
"
audience = "The summary is designed for busy researchers that quickly need
to grasp the newest trends in Large Language Models. "
tone = "The tone should be professional and clear. "
text = "MY TEXT TO SUMMARIZE"
data = f"Text to summarize: text "
# The full prompt - remove and add pieces to view its impact on the
generated output
query = persona + instruction + context + data_format + audience + tone +
data
```

In-Context: Providing Examples

In the previous sections, we tried to accurately describe what the LLM should do. We can provide the LLM with examples of exactly the thing that we want to achieve. This is often referred to as ***in-context learning***, where we provide the model with correct examples.

In-context learning comes in a number of forms depending on how many examples you show the LLM:

1. **Zero-shot prompt:** Prompting without examples.
2. **Few-shot prompt:** Prompting with more than one example.
3. **One-shot prompt:** Prompting with a single example

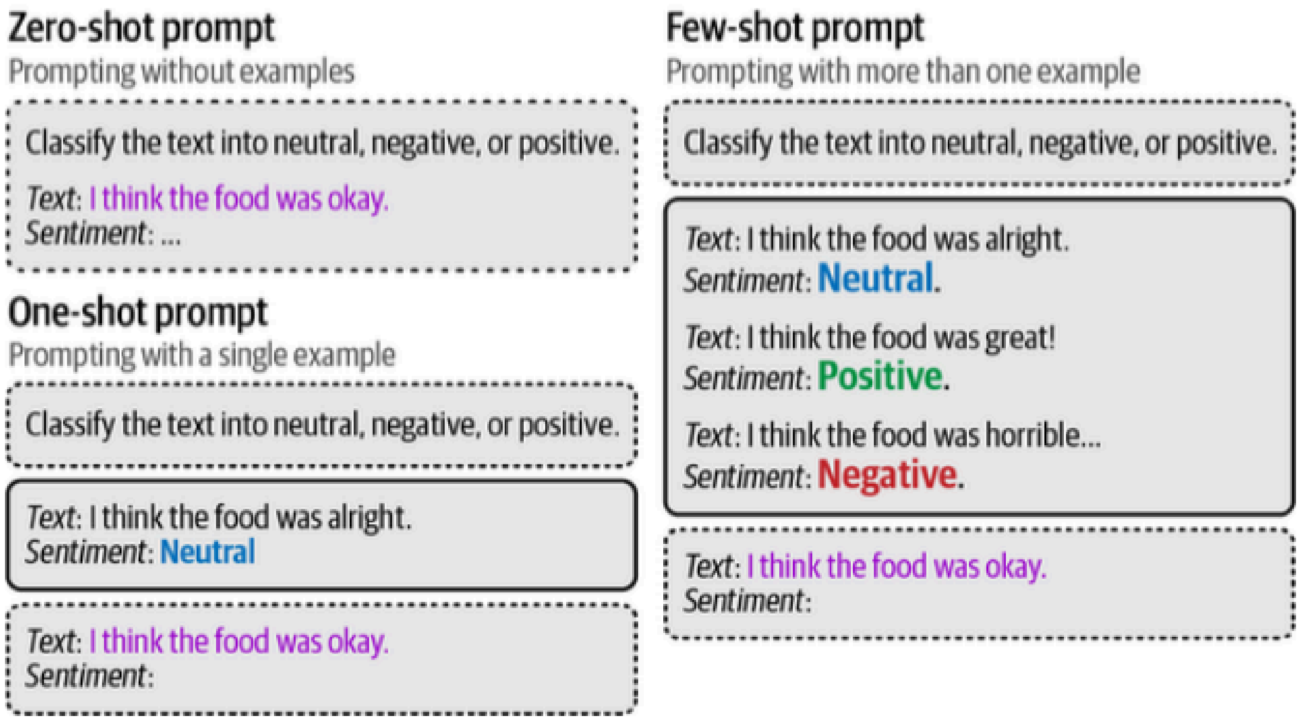

Zero-shot prompt
Prompting without examples
Classify the text into neutral, negative, or positive.
Text: I think the food was okay.
Sentiment: ...
One-shot prompt
Prompting with a single example
Classify the text into neutral, negative, or positive.
Text: I think the food was alright.
Sentiment: Neutral
Text: I think the food was okay.
Sentiment:
Few-shot prompt
Prompting with more than one example
Classify the text into neutral, negative, or positive.
Text: I think the food was alright.
Sentiment: Neutral.
Text: I think the food was great!
Sentiment: Positive.
Text: I think the food was horrible...
Sentiment: Negative.
Text: I think the food was okay.
Sentiment:

Figure 6-13. An example of a complex prompt with many components.

Below, we see an example in the prompt in action. We will need to differentiate between our (`user`) question and the answers that were provided by the model (`assistant`). The template below is used for this process:

```
one_shot_prompt = [
    {
        "role":"user",
        "content": "A 'Gigamuru' is a type of Japanese musical instrument.
An example of a sentence that uses the word Gigamuru is:"
    },
    {
        "role":"user",
        "content":"I have a Gigamuru that my uncle gave me as a gift. I love
to play it at home."
    },
    {
        "role":"user",
        "content":"To 'screeg' something is to swing a sword at it. An
example of a sentence that uses the word screeg is:"
    }
]

print(tokenizer.apply_chat_template(one_shot_prompt, tokenize=False))
```

This outputs:

```
<s><|user|>
A 'Gigamuru' is a type of Japanese musical instrument. An example of a
sentence that uses the word Gigamuru is:<|end|>
<|assistant|>
I have a Gigamuru that my uncle gave me as a gift. I love to play it at
home.<|end|>
<|user|>
To 'screeg' something is to swing a sword at it. An example of a sentence
that uses the word screeg is:<|end|>
<|assistant|>
```

As you can see, it is important to the tokenizer and the LLM to have different roles like `user` and `assistant`. Without them, it would look like it is a conversation of a single person.

Using the conversation above, you can generate the output as follows:

```
outputs = pipe(one_shot_prompt)
print(output[0]["generated_text"])
```

This outputs:

```
During the intense duel, the knight skillfully screeged his opponent's
shield, forcing him to defend himself.
```

As you can see, this generated the answer.

This demonstrates the effectiveness of one- or few-shot prompting.

Chain Prompting: Breaking up the Problem

In the previous examples, we deconstructed the prompts into multiple components within a prompt (i.e., persona, instruction, tone, etc.).

While that works for most simple use cases, this technique will not be enough for highly complex prompts.

As an example, we can ask the LLM to create a product name, slogan, and sales pitch. While this can probably be done in one prompt, we can break it up into pieces. Doing this can also improve the quality of the response. This process is illustrated in the diagram below:

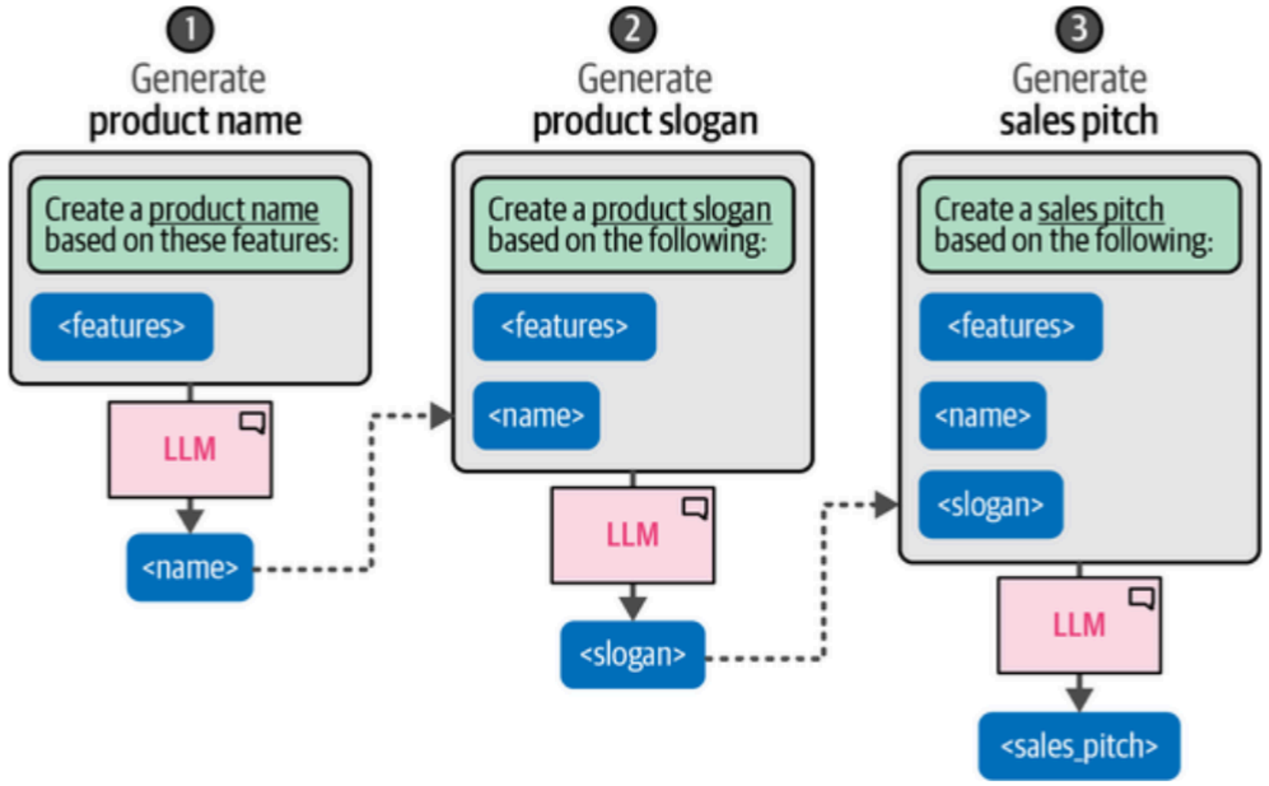


Figure 6-14. Using a description of a product's features, chain prompts to create a suitable name, slogan, and sales pitch.

This technique of chaining prompts allows the LLM to focus fully on each individual question instead of dealing with multiple requirements at once. This reduces cognitive load and allows each step to build on the previous one, resulting in higher quality responses. Additionally, this technique addresses the 'lost in the middle' problem where information in the middle of long prompts receives less attention than the beginning and end.

An example in python is provided below:

```python
product_prompt = [
    {"role":"user", "content": "Create a name and slogan for a chatbot that leverages LLMs."}
]

outputs = pipe(product_prompt)
product_description = outputs[0]["generated_text"]
print(product_description)
```

This outputs:

```
Name: 'MindMeld Messenger'
Slogan: 'Unleashing Intelligent Conversations, One Response at a Time'
```

Using the generated output above, we can prompt the LLM to generate a sales pitch:

```
# Based on the name and the slogan for a product, generate a sales pitch:

sales_prompt = [
    {"role":"user", "content": f"Generate a very short sales pitch for the following product: '{product_description}'"}
]

outputs = pipe(sales_prompt)
sales_pitch = outputs[0]["generated_text"]
print(sales_pitch)
```

This outputs:

```
Introducing MindMeld Messenger - your ultimate communication partner!
Unleash intelligent conversations with our innovative AI-powered messaging
platform. With MindMeld Messenger, every response is thoughtful,
personalized, and timely. Say goodbye to generic replies and hello to
meaningful interactions. Elevate your communication game with MindMeld
Messenger - where every message is a step toward smarter conversations. Try
it now and experience the future of messaging!
```

In addition to the previously mentioned benefits of using chaining, a major one is that we can give each call different parameters. For instance, **the number of tokens** created was relatively small for the name and slogan whereas the pitch can be much longer.

In addition to that, parameters like *temperature, top_p*, etc. can also be adjusted according to the prompt, isolated from the entire prompt.

This can be used for a variety of use cases, including:

- ***Response validation***
  Ask the LLM to double check previously generated outputs.
- ***Parallel prompts***
  Create multiple prompts in parallel and do a final pass to merge them. For example, ask multiple LLMs to generate multiple recipes in parallel and use the combined result to create a shopping list.
- ***Writing Stories***
  Leverage the LLM to write books or stories by breaking down the problem into components. For example, by first writing a summary, developing characters, and building the story beats before diving into creating the dialogue.

Reasoning with Generative Models

Reasoning is a core component of human intelligence and is often compared to the emergent behavior of LLMs that often `resembles` reasoning.

Note that we use the word `resemble`. As of the time of writing, outputs of LLMs are simply results of the data it has been trained on, and pattern matching.

We try to mimic actual reasoning by using prompt engineering.

To be able to effectively mimic this reasoning behavior, we first have to understand what actually constitutes `reasoning`.

```
Oue methods of reasoning can be divided into system 1 and system 2 thinking
processes:
```

- **System 1:** represents an automatic, intuitive, and near-instantaneous process. *It shares similarities with generative models that automatically generate tokens without any self-reflective behavior.*
- **System 2:** is a conscious, slow, and logical process, akin to brainstorming and *self-reflection*.

Chain-of-Thought: Think Before Answering

The first and major step toward complex reasoning in generative models was through a method called chain-of-thought. Chain-of-thought aims to have the generative model *think first* rather than answering the question directly without any reasoning.

This is illustrated in the diagram below. The reasoning processes are referred to as "thoughts". Adding this reasoning step allows the model to distribute more compute over the reasoning process.

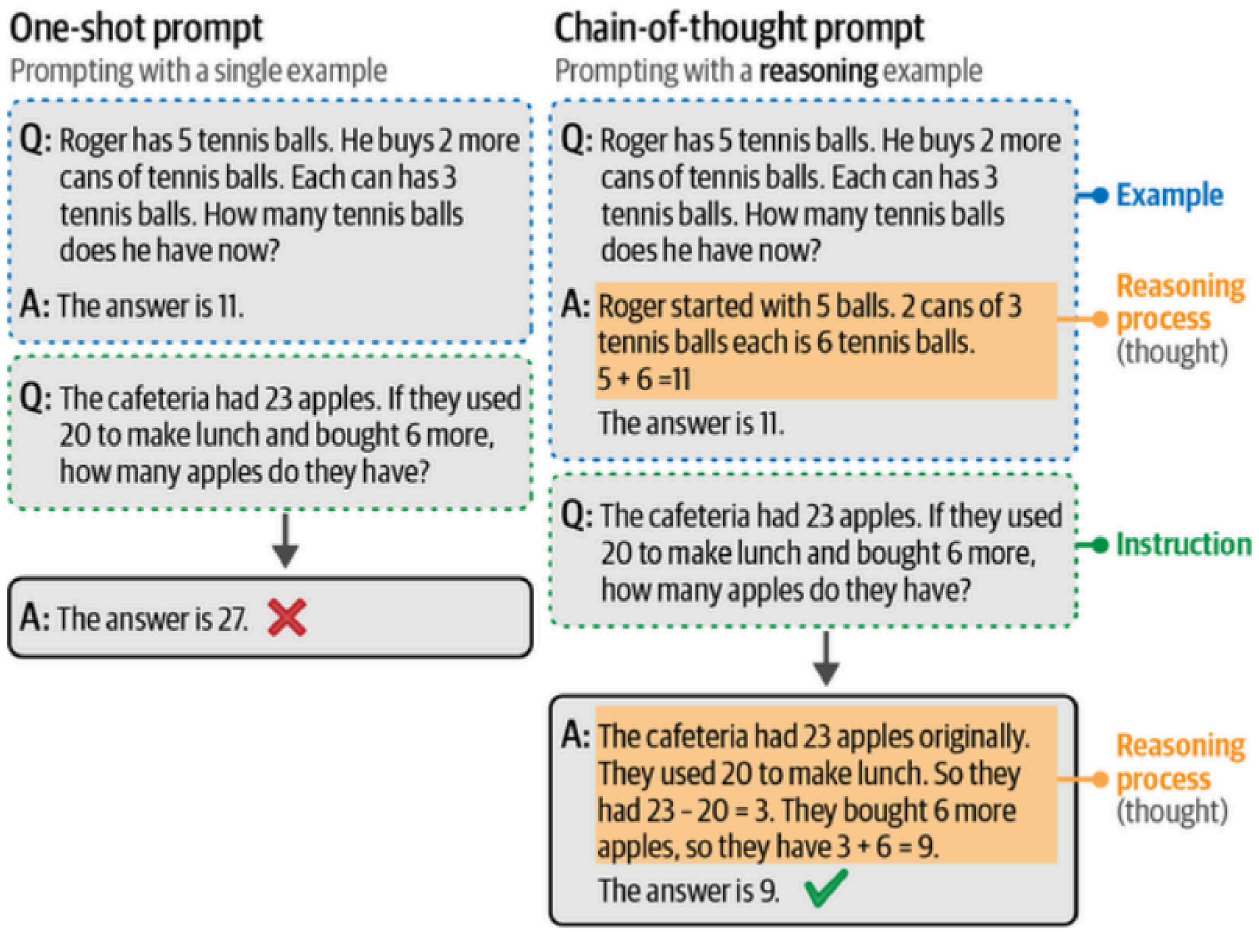

One-shot prompt
Prompting with a single example
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
A: The answer is 27.
Chain-of-thought prompt
Prompting with a reasoning example
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 =11
The answer is 11.
Example
Reasoning process (thought)
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Instruction
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9.
The answer is 9.
Reasoning process (thought)

Figure 6-15. Chain-of-thought prompting uses reasoning examples to persuade the generative model to use reasoning in its answer.

We use the example the authors used in their paper to demonstrate this:

```
# Answering with chain-of-thought
cot_prompt = [
    {"role":"user","content":"Roger has 5 tennis balls. He buys 2 more cans
of tennis balls. Each can has 3 tennis balls. How many tennis balls does he
have now?"},
    {"role":"assitant", "content":"Roger started with 5 balls. 2 cans of 3
tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11."},
    {"role":"user", "content":"The cafeteria had 3 apples. If they used 20
to make lunch and bought 6 more, how many apples do they have?"}
]
```

Generate output:

```
outputs = pipe(cot_prompt)
print(outputs[0]["generated_text"])
```

This outputs:

```
The cafeteria started with 23 apples. They used 20 apples, so they had 23 -
20 - 3 apples left. Then they bought 6 more apples, so now they have 3 + 6 =
9 apples. The answer is 9.
```

As you can see in the example above, the response is not just the answer itself, but it also includes a step-by-step *'reasoning'*.

Although this is a great method of enhancing the output, it requires one or more examples of reasoning in the prompt.

Instead of having to provide multiple examples, we can instead use ***zero-shot chain-of-thought.***

Zero-shot COT is illustrated below:

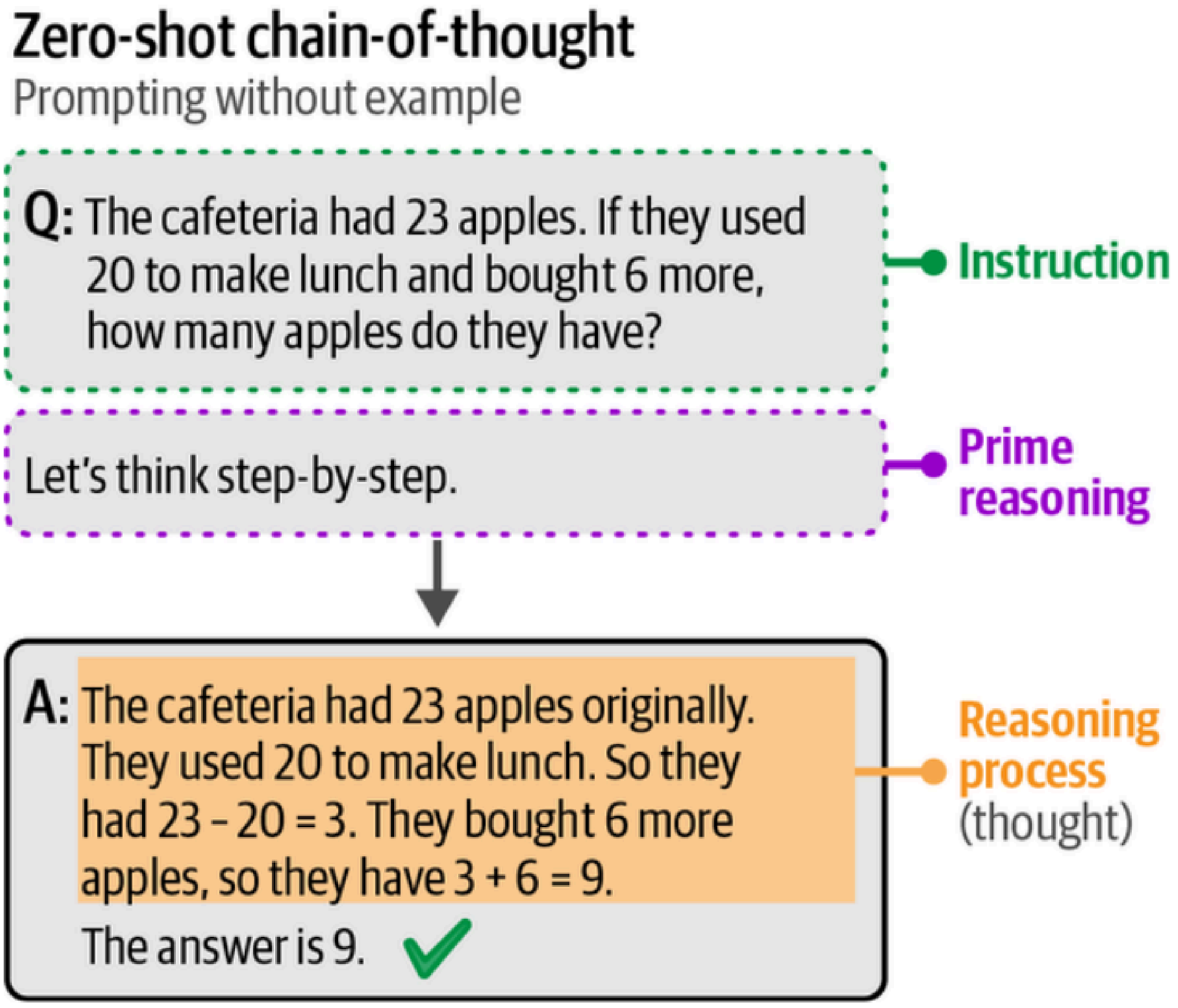

Zero-shot chain-of-thought
Prompting without example
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Instruction
Let's think step-by-step.
Prime reasoning
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9.
The answer is 9.
Reasoning process (thought)

Figure 6-16. Chain-of-thought prompting without using examples. Instead, it uses the phrase "Let's think step-by-step" to prime reasoning in its answer.

In this example, instead of providing examples, we instead give it the phrase *"Let's think step-by-step."*

```
# Zero-shot chain-of-thought
zeroshot_cot_prompt = [
    {"role":"user","content":"The cafeteria had 23 apples. If they used 20 
to make lunch and bought 6 more, how many apples do they have? Let's think 
step-by-step."}
]
```

Generate output:

```
outputs = pipe(zeroshot_cot_prompt)
print(outputs[0]["generated_text"])
```

This outputs:

```
Step 1: Start with the initial number of apples, which is 23.
Step 2: Subtract the number of apples used to make lunch, which is 20. So, 
23 - 20 = 3 apples remaining.
Step 3: Add the number of apples bought, which is 6. So, 3 + 6 = 9 apples.

The cafeteria has now 9 apples.
```

As you can see, even without any example, the model is able to generate an output similar to the one where examples were used.

TIP

Although in the example *'Let's think step-by-step'* is used, other instructions can also be used.

Alternatives like *'Take a deep breath and think step-by-step'*, and *'Let's work through this problem step-by-step'* are also used.

Self-Consistency: Sampling Outcomes

Using the same prompt can lead to different prompts if we configure parameters like `top_p` and `temperature` to allow for more *creativity*.

To counteract this, ***self-consistency was introduced***. It works by prompting the model multiple times using the same prompt, and by taking the majority result as the final result. During this process, each answer could be affected by different `temperature` and `top_p` values to increase diversity.

Note that only **extractable information** is considered, not every word itself. This is illustrated below.

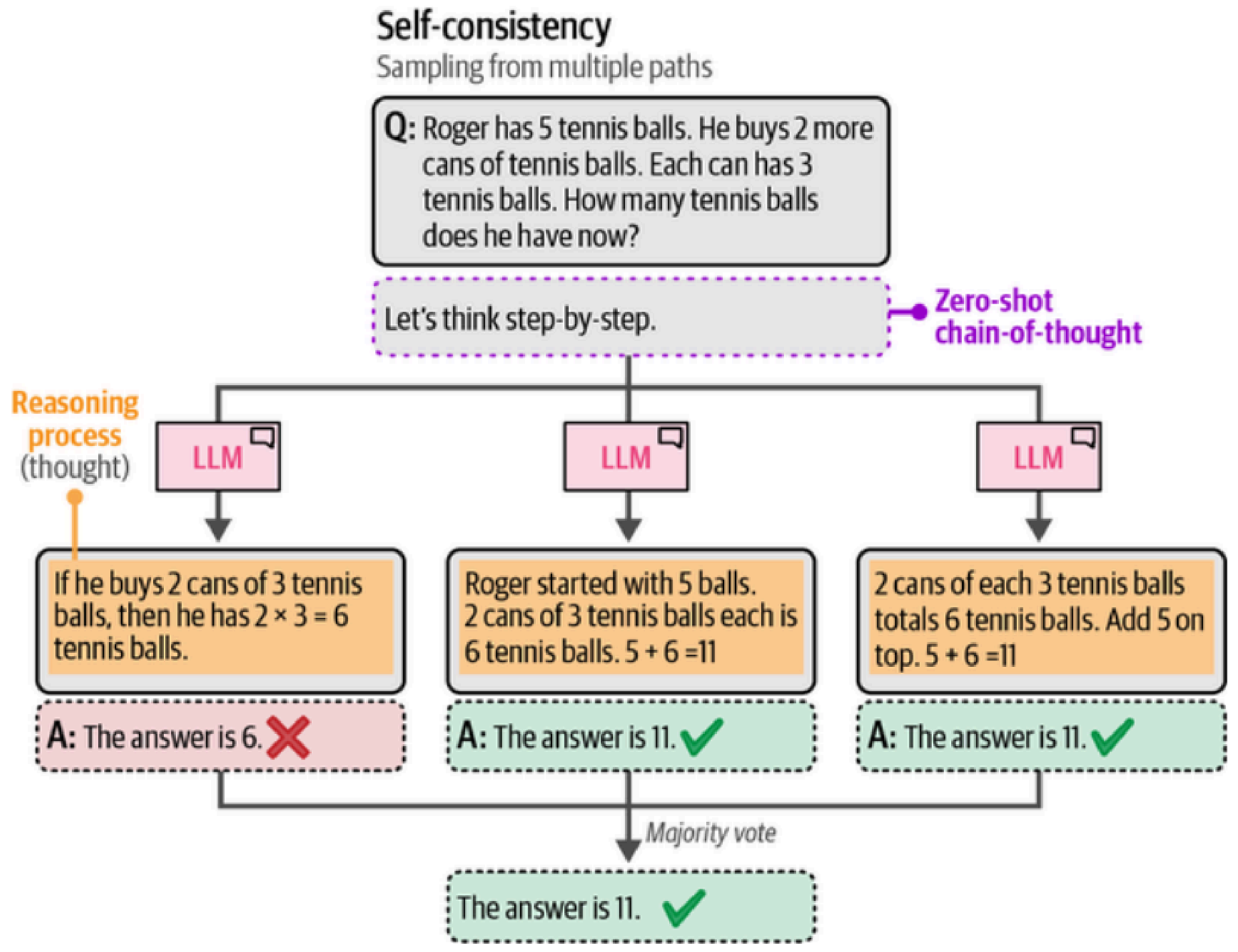

Self-consistency
Sampling from multiple paths
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
Let's think step-by-step.
Zero-shot chain-of-thought
Reasoning process (thought)
LLM
LLM
LLM
If he buys 2 cans of 3 tennis balls, then he has 2 × 3 = 6 tennis balls.
Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 =11
2 cans of each 3 tennis balls totals 6 tennis balls. Add 5 on top. 5 + 6 =11
A: The answer is 6.
A: The answer is 11.
A: The answer is 11.
Majority vote
The answer is 11.

Figure 6-17. By sampling from multiple reasoning paths, we can use majority voting to extract the most likely answer.

An obvious issue with this technique is the need to prompt the model multiple times. Prompting the model *n* times slows down the model by *n* times.

Tree-of-Thought: Exploring Intermediate Steps

While the techniques ***chain-of-thought*** and ***self-consistency*** enable more complex reasoning by sampling from multiple thought patterns, ***tree-of-thought*** takes this to a deeper level.

Tree-of-thought works as follows:

- When faced with a problem that requires multiple reasoning steps, it often helps to break it down to pieces.

- At each step, the model is prompted to explore different solutions at hand. It then votes for the best solution and continues to the next step.

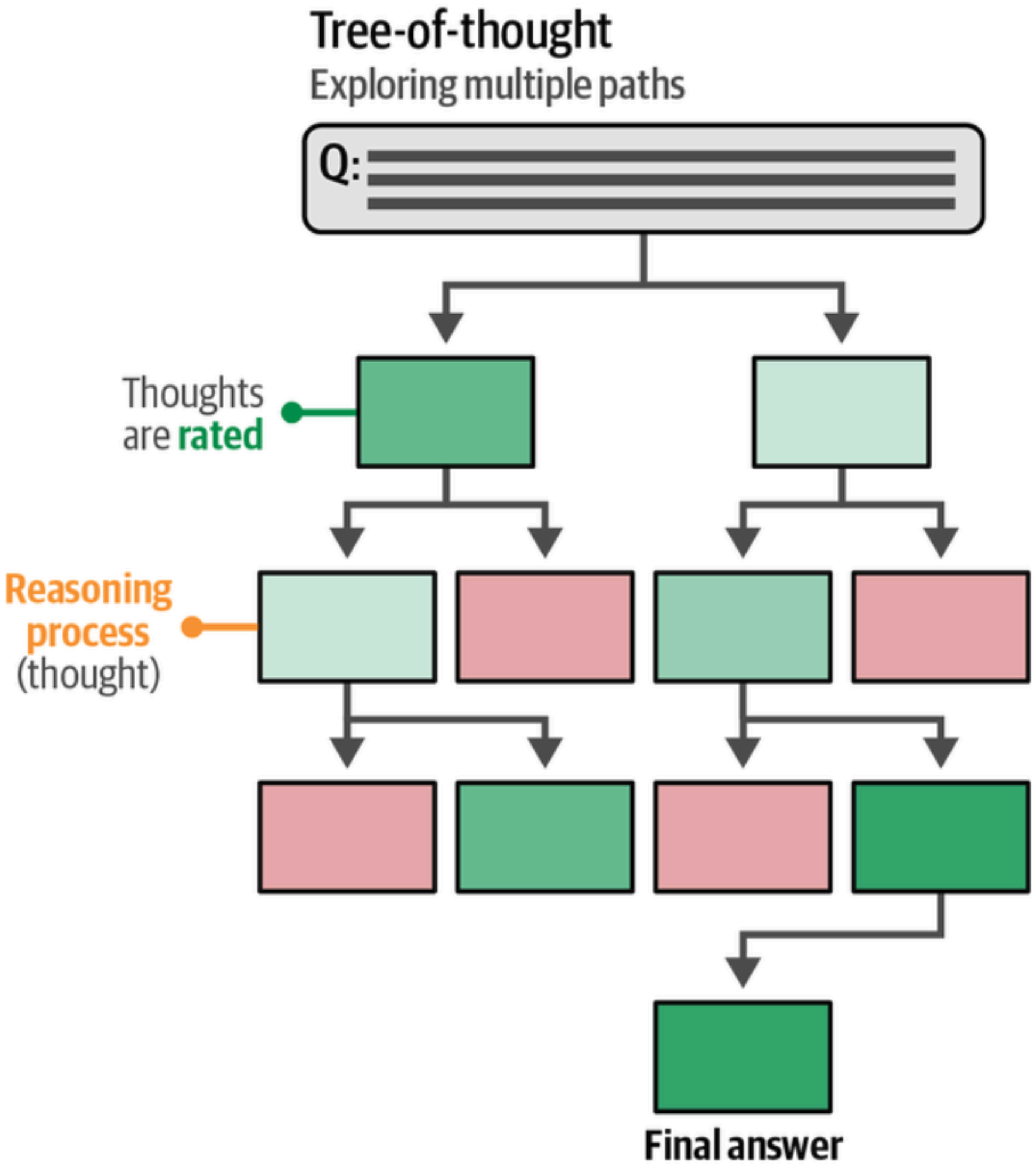


Figure 6-18. By leveraging a tree-based structure, generative models can generate intermediate thoughts to be rated. The most promising thoughts are kept and the lowest are pruned.

This method is helpful when needing to consider multiple paths. A disadvantage is the need for many calls to the model, which slows the process significantly.

Fortunately, there has been a successful attempt to convert the tree-of-thought framework into a single prompting technique.

Instead of calling multiple times, we prompt the model to mimic that behavior by creating a conversation between multiple experts.

```
# Zero-shot tree-of-thought
zeroshot_tot_prompt = [
    {"role":"user", "content":"Imagine three different experts are answering this question. All experts will write down 1 step of their thinking, then share it with the group. Then all experts will go to the next step, etc. If any expert realizes they're wrong at any point then they leave. The question is 'The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?' Make sure to discuss the results."}
]
```

Use this prompt:

```
outputs = pipe(zeroshot_tot_prompt)
print(outputs[0]["generated_text"])
```

This outputs:

```
Expert 1: Step 1 - Start with the initial number of apples: 23 apples.
Expert 2: Step 1 - Subtract the apples used for lunch: 23 - 20 = 3 apples remaining.
Expert 3: Step 1 - Add the newly bought apples: 3 + 6 = 9 apples.

Expert 1: Step 2 - Confirm the final count: The cafeteria has 9 apples.
Expert 2: Step 2 - Review the calculations: 23 - 20 = 3, then 3 + 6 = 9. The calculations are correct.
Expert 3: Step 2 - Agree with the result: The cafeteria indeedhas 9 apples.

All experts agree on the final count: The cafeteria has 9 apples.
```

Output Verification

Reasons for validating the output might include:

- **Structured output**
  Most generative models create free-form text without following a particular template. Some use cases require structured output in formats like JSON.
- **Valid output**
  Even with a correct format, it is important to validate the actual content of the output.

- **Ethics**
  Some models have no guardrails and generate inappropriate outputs. Examples are bias, cultural stereotypes, personally identifiable information (PII), etc.
- **Accuracy**
  The aim is to double-check whether the output is factual, coherent, and free from hallucination.

While parameters like `temperature` and `top_p` influence the output, there are generally three ways to control it:

- **Examples**
  As we have seen use in few-shot COT, providing a number of examples of the expected output helps improve the performance of the model.
- **Grammar**
  Control the token selection process.
- **Fine-tuning**
  Also discussed in this chapter, fine-tuning helps control what and how the the model generates.

Providing Examples

As we have seen, few-shot is helpful in telling the model how we want the output to be. The following is an example of this:

In this example, we want the model to create a character profile from an RPG game.

```
# Zero-shot learning: Providing no examples
zeroshot_prompt = [
    {"role":"user", "content":"Create a character profile for a RPG game in 
JSON format."}
]
```

Generate output:

```
outputs = pipe(zeroshot_prompt)
print(outputs[0]["generated_text"])
```

This outputs:

```
{
"characterProfile": {
"name": "Eldrin Stormbringer",
"class": "Warlock",
"race": "Half-Elf",
"age": 27,
"gender": "Male",
"alignment": "Chaotic Good",
"background": "Rogue",
…
},
"attributes": {
"strength": 10,
"dexterity": 17,
"constitution": 12,
"intelligence": 12,
"wisdom": 10,
"charisma
```

As you can see, the output is not completed, and is truncated after `charisma`. This is not valid and may cause issues in production.

Let's see how applying one-shot learning addresses this:

```
one_shot_template = """Create a short character profile for an RPG game.
Make sure to only use this format:
{
    "description": "A SHORT DESCRIPTION",
    "name": "THE CHARACTER'S NAME",
    "armor": "ONE PIECE OF ARMOR",
    "weapon": "ONE OR MORE WEAPONS"
}
"""
one_shot_prompt = [
    {"role": "user", "content": one_shot_template}
]
# Generate the output
outputs = pipe(one_shot_prompt)
print(outputs[0]["generated_text"])
```

This outputs:

```
{
    "description": "A cunning rogue with a mysterious past,
    skilled in stealth and deception.",
    "name": "Lysandra Shadowstep",
    "armor": "Leather Cloak of the Night",
    "weapon": "Dagger of Whispers, Throwing Knives"
}
```

The model perfectly followed the example we gave it, which allows for more consistent behavior.

It is important to realize that while this model generated an output in the desired format, this is not always the case. Some models are better at following patterns and instructions than others.

Grammar: Constrained Sampling

Few-shot learning has a big disadvantage: we cannot explicitly prevent certain output from being generated.

Packages have been developed to constrain and validate the output of generative models. They use the model to validate their own output. This process can be seen in the diagram below.

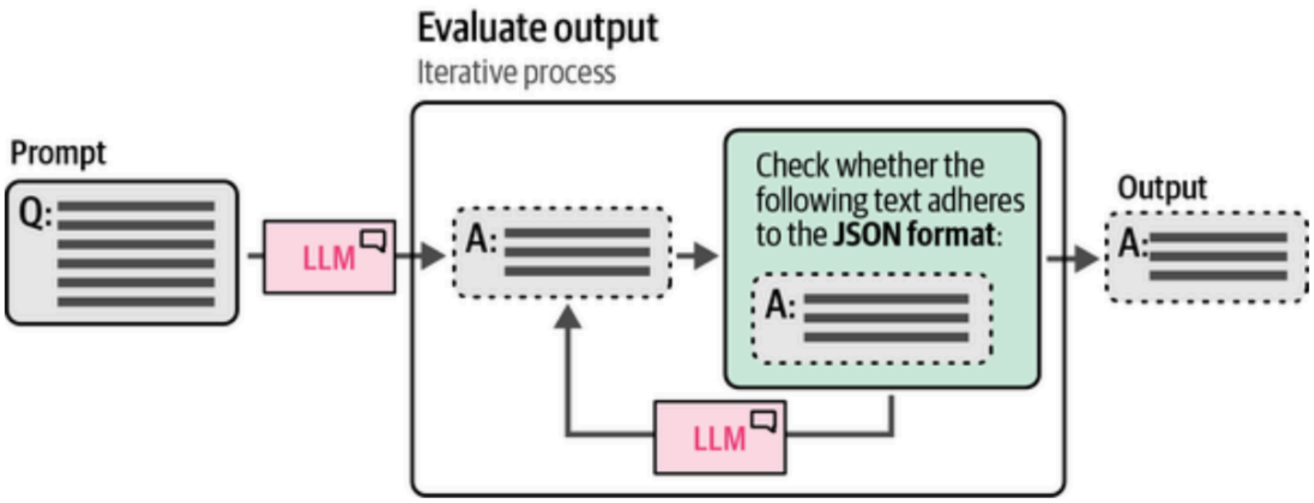

Evaluate output
Iterative process
Prompt
Q:
LLM
A:
Check whether the following text adheres to the JSON format:
A:
LLM
Output
A:

Figure 6-19. Use an LLM to check whether the output correctly follows our rules.

This process can be taken one step further and instead of validating the output we can already perform validation during the token sampling process.

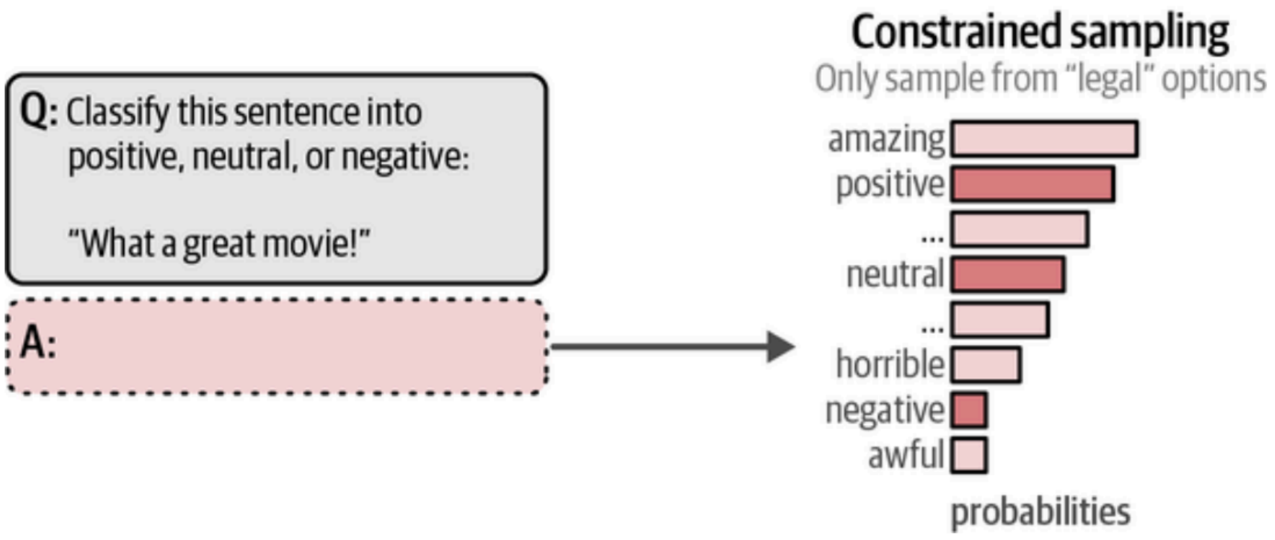


Figure 6-21. Constrain the token selection to only three possible tokens: "positive," "neutral," and "negative."

So instead of taking into account all possible next token, we have a set of words/tokens we consider 'legal'. Instead of picking any highest probability next word, it picks the highest probability legal word.

Let's illustrate this.

Since we are loading a new model, it is advised to restart the notebook:

```python
import gc
import torch
del model, tokenizer, pipe

# Flush memory
gc.collect()
torch.cuda.empty_cache()
```

```python
from llama_cpp.llama import Llama
# Load Phi-3
llm = Llama.from_pretrained(
    repo_id="microsoft/Phi-3-mini-4k-instruct-gguf",
    filename="*fp16.gguf",
    n_gpu_layers=-1,
    n_ctx=2048,
    verbose=False
)
```

To generate the output using the internal JSON grammar, we only need to specify the response_format as a JSON object.

```
# Generate output
output = llm.create_chat_completion(
messages=[
    {"role": "user", "content": "Create a warrior for an RPG in JSON
format."},
    ],
    response_format={"type": "json_object"},
    temperature=0,
)['choices'][0]['message']["content"]
```

To check whether the output actually is JSON:

```
import json

# Format as json
json_output = json.dumps(json.loads(output), indent=4)
print(json_output)
```

This outputs:

```
{
    "name": "Eldrin Stormbringer",
    "class": "Warrior",
    "level": 10,
    "attributes": {
        "strength": 18,
        "dexterity": 12,
        "constitution": 16,
        "intelligence": 9,
        "wisdom": 14,
        "charisma": 10
    },
    "skills": {
        "melee_combat": {
            "weapon_mastery": 20,
            "armor_class": 18,
            "hit_points": 35
    },
    "defense": {
        "shield_skill": 17,
        "block_chance": 90
    },
    "endurance": {
        "health_regeneration": 2,
        "stamina": 30
    }
    },
    "equipment": [
        {
            "name": "Ironclad Armor",
            "type": "Armor",
            "defense_bonus": 15
        },
        {
            "name": "Steel Greatsword",
            "type": "Weapon",
            "damage": 8,
            "critical_chance": 20
        }
    ],
    "background": "Eldrin grew up in a small village on the outskirts of a
war-torn land. Witnessing the brutality and suffering caused by conflict, he
dedicated his life to becoming a formidable warrior who could protect those
unable to defend themselves."
}
```

The output is properly formatted as JSON. As we can see, using this method lets us use the model more confidently where we want the output to follow very specific rules/templates.