Chapter 3. Standard Practices for Text Generation with ChatGPT

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

Simple prompting techniques will help you to maximise the output and formats from large language models (LLMs). You'll start by tailoring the prompts to explore all of the common practices used for text generation.

Generating Lists

Automatically generating lists is incredibly powerful and enables you to focus on higher level tasks while GPT can automatically generate, refine, re-rank and de-duplicate lists on your behalf.

Input:

```
Generate a list of Disney characters.
```

Output:

```
Sure, here is a list of some popular Disney characters:

1. Mickey Mouse
2. Minnie Mouse
...
30. Bagheera (The Jungle Book)
```

GPT-4 is perfectly capable of providing a list of characters. However there are some pitfalls with this approach:

- GPT has decided to provide 30 examples as a numbered list, separated by `\n` characters. However if your downstream Python code was expecting to split on bullet points, then you'll likely end up with undesirable results or a runtime error.
- GPT has provided preceding commentary, removing any preceding/succeeding commentary would make parsing the output easier.
- The list size wasn't controlled and was left to the language model.
- Some of the characters have the name of their corresponding film within brackets, *Bagheera (The Jungle Book)* and others don't. This makes names harder to extract, because you would need to remove the movie titles.
- No filtering or selection has been applied to the LLM generation, based on our desired result.

Below you'll find an optimized prompt:

Input:

```
Generate a bullet point list of 5 male Disney characters.
Only include the name of the character for each line.
Never include the film for each Disney character.
Only return the disney characters, never include any commentary.

Below is an example list:

* Aladdin
* Simba
* Beast
* Hercules
* Tarzan
```

Output:

```
* Woody
* Buzz Lightyear
* Stitch
* Jack Sparrow
* Prince Charming
```

PROVIDING EXAMPLES

Simply re-phrasing your prompt to include examples *(few-shot prompting)* can greatly impact the desired output.

By optimizing the prompt, you've achieved the following:

- Restricted the list to a fixed size of 5.
- Generated only male characters.
- Correctly formatted the list with bullet points.
- Removed any preceding commentary.

Simple lists are fine for most tasks, however they are less structured and for some tasks it's beneficial to obtain nested data structures from a GPT-4 output.

Three typical data structures include:

- Nested text data (Hierarchical list's)
- JSON
- YAML

Hierarchical List Generation

Hierarchical list's are useful for when your desired output is nested. A good example of this would be a detailed article structure.

Input:

```
Generate a hierarchical and incredibly detailed article outline on:

What are the benefits of data engineering.
```

```
See an example of the hierarchical structure below:

Article Title: What are the benefits of digital marketing?

* Introduction
    a. Explanation of digital marketing
    b. Importance of digital marketing in today's business world
* Increased Brand Awareness
    a. Definition of brand awareness
    b. How digital marketing helps in increasing brand awareness
```

Output:

```
Article Title: What are the benefits of data engineering?

* Introduction
    a. Explanation of data engineering
    b. Importance of data engineering in today's data-driven world

...(10 sections later)...

* Conclusion
    a. Importance of Data Engineering in the modern business world
    b. Future of Data Engineering and its impact on the data ecosystem
```

To generate an effective article outline in the preceding output, you've included two key phrases:

- *Hierarchical*: To suggest that the article outline needs to produce a nested structure.
- *Incredibly detailed*: To guide the language model towards producing a larger output. Other words that you could include which have the same effect would be *very long* or by specifying a large number of subheadings, *include at least 10 top level headings*.

NOTE

Asking a language model for a fixed number of items doesn't guarantee the language model will produce the same length. For example, if you ask for 10 headings, you might receive only 8. Therefore your code should either validate that 10 headings exist, or be flexible to handle varying lengths from the LLM.

So you've successfully produced a hierarchical article outline, but how could you parse the string into structured data?

Let's explore Example 3-1 using Python, where you've previously made a successful API call against OpenAI's GPT-4.

Two regular expressions are used to extract the headings and subheadings from `openai_result`. The `re` module in Python is used for working with regular expressions.

Example 3-1. Parsing a hierarchical list

```
import re

# openai_result = generate_article_outline(prompt)
# Commented out to focus on a fake LLM response, see below:

openai_result = '''
* Introduction
    a. Explanation of data engineering
    b. Importance of data engineering in today's data-driven world
* Efficient Data Management
    a. Definition of data management
    b. How data engineering helps in efficient data management.
* Conclusion
    a. Importance of Data Engineering in the modern business world
    b. Future of Data Engineering and its impact on the data ecosystem
'''

# Regular expression patterns
heading_pattern = r'\* (.+)'
subheading_pattern = r'\s+[a-z]\. (.+)'

# Extract headings and subheadings
headings = re.findall(heading_pattern, openai_result)
subheadings = re.findall(subheading_pattern, openai_result)

# Print results
print("Headings:\n")
for heading in headings:
    print(f"* {heading}")

print("\nSubheadings:\n")
for subheading in subheadings:
    print(f"* {subheading}")
```

This code will output:

```
Headings:
- Introduction
- Efficient Data Management
- Conclusion

Subheadings:
- Explanation of data engineering
- Importance of data engineering in today's data-driven world
- Definition of data management
- How data engineering helps in efficient data management.
- Importance of Data Engineering in the modern business world
- Future of Data Engineering and its impact on the data ecosystem
```

The use of regular expressions allows for efficient pattern matching, making it possible to handle variations in the input text, such as the presence or absence of leading spaces or tabs. Let's explore how these patterns work:

```
heading_pattern = r'\* (.+)'
```

This pattern is designed to extract the main headings and consists of:

- `\\*` matches the asterisk `(*)` symbol at the beginning of a heading. The backslash is used to escape the asterisk, as the asterisk has a special meaning in regular expressions (zero or more occurrences of the preceding character).
- *A space character will match after the asterisk.*
- `(.+)`: matches one or more characters, and the parentheses create a capturing group. The `.` is a wildcard that matches any character except a newline, and the `+` is a quantifier that means *one or more* occurrences of the preceding element (the dot, in this case).

By applying this pattern you can easily extract all of the main headings into a list without the asterisk.

```
subheading_pattern = r'\s+[a-z]\. (.+)'
```

The `subheading pattern` will match all of the subheadings within the `openai_result` string:

- `\s+` matches one or more whitespace characters (spaces, tabs, and so on). The `+` means *one or more* occurrences of the preceding element (the `\s`, in this case).
- `[a-z]` matches a single lowercase letter from a to z.
- `\.` matches a period character. The backslash is used to escape the period, as it has a special meaning in regular expressions (matches any character except a newline).
- *A space character will match after the period.*
- `(.+)` matches one or more characters, and the parentheses create a capturing group. The `.` is a wildcard that matches any character except a newline, and the `+` is a quantifier that means *one or more* occurrences of the preceding element (the dot, in this case).

Additionally the `re.findall()` function is used to find all non-overlapping matches of the patterns in the input string and return them as a list. The extracted headings and subheadings are then printed.

So now you're able to extract `headings` and `subheadings` from hierarchical article outlines, however you can further refine the regular expressions so that each `heading` is associated with corresponding `subheadings`.

In Example 3-2, the regex has been slightly modified so that each subheading is attached directly with it's appropriate subheading:

Example 3-2. Parsing a hierarchical list into a Python dictionary

```
import re

openai_result = """
* Introduction
  a. Explanation of data engineering
  b. Importance of data engineering in today’s data-driven world
* Efficient Data Management
    a. Definition of data management
    b. How data engineering helps in efficient data management.
    c. Why data engineering is important for data management
* Conclusion
    a. Importance of Data Engineering in the modern business world
    b. Future of Data Engineering and its impact on the data ecosystem
"""

section_regex = re.compile(r"\* (.+)")
subsection_regex = re.compile(r"\s*([a-z]\..+)")
```

```python
result_dict = {}
current_section = None

for line in openai_result.split("\n"):
    section_match = section_regex.match(line)
    subsection_match = subsection_regex.match(line)

    if section_match:
        current_section = section_match.group(1)
        result_dict[current_section] = []
    elif subsection_match and current_section is not None:
        result_dict[current_section].append(subsection_match.group(1))

print(result_dict)
```

This will output:

```
{
    "Introduction": [
        "a. Explanation of data engineering",
        "b. Importance of data engineering in today’s data-driven world"
    ],
    "Efficient Data Management": [
        "a. Definition of data management",
        "b. How data engineering helps in efficient data management."
    ],
    "Conclusion": [
        "a. Importance of Data Engineering in the modern business world",
        "b. Future of Data Engineering and its impact on the data ecosystem"
    ]
}
```

- The section title regex, `r'\* (.+)'`, matches an asterisk followed by a space and then one or more characters. The parentheses capture the text following the asterisk and space to be used later in the code.
- The subsection regex, `r'\s*([a-z]\..+)'`, starts with `\s*`, which matches zero or more whitespace characters (spaces or tabs). This allows the regex to match subsections with or without leading spaces or tabs. The following part, `([a-z]\..+)`, matches a lowercase letter followed by a period and then one or more characters. The parentheses capture the entire matched subsection text for later use in the code.

The `for loop` iterates over each line in the input string, `openai_result`. Upon encountering a line that matches the section title regex, the loop sets the matched title as the current section and assigns an empty list as its value in the `result_dict` dictionary. When a line matches the subsection regex, the matched subsection text is appended to the list corresponding to the current section.

Consequently, the loop processes the *input string line by line*, categorizes lines as section titles or subsections, and constructs the intended dictionary structure.

When To Avoid Using Regular Expressions

As you work to extract more structured data from LLM responses, relying solely on regular expressions can make the control flow *become increasingly complicated*. However, there are other formats that can facilitate the parsing of structured data from LLM responses with ease. Two common formats are *.json* and *.yml* files.

Generating JSON

Let's start by experimenting with initial some prompt design that will direct an LLM to return a JSON response.

Input:

```
Compose a very detailed article outline on "The benefits of learning code" wi
JSON payload structure that highlights key points.

Only return valid JSON.

Here is an example of the JSON structure:
{
    "Introduction": [
        "a. Explanation of data engineering",
        "b. Importance of data engineering in today's data-driven world"],
    ...
    "Conclusion": [
        "a. Importance of Data Engineering in the modern business world",
```

```
        "b. Future of Data Engineering and its impact on the data ecosystem"]
}
```

Output:

```
{
    "Introduction": [
        "a. Overview of coding and programming languages",
        "b. Importance of coding in today's technology-driven world"],
    ...
    "Conclusion": [
        "a. Recap of the benefits of learning code",
        "b. The ongoing importance of coding skills in the modern world"]
}
```

GIVING DIRECTION & PROVIDING EXAMPLES

Notice that in the preceding prompt, you've provided direction on the type of task, the format and an example JSON output.

Common errors that you'll encounter when working with JSON involve invalid payloads, or the JSON being wrapped within \``` triple backticks, such as:

Output:

````
Sure here's the JSON:
```json
{"Name": "John Smith"} # valid payload
{"Name": "John Smith", "some_key":} # invalid payload
```
````

Ideally you would like the model to respond like so:

Output:

```
{"Name": "John Smith"}
```

This is important because with the first output, you'd have to split after `json` and then parse the exact part of the string which contained valid

JSON.

There are several points that are worth adding to your prompts to improve JSON parsing:

```
You must follow the following principles:
* Only return valid JSON
* Never include backtick symbols such as: `
* The response will be parsed with json.loads(), therefore it must be valid J
```

Now let's examine how you can parse a JSON output with Python

```python
import json

# openai_json_result = generate_article_outline(prompt)

openai_json_result = """
{
    "Introduction": [
        "a. Overview of coding and programming languages",
        "b. Importance of coding in today's technology-driven world"],
    "Conclusion": [
        "a. Recap of the benefits of learning code",
        "b. The ongoing importance of coding skills in the modern world"]
}
"""
parsed_json_payload = json.loads(openai_json_result)
print(parsed_json_payload)

'''{'Introduction': ['a. Overview of coding and programming languages',
"b. Importance of coding in today's technology-driven world"],
'Conclusion': ['a. Recap of the benefits of learning code',
'b. The ongoing importance of coding skills in the modern world']}'''
```

Well done, you've successfully parsed some JSON.

As showcased, structuring data from an LLM response is streamlined when requesting the response in valid JSON format. Compared to the previously demonstrated regular expression parsing, this method is less cumbersome and more straightforward.

So what could go wrong?

- The language model accidentally adds extra text to the response such as `json output:` and your application logic only handles for valid JSON.
- The JSON produced isn't valid and fails upon parsing (either due to the size or simply for not escaping certain characters).

Later on you will examine strategies to gracefully handle for such edge cases.

YAML

.yml files are a structured data format that offer different benefits over *.json*:

- *No need to escape characters:* YAML's indentation pattern eliminates the need for braces, brackets, and commas to denote structure. This can lead to cleaner and less error-prone files, as there's less risk of mismatched or misplaced punctuation.
- *Readability:* YAML is designed to be human-readable, with a simpler syntax and structure compared to JSON. This makes it easier for you to create, read, and edit prompts, especially when dealing with complex or nested structures.
- *Comments*: Unlike JSON, YAML supports comments, allowing you to add annotations or explanations to the prompts directly in the file. This can be extremely helpful when working in a team or when revisiting the prompts after some time, as it allows for better understanding and collaboration.

Input:

```
- Below you'll find the current yaml schema.
- You can update the quantities based on a User Query
- Filter the User Query based on the schema below, if it doesn't match and
there are no items left then return `"No Items"`.
- If there is a partial match, then return only the items that are
within the schema below:

# schema:
- item: Apple Slices
  quantity: 5
  unit: pieces
- item: Milk
  quantity: 1
```

```
  unit: gallon
- item: Bread
  quantity: 2
  unit: loaves
- item: Eggs
  quantity: 1
  unit: dozen

User Query: "5 apple slices, and 2 dozen eggs."

Given the schema below, please return only a valid .yml based on the User
Query.If there's no match, return `"No Items"`. Do not provide any
commentary or explanations.
```

Output:

```
- item: Apple Slices
  quantity: 5
  unit: pieces
- item: Eggs
  quantity: 2
  unit: dozen
```

Notice with the preceding example how an LLM is able to infer the correct *.yml* format from the `User Query` string.

Additionally, you've given the LLM an opportunity to either:

- Return a valid *.ym* response.
- Return a filtered *.yml* response.
- If after filtering, and there are no *.yml* items left, then return *"No Items"*.

Filtering YAML Payloads

You might decide to use this same prompt for cleaning/filtering a *.yml* payload.

First, let's focus on a payload which contains both valid and invalid `schema` in reference to our desired `schema`. Apple slices fit the criteria, however Bananas doesn't exist and you should expect for the `User Query` to be appropriately filtered.

Input:

```
# User Query:
- item: Apple Slices
  quantity: 5
  unit: pieces
- item: Bananas
  quantity: 3
  unit: pieces
```

Output:

```
# Updated yaml list
- item: Apple Slices
  quantity: 5
  unit: pieces
```

In the preceding example, you've successfully filtered the user's payload against a set crtieria and have used the language model as a *reasoning engine*.

By providing the LLM with a set of instructions within the prompt, the response is more closely related to what a human might do if they were manually cleaning the data.

The input prompt facilitates the delegation of more control flow tasks to a language learning model (LLM), tasks which would typically require coding in a programming language like Python or JavaScript.

Figure 3-1 provides a detailed overview of the logic applied when processing user queries by an LLM.

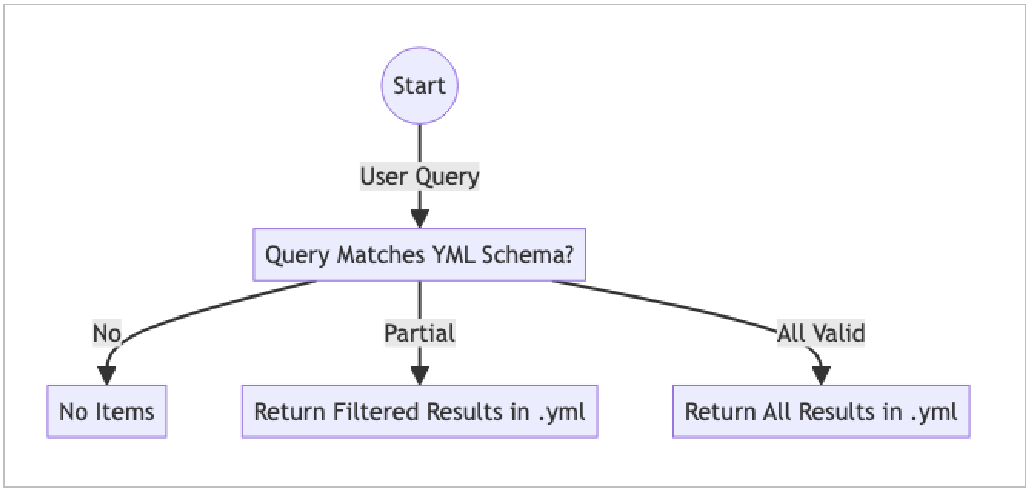

Figure 3-1. Using an LLM to determine the control flow of an application instead of code

Handling Invalid Payloads in YAML

A completely invalid payload might look like this:

Input:

```
# User Query:
- item: Bananas
  quantity: 3
  unit: pieces
```

Output:

```
No Items
```

As expected, the LLM returned `No Items` as none of the `User Query` items matched against the previously defined `schema`.

Let's create a Python script that gracefully accommodates for the various types of LLM results returned.

The core parts of the script will focus on:

1. Creating custom exceptions for each type of error that might occur due to the three LLM response scenarios.
2. Parsing the proposed schema.
3. Running a serious of custom checks against the response so you can be sure that the YML response can be safely passed to downstream

software applications/micro-services.

You could define 6 specific errors that would handle for all of the edge cases:

```python
class InvalidResponse(Exception):
    pass

class InvalidItemType(Exception):
    pass

class InvalidItemKeys(Exception):
    pass

class InvalidItemName(Exception):
    pass

class InvalidItemQuantity(Exception):
    pass

class InvalidItemUnit(Exception):
    pass
```

Then provide the previously proposed `YML schema` as a string:

```python
# Provided schema
schema = """
- item: Apple Slices
  quantity: 5
  unit: pieces
- item: Milk
  quantity: 1
  unit: gallon
- item: Bread
  quantity: 2
  unit: loaves
- item: Eggs
  quantity: 1
  unit: dozen
"""
```

Import the `yaml` module and create a custom parser function called `validate_response` allows you to easily determine whether an LLM output is valid:

```
import yaml

def validate_response(response, schema):
    # Parse the schema
    schema_parsed = yaml.safe_load(schema)
    maximum_quantity = 10

    # Check if the response is a list
    if not isinstance(response, list):
        raise InvalidResponse("Response is not a list")

    # Check if each item in the list is a dictionary
    for item in response:
        if not isinstance(item, dict):
            raise InvalidItemType('''Item is not a dictionary''')

        # Check if each dictionary has the keys "item", "quantity", and "unit"
        if not all(key in item for key in ("item", "quantity", "unit")):
            raise InvalidItemKeys("Item does not have the correct keys")

        # Check if the values associated with each key are the correct type
        if not isinstance(item["item"], str):
            raise InvalidItemName("Item name is not a string")
        if not isinstance(item["quantity"], int):
            raise InvalidItemQuantity("Item quantity is not an integer")
        if not isinstance(item["unit"], str):
            raise InvalidItemUnit("Item unit is not a string")

        # Check if the values associated with each key are the correct value
        if item["item"] not in [x["item"] for x in schema_parsed]:
            raise InvalidItemName("Item name is not in schema")
        if item["quantity"] >  maximum_quantity:
            raise InvalidItemQuantity(f'''Item quantity is greater than
            {maximum_quantity}''')
        if item["unit"] not in ["pieces", "dozen"]:
            raise InvalidItemUnit("Item unit is not pieces or dozen")
```

To test these edge cases, below you'll find several mocked LLM responses:

```
# Fake responses
fake_response_1 = """
- item: Apple Slices
  quantity: 5
  unit: pieces
- item: Eggs
  quantity: 2
```

```
    unit: dozen
"""

fake_response_2 = """
# Updated yaml list
- item: Apple Slices
  quantity: 5
  unit: pieces
"""

fake_response_3 = """Unmatched"""
```

Finally now you can:

- Use `yaml.safe_load(response)` to safely parse the *.yml* schema.
- Call the `validate_response` function for each LLM response to test it against custom *.yml* validation logic.

```
# Parse the fake responses
response_1_parsed = yaml.safe_load(fake_response_1)
response_2_parsed = yaml.safe_load(fake_response_2)
response_3_parsed = yaml.safe_load(fake_response_3)

# Validate the responses against the schema
try:
    validate_response(response_1_parsed, schema)
    print("Response 1 is valid")
except Exception as e:
    print("Response 1 is invalid:", str(e))

try:
    validate_response(response_2_parsed, schema)
    print("Response 2 is valid")
except Exception as e:
    print("Response 2 is invalid:", str(e))

try:
    validate_response(response_3_parsed, schema)
    print("Response 3 is valid")
except Exception as e:
    print("Response 3 is invalid:", str(e))

Response 1 is valid
Response 2 is valid
Response 3 is invalid: Response is not a list
```

Diverse Format Generation with ChatGPT

ChatGPT is not only capable of generating *.json* or *.yml* files but can also support many other data formats. For example, it can aid you in creating a mermaid diagram or generating a mock *.csv* document. Its versatility stretches to even generating code, conversations, and scripts.

Let's explore in detail how ChatGPT can produce diverse formatted content. Mermaid is a Markdown-like scripting language for generating charts, diagrams, and other visualizations.

Input:

```
Create a flowchart diagram in mermaid syntax showing a simple food ordering
process that starts with "Choose food" and ends with "Pay for meal."
Include steps for "Add food to cart" and "Confirm cart."
```

Output:

```
graph TD
    ChooseFood[Choose Food] --> AddToCart[Add to Cart]
    AddToCart --> ConfirmCart[Confirm Cart]
    ConfirmCart --> PayForMeal[Pay for Meal]
```

This output, when input to a mermaid diagram renderer, will create a precise flowchart reflecting the food ordering process (Figure 3-2).

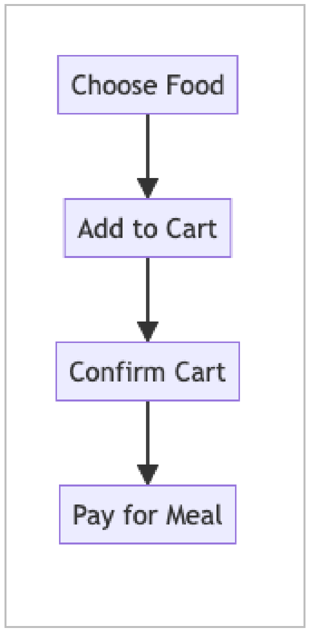

Figure 3-2. A streamlined flow diagram created using mermaid syntax via an LLM

Mock CSV Data

ChatGPT can also generate mock *.csv* data which can be utilized for testing purposes or other uses.

Input:

```
Generate a sample CSV data for five students with the following fields:
name, age, grade.
```

Output:

```
name,age,grade
John,16,A
Alex,17,B
Emma,16,A
Sophia,16,B
Liam,17,A
```

The example generated is a well-formatted CSV file, which can be loaded directly into many tools that read CSV data.

Example 3-3. Specify Format

Remember, GPT models rely heavily on the format you specify. For instance, specifying CSV implies the AI should use commas as separators. Be clear about the desired format for optimal results.

You've explored different ways of extracting structured data from language models including regular expressions, JSON, YML and other formats.

Also, you can now use LLMs to direct parts of your application's control flow.

Explain It Like I'm Five

The *Explain It Like I'm Five* prompt focuses on explaining a given section of text as *if you were talking to a five year old child.*

This style of prompting is very effective and is a great way to convert technical documents into simple summaries that anyone can understand.

Let's apply this prompt to an abstract of [Tumor microenvironment: recent advances in various cancer treatments.]()

Input:

```
Explain the following text as if I was five years old. Explain it like I'm fi

Text: This is a review regarding different types of cancer treatments.
We aimed at analyzing the tumor microenvironment and the recent trends
for the therapeutic applications and effectiveness for several kinds
of cancers. Traditionally the cancer treatment was based on the
neoplastic cells.

...(shortened for brevity)...
```

Output:

```
Cancer is a sickness that makes your body's cells grow and divide in a way
that is not normal. Doctors have different ways to treat cancer, like
cutting it out, giving you medicine, or using special rays to kill the
cancer cells...
```

The ChatGPT output from the preceding example is much more digestible, than the abstract.

Universal Translation Through LLMs

Additionally, language models can potentially act as *universal translators* due to their proficiency in understanding and generating human-like text across multiple languages.

Let's investigate this with a simple example:

1. Generate some simple text.
2. Change the simple text and make it difficult to read.
3. Then transform the same text back into an easy to read format in Spanish (instead of English).
4. Revert the simplistic text back into English.

Input:

```
Create some simple text
```

Output:

```
The sun gently rises, painting the sky with hues of orange and pink. Birds
chirp merrily as they welcome the new day. The cool breeze whispers through
the trees, carrying the scent of fresh grass and blooming flowers. Life
awakens with a sense of tranquility and promise.

...(shortened for brevity)...
```

Input:

```
Now make the text incredibly complex and hard to read
```

Output:

```
In the ethereal realm of celestial ascendancy, wherein the resplendent
orb of the sun commences its gradual ascent, an exquisite phenomenon
unfolds,engendering a symphony of chromatic brilliance upon the
expansive canopy of the firmament. Avian denizens, endowed with
mellifluous voices, bespeak their jubilant salutations, punctuating
the aural tapestry with notes of felicity. The zephyr, tenderly
wafting amidst arboreal sentinels, carries upon its gentle currents
```

```
the redolence of verdant blades and blossoming flora.
...(shortened for brevity)...
```

Input:

```
Now transform the text into a simple, easy to read format that's in Spanish
, make sure that it is very easy to understand and read. You must ensure
that it is a simple to comprehend as possible.

- Use simple language
- Make it concise
```

Output:

```
En un lugar mágico y celestial, cuando el sol empieza a salir, ocurre algo
hermoso en el cielo.Los pájaros cantan alegremente, llenando el aire con
sus dulces melodías. Una brisa suave sopla entre los árboles, llevando
consigo el olor fresco de la naturaleza. Todo cobra vida y promete momentos
felices. Nos invita a disfrutar de las cosas simples y a encontrar la
felicidad en los pequeños detalles. En este escenario encantador,
encontramos paz y aprendemos que hasta los momentos más simples pueden
ser mágicos.
```

Input:

```
Finally translate it back into English.
```

Output:

```
In a magical and celestial place, when the sun begins to rise, something
beautiful happens in the sky. The birds sing cheerfully, filling the air
with their sweet melodies. A gentle breeze blows among the trees, carrying
the fresh scent of nature. Everything comes alive and promises happy
moments. It invites us to enjoy the simple things and find happiness in the
little details. In this enchanting setting, we find peace and learn that
even the simplest moments can be magical.
```

As you can see part of the meaning is lost while doing the translation, and this is to be expected. However large parts of the original text's intent remains, demonstrating that LLMs can act as *highly effective translators* for languages that have adequate resources on the internet. For languages that do not have good amount of resources, it will yield bad results.

The same logic applies to coding languages, LLMs are very good at generating code for established programming languages such as Python and JavaScript but perform worse for newer coding languages and packages.

The boundaries between different forms of information are becoming *increasingly fluid*. The essence of information itself is evolving, allowing for effortless transformations of summaries into stories, poems, or other creative expressions, ultimately enriching our understanding and engagement with the content.

Diffusion models are a unique class of generative models utilized in machine learning, specifically designed to produce new images that mimic those found in the training set.

Moreover, when you combine language models with diffusion models, it enables seamless transitions between text, video, and other modalities. This makes it even simpler for you to convey complex ideas across various formats, facilitating a more accessible and comprehensive experience.

Ask For Context

LLMs are not only capable of generating text but can also act as simple agents with a limited amount of *reasoning capability.* This allows you to write a prompt asking the language model to either:

- Return a valid result to a question or statement.
- Ask for more context to appropriately answer the question.

In this section, you'll learn about the importance of *asking for context* when working with LLMs such as GPT-4. We will start with an example of a prompt that doesn't provide enough context, resulting in a less useful response.

Then, we will provide a better prompt that encourages the model to ask for additional context if needed. Finally, we will use the additional context provided to generate a more informed response.

Input:

```
Should I use MongoDB or PostgreSQL for my project?
```

Output:

```
It's difficult to make a recommendation without knowing more about your
specific software requirements. Both MongoDB and PostgreSQL have their
advantages, but the best choice depends on the specific needs of your
project.
```

In the prompt above, the model does not have enough context to make a meaningful recommendation. Instead, you can ask ChatGPT for a list of recommended points that would help it to make an effective decision.

Input:

```
Should I use MongoDB or PostgreSQL for my project?

Provide a list of requirements that would help you to make an informed
and high quality decision about which database solution I should choose?
```

Output:

```
To make an informed decision between MongoDB and PostgreSQL for your
project, consider the following requirements:

1. Data Structure: Evaluate the structure of your data. If your data
is highly unstructured or schema-less, MongoDB's flexible document model
may be more suitable. On the other hand, if your data has a fixed schema
or requires complex relationships, PostgreSQL's relational model might
be a better fit.

...

10. Cost: Assess the cost implications of each database solution. MongoDB
offers both a community edition (free) and a commercial edition with
additional features and support.PostgreSQL is open-source and free to use,
with optional commercial support available from various vendors.
```

After prompting ChatGPT about how to make an informed decision, now you're aware of what to include within your prompt to help the language model with deciding.

Input:

```
Should I use MongoDB or PostgreSQL for my project?

Context:
- Data Structure: Highly unstructured data with varying document formats
and no fixed schema. Scalability: Project requires the ability to handle
massive amounts of data and distribute it across multiple servers.
- ACID Compliance: Strong data consistency and integrity are crucial for
the project's financial transactions.
- Querying Capabilities: Complex geospatial queries and aggregations on
location-based data.
Community and Ecosystem: Active community with numerous third-party
libraries and tools specifically tailored for the project's industry.
- Data Integrity and Constraints: Enforcing strict foreign key
relationships and complex data validation rules.
- Transactions and Concurrency: Requires handling high levels of
concurrent read and write operations while maintaining data consistency.
- Maturity and Stability: Prioritizing a well-established and extensively
tested database for long-term stability.
- Development and Operational Ease: Agile development approach with a need
for rapid prototyping and iterative schema design.
- Cost: Budget constraints necessitate using an open-source solution with
no additional licensing or support costs.

If you need more context, please specify what would help you to make
a better decision.
```

Output:

```
Based on these considerations, both MongoDB and PostgreSQL have their
strengths, but given the emphasis on strong data consistency, complex
querying, and data integrity constraints, PostgreSQL seems to be a more
suitable choice for your project.
```

In this final example, the model uses the additional context provided to give a well-informed recommendation for using PostgresSQL. By asking for context when necessary, LLMs like ChatGPT and GPT-4 can deliver more valuable and accurate responses.

Figure 3-3 demonstrates how *asking for context* changes the decision-making process of LLMs. Upon receiving user input, the model first assesses whether the context given is sufficient. If not, it prompts the user to provide more detailed information, emphasizing the model's reliance on context-rich inputs. Once adequate context is acquired, the LLM then generates an informed and relevant response.

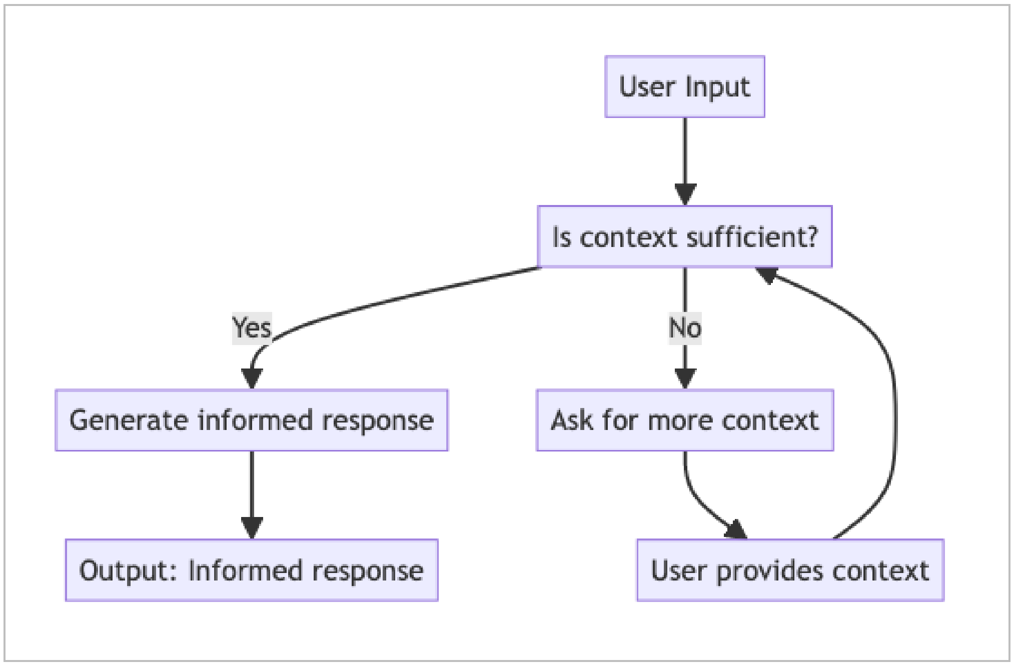

Figure 3-3. The decision process of an LLM while using asking for context

ALLOW THE LLM TO ASK FOR MORE CONTEXT BY DEFAULT

You can allow the LLM to ask for more context as a default by including the key phrase: *If you need more context, please specify what would help you to make a better decision.*

In this section, you've seen how LLMs can act as agents which use environmental context to make decisions. By iteratively refining the prompt based on the model's recommendations, we eventually reach a point where the model has *enough context to make a well informed decision.*

This process highlights the importance of providing sufficient context in your prompts and being prepared to ask for more information when necessary. By doing so, you can leverage the power of LLMs like GPT-4 to make more accurate and valuable recommendations.

In agent-based systems like GPT-4, the ability to ask for more context and provide a finalized answer is crucial for making well-informed decisions. AutoGPT, a multi-agent system has a self-evaluation step that automatically checks whether the task can be completed given the current context within the prompt. This technique uses an actor-critic relationship, where the existing prompt context is being analysed to see whether it could be further refined before being executed.

Text Style Unbundling

Text Style Unbundling is a powerful technique in prompt engineering that allows you to extract and isolate specific textual features from a given document, such as tone, length, vocabulary, and structure.

This allows you to create new content that shares similar characteristics with the original document, ensuring consistency in style and tone across various forms of communication.

This consistency can be crucial for businesses and organizations that need to communicate with a unified voice across different channels and platforms. The benefits of this technique include:

1. *Improved brand consistency*: By ensuring that all content follows a similar style, organizations can strengthen their brand identity and maintain a cohesive image.
2. *Streamlined content creation*: By providing a clear set of guidelines, writers and content creators can more easily produce materials that align with a desired style.
3. *Adaptability*: Text style unbundling allows for the easy adaptation of existing content to new formats or styles while preserving the core message and tone.

The process of text style unbundling involves *identifying the desired textual features* or creating a meta prompt (a prompt to create prompts) to extract these features, and then using the extracted features to guide the generation of new content.

Identifying the Desired Textual

Features

To successfully unbundle a text style, you must first identify the specific features you wish to extract from the input document. Common textual features to consider include:

- *Tone of voice*: The overall mood or attitude conveyed by the text, such as formal, casual, humorous, or authoritative.
- *Length*: The desired word count or general length of the content.
- *Vocabulary and phrasing*: The choice of words and expressions, including industry-specific jargon, colloquialisms, and complexity.

Additionally, you can even create a meta prompt that will dynamically show you these.

Input:

```
Analyze the following text and identify core features which will allow
further documents to imitate the same style, consistency and formatting
as the document below.

Provide all of the features as a style writing guide that will be used
for future documents.

Analyze the following text:

Artificial intelligence is rapidly transforming the world around us,
opening up new possibilities and revolutionizing industries. From
healthcare to entertainment, AI is making our lives better and more
convenient in countless ways. However, with great power comes great
responsibility. We must ensure that AI technologies are developed and
deployed ethically and responsibly to protect our privacy and uphold our
values.
```

Output:

```
Style Writing Guide:

Tone of Voice
- Maintain an informative and optimistic tone throughout the document.
- Emphasize the positive impact of the subject matter while acknowledging
any - potential concerns or responsibilities.
```

```
Length:
...

Vocabulary:
...

Structure:
...

Content:
- Focus on the transformative nature and positive impacts of the subject
matter.
- Highlight specific industries or areas where the subject matter has
made a difference.
- Acknowledge any associated responsibilities or potential risks,
emphasizing the need for ethical and responsible practices.
Stress the importance of privacy protection and upholding values in
relation to the subject matter.
```

Generating New Content with the Extracted Features

After you've extracted the desired features from the input document, you can use this information to guide future content generation. Remember to ensure that the AI model incorporates the extracted features into its output by specifying the desired style in your prompt.

For example, `Write a new blog post on [topic] using the same tone of voice, length, vocabulary, and structure as the previously analyzed text.`

By combining this technique with *reference text* (documents that act as grounding truth), you can produce credible, branded content that requires minimal revisions.

Extracting Specific Textual Features with LLMs

You can easily tailor a prompt to guide an LLM in extracting particular textual features from a document. This can be applied beyond just analyzing text for copywriting purposes. For instance, recognizing entities or discerning sentiment from the text can be achieved by crafting a precise instruction for the LLM.

Input:

```
Analyze the following text to identify and list the entities mentioned:

Artificial intelligence is rapidly transforming the world around us,
opening up new possibilities and revolutionizing industries. From
healthcare to entertainment, AI is making our lives better and more
convenient in countless ways. However, with great power comes great
responsibility. We must ensure that AI technologies are developed
and deployed ethically and responsibly to protect our privacy and
uphold our values.
```

Output:

```
Entities:
- Artificial Intelligence (AI)
- Healthcare
- Entertainment
- Privacy
- Values
```

Summarization

In the era of information overload, the ability to condense large amounts of text into concise, digestible summaries has become an essential skill. As the volume of data we produce and consume continues to grow, the demand for effective summarization techniques has increased significantly.

AI has made significant advancements in this domain, providing powerful tools for generating summaries that retain the most critical information while discarding less relevant details.

Let’s explore how to summarise using a language model.

Input:

```
Summarize the following text:

Whereas the party of the first part, hereinafter referred to as the
"Purchaser," and the party of the second part, hereinafter referred to
as the "Vendor," being fully cognizant of their respective legal
obligations and responsibilities pursuant to the applicable statutory
provisions and common law doctrines, have duly entered into a legally
binding agreement, hereinafterreferred to as the "Agreement," for the
purpose of effectuating a transaction involving the transfer of goods,
hereinafter referred to as the "Goods," between the Purchaser and the
Vendor, subject to the terms and conditions set forth herein, the Parties
do hereby solemnly acknowledge and affirm the following:
...
```

Output:

```
This is a legally binding agreement between the Purchaser and the Vendor
for the transfer of goods. The agreement incorporates the recitals as an
integral part, but in case of any conflict, the operative provisions of the
agreement prevail. The interpretation clause states that singular words
include the plural and vice versa, and words referring to a gender include
all genders. References to statutory provisions include any amendments or
replacements.
```

Summarisation is an invaluable application of AI, enabling users to quickly extract key insights from lengthy articles, reports, or research papers. This process can help individuals make informed decisions, save time, and prioritize their reading. AI-generated summaries can also facilitate information sharing among teams, allowing for more efficient collaboration and communication.

Summarizing Given Context Window Limitations

For documents larger than an LLM can handle in a single API request, a common approach is to chunk the document, summarize each chunk and then combine these summaries into a final summary as seen in Figure 3-4:

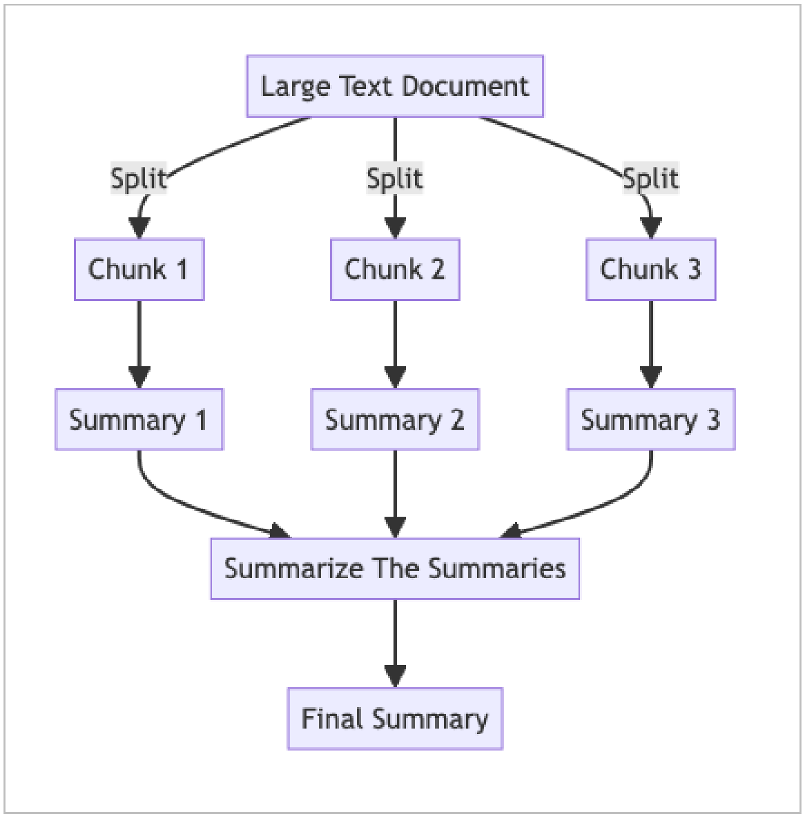

Figure 3-4. A summarization pipeline that uses text splitting and multiple summarization steps

Additionally, people may require different types of summaries for various reasons, and this is where AI summarization comes in handy. As illustrated in the preceding diagram, a large PDF document could easily be processed using AI summarization to generate distinct summaries tailored to individual needs:

- Summary A: Provides key insights is perfect for users seeking a quick understanding of the document's content, enabling them to focus on the most crucial points.
- Summary B: On the other hand, offers decision-making information, allowing users to make informed decisions based on the content's implications and recommendations.
- Summary C: Caters to collaboration and communication, ensuring that users can efficiently share the document's information and work together seamlessly.

By customizing the summaries for different users, AI summarization contributes to increased information retrieval for all users, making the entire process more efficient and targeted.

Let's assume you're only interested in finding and summarising information about the advantages of digital marketing. Simply change your summarization prompt to `Provide a concise, abstractive summary of the above text. Only summarise the advantages: ...`

AI-powered summarization has emerged as an essential tool for quickly distilling vast amounts of information into concise, digestible summaries that cater to various user needs. By leveraging advanced language models like GPT-4, AI summarization techniques can efficiently extract key insights, decision-making information, and facilitate collaboration and communication.

As the volume of data continues to grow, the demand for effective and targeted summarization will only increase, making AI a crucial asset for individuals and organizations alike in navigating the information age.

Chunking Text

LLMs continue to develop and play an increasingly crucial role in various applications, the ability to process and manage large volumes of text becomes ever more important. An essential technique for handling large-scale text is known as *chunking*.

Chunking refers to the process of breaking down large pieces of text into smaller, more manageable units or chunks. These chunks can be based on various criteria, such as sentence, paragraph, topic, complexity, or length. By dividing text into smaller segments, AI models can more efficiently process, analyze, and generate responses.

Figure 3-5 illustrates the process of chunking a large piece of text and subsequently extracting topics from the individual chunks.

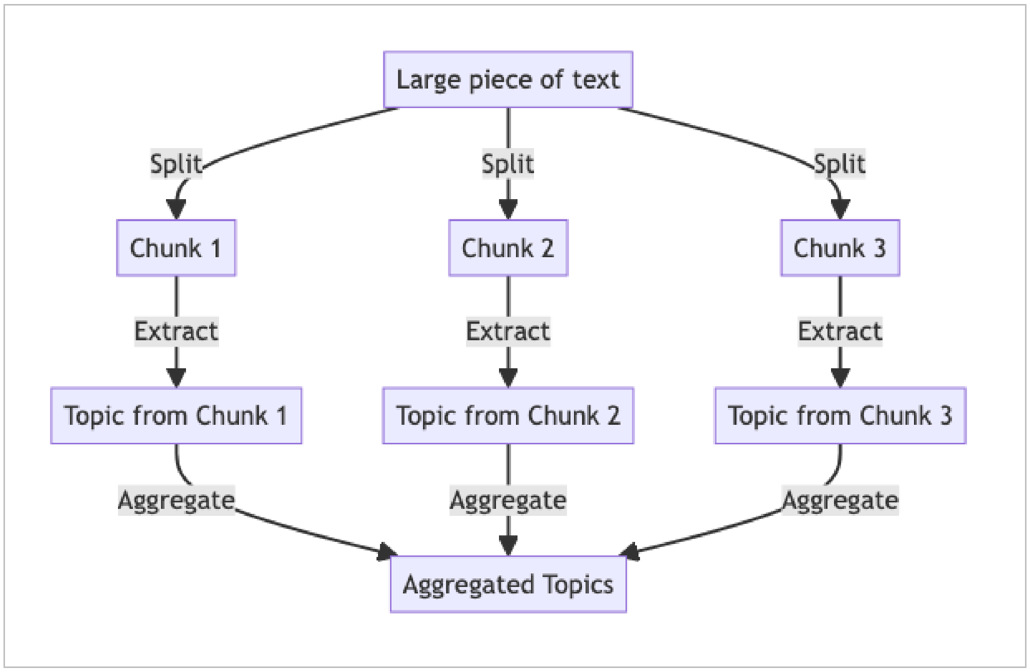

Figure 3-5. Topic extraction with an LLM after chunking text

Benefits of chunking text

There are several advantages to chunking text which include:

- *Fitting within a given context length*: LLMs only have a certain amount of input and output tokens which is called a *context length*. By reducing the input tokens you can make sure the output won't be cut off and the initial request won't be rejected.
- *Reducing Cost*: Chunking helps you to only retrieve the most important points from documents, which reduces your token usage and API costs.
- *Improved performance*: Chunking reduces the processing load on LLMs, allowing for faster response times and more efficient resource utilization.
- *Increased flexibility*: Chunking allows developers to tailor AI responses based on the specific needs of a given task or application.

Scenarios for Chunking Text

Chunking text can be particularly beneficial in certain scenarios, while in others it may not be required. Understanding when to apply this technique can help in optimizing the performance and cost-efficiency of LLMs.

When to Chunk:

- Large Documents: When dealing with extensive documents that exceed the maximum token limit of the LLM.
- Complex Analysis: In scenarios where a detailed analysis is required and the document needs to be broken down for better comprehension and processing.
- Multi-topic Documents: When a document covers multiple topics and it's beneficial to handle them individually.

When Not to Chunk:

- Short Documents: When the document is short and well within the token limits of the LLM.
- Simple Analysis: In cases where the analysis or processing required is straightforward and doesn't benefit from chunking.
- Single-topic Documents: When a document is focused on a single topic and chunking doesn't add value to the processing.

Poor Chunking Example

When text is not chunked correctly it can lead to reduced LLM performance. Consider the following paragraph from a news article:

```
The local council has decided to increase the budget for education by 10%
this year, a move that has been welcomed by parents and teachers alike. The
additional funds will be used to improve school infrastructure, hire more
teachers, and provide better resources for students. However, some critics
argue that the increase is not enough to address the growing demands of the
education system.
```

When the text is fragmented into isolated words, the resulting list lacks the original context:

`["The", "local", "council", "has", "decided", "to", "increase", "the", "budget", ...]`

The main issues with this poor chunking example include:

- *Loss of context*: By splitting the text into individual words, the original meaning and relationships between the words are lost. This makes it difficult for AI models to understand and respond effectively.

- *Increased processing load*: Processing individual words requires more computational resources, making it less efficient than processing larger chunks of text.

As a result of the poor chunking in this example, an LLM may face several challenges:

- Difficulty understanding the main ideas or themes of the text.
- Struggling to generate accurate summaries or translations.
- Inability to effectively perform tasks such as sentiment analysis or text classification.

By understanding the pitfalls of poor chunking, yous can apply prompt engineering principles to improve the process and achieve better results with AI language models.

Let's explore an improved chunking example using the same news article paragraph from the previous section, you'll now chunk the text by sentence:

```
["The local council has decided to increase the budget for education
by 10% this year, a move that has been welcomed by parents and teachers alike

"The additional funds will be used to improve school infrastructure,
hire more teachers, and provide better resources for students.",

"However, some critics argue that the increase is not enough to
address the growing demands of the education system."]
```

DIVIDING LABOR & EVALUATING QUALITY

Define the granularity at which the text should be chunked, such as by sentence, paragraph, or topic. Adjust parameters like the number of tokens or model temperature to optimize the chunking process.

By chunking the text in this manner, you could insert whole sentences into an LLM prompt with the most relevant sentences.

Chunking Strategies

There are many different chunking strategies including:

- *Splitting by sentence*: Preserves the context and structure of the original content, making it easier for LLMs to understand and process the information. Sentence-based chunking is particularly useful for tasks like summarization, translation, and sentiment analysis.
- *Splitting by paragraph*: This approach is especially effective when dealing with longer content, as it allows the LLM to focus on one cohesive unit at a time. Paragraph-based chunking is ideal for applications like document analysis, topic modeling, and information extraction.
- *Splitting by topic or section*: This method can help AI models better identify and understand the main themes and ideas within the content. Topic-based chunking is well-suited for tasks like text classification, content recommendations, and clustering.
- *Splitting by complexity*: For certain applications, it might be helpful to split text based on its complexity, such as the reading level or technicality of the content. By grouping similar complexity levels together, LLMs can more effectively process and analyze the text. This approach is useful for tasks like readability analysis, content adaptation, and personalized learning.
- *Splitting by length*: This technique is particularly helpful when working with very long or complex documents, as it allows LLMs to process the content more efficiently. Length-based chunking is suitable for applications like large-scale text analysis, search engine indexing, and text preprocessing.
- *Splitting by tokens using a Tokenizer*: Utilizing a tokenizer is a crucial step in many natural language processing tasks, as it enables the process of splitting text into individual tokens. Tokenizers divide text into smaller units, such as words, phrases, or symbols, which can then be analyzed and processed by AI models more effectively. You'll shortly be using a package called `tiktoken` which is a bytes-pair encoding tokenizer (BPE) for chunking.

Table 3-1 provides a high level overview of the different chunking strategies, it's worth considering what matters to you most when performing chunking.

Are you more interested in preserving semantic context or would naively splitting by length suffice?

Table 3-1. Six chunking strategies highlighting their advantages and disadvantages.

Splitting Strategy	Advantages	Disadvantages
Splitting by sentence	Preserves context, suitable for various tasks	May not be efficient for very long content
Splitting by paragraph	Handles longer content, focuses on cohesive units	Less granularity, may miss subtle connections
Splitting by topic	Identifies main themes, better for classification	Requires topic identification, may miss fine details
Splitting by complexity	Groups similar complexity levels, adaptive	Requires complexity measurement, not suitable for all tasks
Splitting by length	Manages very long content, efficient processing	Loss of context, may require more preprocessing steps
Using a tokenizer: Splitting by tokens	Accurate token counts, which helps in avoiding LLM prompt token limits	Requires tokenization, may increase computational complexity

By choosing the appropriate chunking strategy for your specific use case, you can optimize the performance and accuracy of AI language models.

Sentence Detection using SpaCy

Sentence detection, also known as sentence boundary disambiguation is the process used in NLP that involves identifying the start and end of sentences within a given text.

It can be particularly useful for tasks that require preserving the context and structure of the original content. By splitting the text into sentences, LLMs can better understand and process the information for tasks such as summarization, translation, and sentiment analysis.

Splitting by sentence is possible using NLP libraries such as SpaCy.

Ensure that you have SpaCy installed in your Python environment. You can install it with `pip install spacy`.

Download the `en_core_web_sm model` using the command `python -m spacy download en_core_web_sm`

In Example 3-4, the code demonstrates sentence detection using the SpaCy library in Python:

Example 3-4. Sentence detection with SpaCy

```
import spacy

nlp = spacy.load("en_core_web_sm")

text = "This is a sentence. This is another sentence."

doc = nlp(text)

for sent in doc.sents:
    print(sent.text)
```

Output:

```
This is a sentence.
This is another sentence.
```

First, you'll import the `spacy` library and load the English model `(en_core_web_sm)` to initialize an `nlp object`. Define an input text with two sentences, then the text is processed with `doc = nlp(text)`, creating a `doc object` as a result. Finally, the code iterates through the detected sentences using the `doc.sents` attribute and prints each sentence.

Building a simple chunking algorithm in Python

After exploring many chunking strategies, it's important to build your intuition by writing a simple chunking algorithm from scatch.

Example 3-5 shows how to chunk text based on the length of characters from the blog post "Hubspot - What is digital marketing?". This file can be found in the Github repository at *content/chapter_3/hubspot_blog_post.txt*.

To correctly read the `hubspot_blog_post.txt` file, make sure that your current working directory is set to the content/chapter_3 Github directory. This applies for both running the Python code or launching the Jupyter Notebook server.

Example 3-5. Character chunking

```
with open("hubspot_blog_post.txt", "r") as f:
    text = f.read()

chunks = [text[i : i + 200] for i in range(0, len(text), 200)]

for chunk in chunks:
    print("-" * 20)
    print(chunk)
```

Output:

```
search engine optimization strategy for many local businesses is an optimized
Google My Business profile to appear in local search results when people look
products or services related to what yo
--------------------
u offer.

For Keeps Bookstore, a local bookstore in Atlanta, GA, has optimized its
Google My Business profile for local SEO so it appears in queries for
“atlanta bookstore.”
--------------------
...(shortened for brevity)...
```

First, you open the text file `hubspot_blog_post.txt` with the `open` function, and read its contents into the variable text. Then using a list comprehension you create a list of chunks, where each `chunk` is a 200-character substring of text.

Then you use the `range` function to generate indices for each 200-character substring, and the `i:i+200` slice notation to extract the substring from text.

Finally, you loop through each `chunk` in the `chunks` list and `print` it to the console.

As you can see because the chunking implementation is relatively simple and only based on length there are gaps within the sentences and even words.

For these reasons we believe that good NLP chunking has the following properties:

- Preserves entire words, ideally sentences and contextual points made by speakers.
- Handles for when sentences span across several pages, for example, page 1 into page 2.
- Provides an adequate token count for each `chunk`, so that the total number of `input` tokens will appropriately fit into a given token context window for any LLM.

Sliding Window Chunking

Sliding window chunking is a technique used for dividing text data into overlapping chunks, or *windows*, based on a specified number of characters.

But what exactly is a sliding window?

Imagine viewing a long piece of text through a small window. This window is only capable of displaying a fixed number of characters at a time. As you slide this window from the beginning to the end of the text, you see *overlapping chunks of text*. This mechanism forms the essence of the sliding window approach.

Each window size is defined by a *fixed number of characters*, and the *step size* determines how far the window moves with each slide.

In Figure 3-6, with a window size of 5 characters and a step size of 1, the first chunk would contain the first 5 characters of the text. The window then slides 1 character to the right to create the second chunk, which contains characters 2 through 6.

This process repeats until the end of the text is reached, ensuring each chunk overlaps with the previous and next ones to retain some shared context.

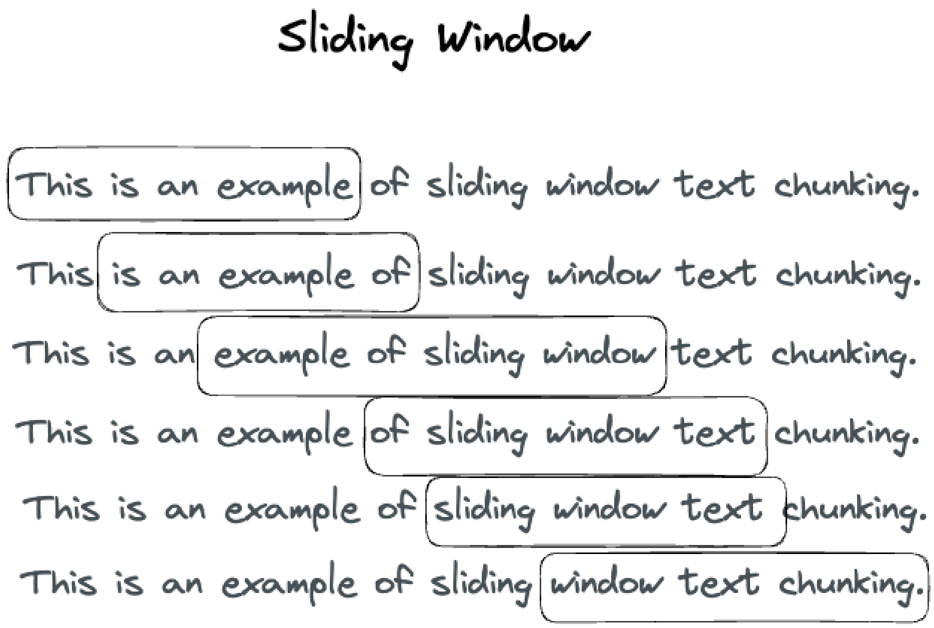

Figure 3-6. A sliding window, window size of 5 and a step size of 1

Due to the step size being 1, there is a lot of duplicate information between chunks and at the same time the risk of loosing information between chunks is dramatically reduced.

This is in stark contrast to Figure 3-7, which has a window size of 4 and a step size of 2. You'll notice that because of the 100% increase in step size, the amount of information shared between the chunks is greatly reduced.

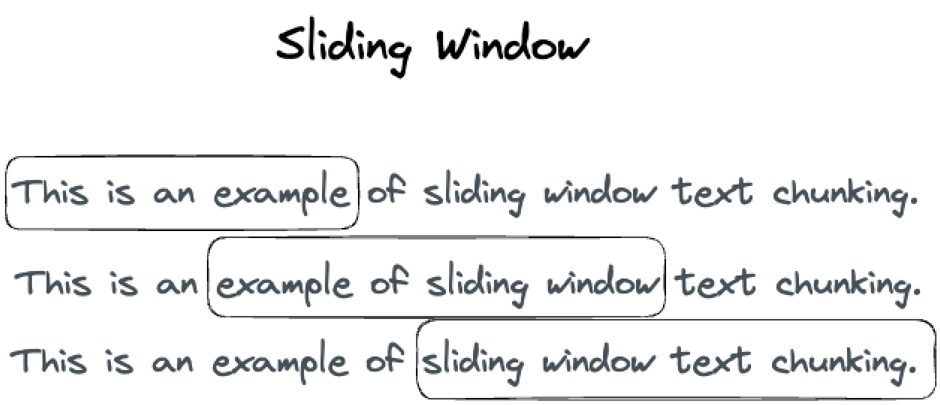

Figure 3-7. A sliding window, window size of 4 and a step size of 2

You will likely need a larger overlap if accuracy and preserving semantic context are more important than minimizing token inputs or the number of requests made to an LLM.

Example 3-6 shows how you can implement a sliding window using Python's `len()` function. The `len()` function provides us with the total number of characters in a given text string, which subsequently aids in defining the parameters of our sliding windows:

Example 3-6. Sliding Window

```
def sliding_window(text, window_size, step_size):
    if window_size > len(text) or step_size < 1:
        return []
    return [text[i:i+window_size] for i
    in range(0, len(text) - window_size + 1, step_size)]

text = "This is an example of sliding window text chunking."
window_size = 20
step_size = 5

chunks = sliding_window(text, window_size, step_size)

for idx, chunk in enumerate(chunks):
    print(f"Chunk {idx + 1}: {chunk}")
```

This code outputs:

```
Chunk 1: This is an example o
Chunk 2: is an example of sli
Chunk 3:  example of sliding
Chunk 4: ple of sliding windo
Chunk 5: f sliding window tex
Chunk 6: ding window text chu
Chunk 7: window text chunking
```

In the context of prompt engineering, the sliding window approach offers several benefits over fixed chunking methods. It allows LLMs to retain a higher degree of context, as there is an overlap between the chunks and offers an alternative approach to preserving context compared to sentence detection.

Text Chunking Packages

When working with LLMs such as GPT-4, always remain wary of the maximum context length:

`maximum_context_length = input_tokens + output_tokens`

There are various tokenizers available to break your text down into manageable units. The most popular ones being *NLTK*, *SpaCy*, and *tiktoken*.

Both NLTK and SpaCy provide comprehensive support for text processing, but you'll be focusing on *tiktoken*.

Text Chunking with Tiktoken

Tiktoken is a fast *Byte Pair Encoding (BPE)* tokeniser that breaks down text into subword units and is designed for use with OpenAI's models. Tiktoken offers faster performance than comparable open-source tokenisers.

As a developer working with GPT-4 applications, using tiktoken offers you several key advantages:

- *Accurate token breakdown*: It's crucial to divide text into tokens because GPT models interpret text as individual tokens. Identifying the number of tokens in your text helps you figure out whether the text is too lengthy for a model to process.
- *Effective resource utilization*: Having the correct token count enables you to manage resources efficiently, particularly when using the OpenAI API. Being aware of the exact number of tokens helps you regulate and optimize API usage, maintaining a balance between costs and resource usage.

Encodings

Encodings define the method of converting text into tokens, with different models utilizing different encodings. Tiktoken supports three encodings commonly used by OpenAI models:

Encoding name	OpenAI models
cl100k_base	GPT-4, GPT-3.5-turbo, text-embedding-ada-002
p50k_base	Codex models, text-davinci-002, text-davinci-003
r50k_base (or gpt2)	GPT-3 models like davinci

Understanding the tokenization of strings

In English, tokens can vary in length, ranging from a single character like *t*, to an entire word such as *great*. This is due to the adaptable nature of tokenization, which can accommodate even tokens shorter than a character in complex script languages or tokens longer than a word in languages without spaces or where phrases function as single units.

It is not uncommon for spaces to be included within tokens, such as `"is"` rather than `"is "` or `" "+"is"`. This practice helps maintain the original text formatting and can capture specific linguistic characteristics.

NOTE

To easily examine the tokenization of a string, you can use the OpenAI Tokenizer.

You can install Tiktoken from PyPI with `pip install tiktoken`.

In the following example, you'll see how to easily encode text into tokens and decode tokens into text.

```
# 1. Import the package:
import tiktoken

# 2. Load an encoding with tiktoken.get_encoding()
encoding = tiktoken.get_encoding("cl100k_base")

# 3. Turn some text into tokens with encoding.encode()
# while turning tokens into text with encoding.decode()
print(encoding.encode("Learning how to use Tiktoken is fun!"))
print(encoding.decode([1061, 15009, 374, 264, 2294, 1648,
311, 4048, 922, 15592, 0]))

# [48567, 1268, 311, 1005, 73842, 5963, 374, 2523, 0]
# "Data engineering is a great way to learn about AI!"
```

Additionally let's write a function that will tokenize the text and then count the number of tokens given a `text_string` and `encoding_name`.

```
def count_tokens(text_string: str, encoding_name: str) -> int:
    """
    Returns the number of tokens in a text string using a given encoding.

    Args:
        text: The text string to be tokenized.
        encoding_name: The name of the encoding to be used for tokenization.

    Returns:
        The number of tokens in the text string.

    Raises:
        ValueError: If the encoding name is not recognized.
    """
    encoding = tiktoken.get_encoding(encoding_name)
    num_tokens = len(encoding.encode(text_string))
    return num_tokens

# 4. Use the function to count the number of tokens in a text string.
text_string = "Hello world! This is a test."
print(count_tokens(text_string, "cl100k_base"))
```

Code Output:

```
8
```

Estimating Token Usage for Chat API Calls

ChatGPT models, such as GPT-3.5-turbo and GPT-4, utilize tokens similarly to previous completion models. However, the message-based structure makes token counting for conversations more challenging.

```
def num_tokens_from_messages(messages, model="gpt-3.5-turbo-0613"):
    """Return the number of tokens used by a list of messages."""
    try:
        encoding = tiktoken.encoding_for_model(model)
    except KeyError:
```

```
            print("Warning: model not found. Using cl100k_base encoding.")
            encoding = tiktoken.get_encoding("cl100k_base")
        if model in {
            "gpt-3.5-turbo-0613",
            "gpt-3.5-turbo-16k-0613",
            "gpt-4-0314",
            "gpt-4-32k-0314",
            "gpt-4-0613",
            "gpt-4-32k-0613",
            }:
            tokens_per_message = 3
            tokens_per_name = 1
        elif model == "gpt-3.5-turbo-0301":
            tokens_per_message = 4  # every message follows
            # <|start|>{role/name}\n{content}<|end|>\n
            tokens_per_name = -1  # if there's a name, the role is omitted
        elif "gpt-3.5-turbo" in model:
            print('''Warning: gpt-3.5-turbo may update over time. Returning
            num tokens assuming gpt-3.5-turbo-0613.''')
            return num_tokens_from_messages(messages, model="gpt-3.5-turbo-0613")
        elif "gpt-4" in model:
            print('''Warning: gpt-4 may update over time.
            Returning num tokens assuming gpt-4-0613.''')
            return num_tokens_from_messages(messages, model="gpt-4-0613")
        else:
            raise NotImplementedError(
                f"""num_tokens_from_messages() is not implemented for model
                {model}."""
            )
        num_tokens = 0
        for message in messages:
            num_tokens += tokens_per_message
            for key, value in message.items():
                num_tokens += len(encoding.encode(value))
                if key == "name":
                    num_tokens += tokens_per_name
        num_tokens += 3  # every reply is primed with
        # <|start|>assistant<|message|>
        return num_tokens
```

Example 3-7 highlights the specific structure required to make a request against any of the chat models which are currently GPT-3x and GPT-4.

Normally, chat history is structured with a `system` message first, then succeeded by alternating exchanges between the `user` and the `assistant`.

Example 3-7. A payload for the Chat Completions API on OpenAI

```
example_messages = [
    {
        "role": "system",
        "content": '''You are a helpful, pattern-following assistant that
        translates corporate jargon into plain English.''',
    },
    {
        "role": "system",
        "name": "example_user",
        "content": "New synergies will help drive top-line growth.",
    },
    {
        "role": "system",
        "name": "example_assistant",
        "content": "Things working well together will increase revenue.",
    },
    {
        "role": "system",
        "name": "example_user",
        "content": '''Let's circle back when we have more bandwidth to touch
        base on opportunities for increased leverage.''',
    },
    {
        "role": "system",
        "name": "example_assistant",
        "content": '''Let's talk later when we're less busy about how to
        do better.''',
    },
    {
        "role": "user",
        "content": '''This late pivot means we don't have
        time to boil the ocean for the client deliverable.''',
    },
]

for model in ["gpt-3.5-turbo-0301", "gpt-4-0314"]:
    print(model)
    # example token count from the function defined above
    print(f'''{num_tokens_from_messages(example_messages, model)}
    prompt tokens counted by num_tokens_from_messages().''')
```

- `"role": "system"` describes a system message that's useful for *providing prompt instructions*. It offers a means to tweak the assistant's character or provide explicit directives regarding its interactive approach. It's crucial to understand, though, that the system command isn't a prerequisite, and the model's default demeanor without a system command could closely resemble the behavior of "You are a helpful assistant."

- The roles that you can have are `["system", "user", "assistant"]`.
- `"content": "Some content"` is where you place the prompt or responses from a language model, depending upon what role.

Sentiment Analysis

Sentiment analysis is a widely used NLP technique that helps in identifying, extracting, and understanding the emotions, opinions, or sentiments expressed in a piece of text. By leveraging the power of LLMs like GPT-4, sentiment analysis has become an essential tool for businesses, researchers, and developers across various industries.

The primary goal of sentiment analysis is to determine the attitude or emotional tone conveyed in a text, whether it's positive, negative, or neutral. This information can provide valuable insights into consumer opinions about products or services, help monitor brand reputation, and even assist in predicting market trends.

Below are several prompt engineering techniques for creating effective sentiment analysis prompts:

Input:

```
Is this text positive or negative?

I absolutely love the design of this phone, but the battery life is quite
disappointing.
```

Output:

```
The text has a mixed tone, as it contains both positive and negative
aspects. The positive part is "I absolutely love the design of this phone,"
while the negative part is "the battery life is quite disappointing."
```

Although GPT-4 identifies a “mixed tone”, the outcome is a result of several shortcomings in the prompt:

1. *Lack of clarity*: The prompt does not clearly define the desired output format.
2. *Insufficient examples*: The prompt does not include any examples of positive, negative, or neutral sentiments, which could help guide the LLM in understanding the distinctions between them.
3. *No guidance on handling mixed sentiments*: The prompt does not specify how to handle cases where the text contains a mix of positive and negative sentiments.

Input:

```
Using the following examples as a guide:
positive: 'I absolutely love the design of this phone!'
negative: 'The battery life is quite disappointing.'
neutral: 'I liked the product, but it has short battery life.'

Only return either a single word of:
- positive
- negative
- neutral

Please classify the sentiment of the following text as positive, negative,
or neutral: I absolutely love the design of this phone, but the battery
life is quite disappointing.
```

Output:

```
neutral
```

This prompt is much better because it:

1. *Provides clear instructions*: The prompt clearly states the task, which is to classify the sentiment of the given text into one of three categories: positive, negative, or neutral.
2. *Offers examples*: The prompt provides examples for each of the sentiment categories, which helps in understanding the context and desired output.

3. *Defines the output format*: The prompt specifies that the output should be a single word, ensuring that the response is concise and easy to understand.

Techniques for Improving Sentiment Analysis

To enhance sentiment analysis accuracy, pre-processing the input text is a vital step. This involves the following:

- *Special characters removal*: Exceptional characters such as emojis, hashtags, and punctuation may skew the rule-based sentiment algorithm's judgment. Besides, these characters might not be recognized by machine learning and deep learning models, resulting in misclassification.
- *Lowercase conversion*: Converting all the characters to lowercase aids in creating uniformity. For instance, words like *Happy* and *happy* are treated as different words by models, which can cause duplication and inaccuracies.
- *Spelling correction*: Spelling errors can cause misinterpretation and misclassification. Creating a spell-check pipeline can significantly reduce such errors and improve results.

For industry or domain-specific text, embedding domain-specific instances in the prompt helps in navigating the LLMs sense of the text's framework and sentiment. It enhances accuracy in the classification and provides a heightened understanding of particular jargon and expressions.

Limitations and Challenges in Sentiment Analysis

Despite the advancements in LLMs and the application of prompt engineering techniques, sentiment analysis still faces some limitations and challenges:

- *Handling sarcasm and irony*: Detecting sarcasm and irony in text can be difficult for LLMs, as it often requires understanding the context and subtle cues that humans can easily recognize. Misinterpreting sarcastic or ironic statements may lead to inaccurate sentiment classification.
- *Identifying context-specific sentiment*: Sentiment analysis can be challenging when dealing with context-specific sentiments, such as those

related to domain-specific jargon or cultural expressions. LLMs may struggle to accurately classify sentiments in these cases without proper guidance or domain-specific examples.

Least To Most

The *least to most* technique in prompt engineering is a powerful method for sequentially generating or extracting increasingly detailed knowledge on a given topic. This method is particularly effective when dealing with complex subjects or when a high level of detail is necessary.

Least to most uses a *chain* of prompts where each new prompt is based on the last answer. This step-by-step approach helps gather more detailed information each time, making it easier to dive deeper into any topic.

This technique can also be applied to code generation, as demonstrated in a Flask `Hello World` app example.

Planning the Architecture

Before diving into the architecture, let's briefly understand what Flask is. Flask is a lightweight web application framework in Python, widely used for creating web applications quickly and with minimal code. (Flask is only used for demonstration purposes here and isn't included within the requirements.txt for the book.

Now, let's ask an LLM to outline the basic architecture for a simple Flask "Hello World" application:

Input:

```
Outline the basic architecture for a simple Flask "Hello World" application.
```

Output:

```
Sure! Here's an outline for a basic Flask "Hello World" application:

Import Flask module: Import the Flask module to use its functionality.

Create an instance of the Flask application: Create an instance of the
```

```
Flask class to represent your web application.
Define a route and view function: ...
Implement the view function: ...
Run the application: Start the Flask development server to run the applicatio
```

Coding Individual Functions

Before coding, let's clarify what a Flask route is, it's a function linked to a URL pattern that determines what users see on a specific web page. Next, we'll provide the code snippet for a simple "Hello World" route in Flask.

Input:

```
Provide a code snippet for a simple "Hello World" route in Flask.
```

Output:

```
Certainly! Here's a simple code snippet for a "Hello World" route using Flask

from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

In this code, we import the Flask module and create a Flask application instance named app. We then define a route using the `@app.route` decorator, specifying the root URL path `'/'`. The `hello_world` function is associated with this route, and it returns the string `'Hello, World!'` as the response.

The Flask application will start by calling `app.run()` when your Python script is executed directly.

Adding Tests

Finally, let's ask it to generate a test case for the "Hello World" route in the Flask application.

Input:

```
Provide an example of a test case for the "Hello World" route in a Flask
application.
```

Output:

```
Sure! Here's an example of a test case for the "Hello World" route in a
Flask application:

import unittest
from flask import Flask
from flask.testing import FlaskClient

# Create the Flask application
app = Flask(__name__)

# Create the route
@app.route('/')
def hello_world():
    return 'Hello, World!'

class FlaskAppTestCase(unittest.TestCase):
    def setUp(self):
        # Create a test client
        self.app = app.test_client()

    def test_hello_world_route(self):
        # Send a GET request to the route
        response = self.app.get('/')

        # Check the response status code
        self.assertEqual(response.status_code, 200)

        # Check the response data
        self.assertEqual(response.data, b'Hello, World!')

if __name__ == '__main__':
    unittest.main()
```

By using the *least to most* prompt technique, you can gradually build up the knowledge and understanding required for the task at hand.

Benefits of the Least To Most Technique

This method is particularly useful for complex tasks, as it allows an LLM to generate relevant knowledge, that will subsequently be used as context for future tasks.

Let's dive deeper into the benefits of using this approach in various other scenarios:

1. *Progressive exploration*: Breaking a complex problem into smaller tasks allows an LLM to provide more detailed and accurate information at each step. This approach is especially helpful when working with a new subject matter or a multifaceted problem.
2. *Flexibility*: The least to most technique offers flexibility in addressing different aspects of a problem. It enables you to pivot, explore alternative solutions, or dive deeper into specific areas as needed.
3. *Improved comprehension*: By breaking down a task into smaller steps, an LLM can deliver information in a more digestible format, making it easier for you to understand and follow.
4. *Collaborative learning*: This technique promotes collaboration between you and an LLM, as it encourages an iterative process of refining the output and adjusting your responses to achieve the desired outcome.

Challenges with the Least To Most Technique

1. *Over-reliance on previously generated knowledge*: Using previous chat history to store the state may lead to larger tasks forgetting their initial tasks/outputs due to limitations in context length.
2. *Dependence on prior prompts*: Since each prompt is built upon preceding outputs, it is imperative to ensure that the quality and responses of previous prompts provide ample information for the next step.

EVALUATE QUALITY

In the process of designing prompts, make sure to evaluate the quality of each prior LLM response. The performance of the next task depends upon the quality of information provided by the preceding one.

Role Prompting

Role prompting is a technique in which the AI is given a specific role or character to assume while generating a response. This helps guide the AI's response style and content to better align with the user's needs or desired outcome.

NOTE

Awesome ChatGPT prompts provides an extensive list of role prompts that you can use.

Let's ask ChatGPT to take on the role of a tech reviewer for MongoDB:

Input:

```
I want you to act as a tech reviewer. I will give you the name of a piece
of technology and you will provide me with an in-depth review - including
pros, cons, features, and comparisons to other technologies on the market.
My first suggestion request is: MongoDB.
```

Output:

```
MongoDB is a popular NoSQL, document-oriented database management system
designed to handle high-volume data storage and retrieval.

Features:

1. Document-Oriented Storage: MongoDB stores data in a flexible, JSON-like
format called BSON, making it easy to store complex data structures like
arrays and nested objects.
...

Pros:
1. High Performance: MongoDB's indexing and caching capabilities deliver
```

```
fast read and write operations.
...

Cons:
1. Limited Support for Transactions: While MongoDB supports multi-document
ACID transactions, it may not be the best choice for applications that
require complex transactional operations.
...


Comparisons to other technologies:
1. Couchbase, another popular NoSQL database, offers similar features to
MongoDB, including flexible schema and JSON-based storage. However,
Couchbase is more focused on distributed caching and may be more
suitable for applications requiring low-latency data access.

Conclusion:

MongoDB is a powerful and versatile NoSQL database solution suitable for
various applications, particularly those requiring high performance and
scalability. Its flexible data model, rich query language, and strong
consistency make it a popular choice among developers.
```

The prompt is an excellent example of role prompting, as it clearly defines the role the AI should assume (a tech reviewer) and sets expectations for the type of response desired (an in-depth review of MongoDB).

GIVE DIRECTION

When crafting prompts, consider assigning a specific role to the AI. This sets the proper context for the response, creating a more focused and relevant output.

Benefits of Role Prompting

Role prompting helps narrow down the AI's responses, ensuring more focused, contextually appropriate, and tailored results. It can also enhance creativity by pushing the AI to think and respond from unique perspectives.

Challenges of Role Prompting

Role prompting can pose certain challenges. There might be potential risks for bias or stereotyping based on the role assigned. Assigning stereotyped roles can lead to generating biased responses, which could harm usability or offend individuals. Additionally, maintaining consistency in the role throughout an extended interaction can be difficult. The model might drift off-topic or respond with information irrelevant to the assigned role.

EVALUATE QUALITY

Consistently check the quality of the LLMs responses, especially when role prompting is in play. Monitor if the AI is sticking to the role assigned or if it is veering off-topic.

When to use Role Prompting

Role prompting is particularly useful when you want to:

- *Elicit specific expertise*: If you need a response that requires domain knowledge or specialized expertise, role prompting can help guide the LLM to generate more informed and accurate responses.
- *Tailor response style*: Assigning a role can help an LLM generate responses that match a specific tone, style, or perspective, such as a formal, casual, or humorous response.
- *Encourage creative responses*: Role prompting can be used to create fictional scenarios or generate imaginative answers by assigning roles like a storyteller, a character from a novel, or a historical figure.
- *Explore diverse perspectives*: If you want to explore different viewpoints on a topic, role prompting can help by asking the AI to assume various roles or personas, allowing for a more comprehensive understanding of the subject.
- *Enhance user engagement*: Role prompting can make interactions more engaging and entertaining by enabling an LLM to take on characters or personas that resonate with the user.

If you're using OpenAI, then the best place to add a role is within the `System Message` for chat models.

GPT Prompting Tactics

So far you've already covered several prompting tactics including:

- Asking for Context.
- Text Style Bundling.
- Least to Most.
- Role Prompting.

Let's cover several more tactics, from managing potential hallucinations with appropriate reference text, to providing an LLM with critical *thinking time*, to understanding the concept of *task decomposition*, we have plenty for you to explore.

These methodologies have been designed to significantly boost the precision of your AI's output and are recommended by OpenAI. Also, each tactic utilizes one or more of the prompt engineering principles discussed in Chapter 1.

Avoiding Hallucinations with Reference

The first method for avoid text based hallucinations is to instruct the model to *only answer using reference text.*

By supplying an AI model with accurate and relevant information about a given query, the model can be directed to use this information to generate its response.

Input:

```
Refer to the articles enclosed within triple quotes to respond to queries.

You must follow the following principles:
- In cases where the answer isn't found within these articles, simply
return "I could not find an answer".

"""
B2B clients tend to have longer decision-making processes, and thus longer
sales funnels. Relationship-building strategies work better for these
```

```
clients, whereas B2C customers tend to respond better to short-term offers
and messages.
"""

Example responses:
- I could not find an answer.
- Yes, B2B clients tend to have longer decision-making processes, and thus
longer sales funnels.
```

Output:

```
Yes, B2B clients tend to have longer decision-making processes, which leads
to longer sales cycles.
```

If you were to ask the same reference text this question:

Input:

```
...The rest of the prompt...

Question: Are B2C sales more cost effective?
```

Output:

```
I could not find an answer.
```

GIVE DIRECTION & SPECIFY FORMAT

The above prompt is excellent as it both instructs the model on how to find answers, and also sets a specific response format for any unanswerable questions.

Considering the constrained context windows of GPTs, a method for dynamically retrieving information relevant to the asked query might be necessary to utilize this strategy.

Another approach is to direct the model to *incorporate references* from a given text in its response. When the input is enhanced with relevant information, the model can be guided to include citations in its responses by referring to sections of the supplied documents. This approach has the added benefit that citations in the output can be *authenticated automatically by matching strings* within the given documents:

Input:

```
You will be provided with a document delimited by triple quotes and a
question. Your task is to answer the question using only the provided
document and to cite the passage(s) of the document used to answer the
question. If the document does not contain the information needed to answer
this question then simply write: "Insufficient information." If an answer
to the question is provided, it must be annotated with a citation. Use the
following format for to cite relevant passages ({"citation": …}).

"""
A nuclear reactor is a device that initiates, controls, and maintains a
nuclear chain reaction. The nuclear reactor uses the energy released by
nuclear fission to generate heat, which is then often used to produce steam
to drive a turbine and produce electricity.

-Excerpted from https://en.wikipedia.org/wiki/Nuclear_reactor
"""

Question: What is a nuclear reactor?
```

Output:

```
A nuclear reactor is a device that initiates, controls, and maintains a
nuclear chain reaction.It uses the energy released by nuclear fission to
generate heat, which is then often used to produce steam to drive a turbine
and produce electricity. {"citation": "A nuclear reactor is a device that
initiates, controls, and maintains a nuclear chain reaction. The nuclear
reactor uses the energy released by nuclear fission to generate heat, which
is then often used to produce steam to drive a turbine and produce
electricity."}
```

Give GPTs “Thinking Time”

Often, by explicitly guiding an LLM to *derive solutions from first principles* before reaching a verdict, you can garner more accurate responses. Providing an LLM with *thinking time* can often lead to better results.

Input:

```
First, try to solve the problem on your own. Then, see if your answer
matches the proposed solution. Don't judge the given answer until you've
tried solving it yourself.

Problem:
Jane has 5 apples, then John takes 2 from Mary. Mary originally had 15
apples. John then gives his apples to Jane.

How many apples does Jane have?

Proposed solution:
1. Janes originally had 5 apples.
2. John takes 2 apples from Mary, but this doesn't affect Jane's count.
3. John then gives his apples to Jane.

So, Jane's final count of apples is her original 5 apples plus the 2 apples
that John gave her, for a total of 7 apples.
```

Output:

```
Let's break down the problem:

1. Jane originally has 5 apples.
2. John takes 2 apples from Mary. However, this action doesn't directly
impact the number of apples Jane has at this point.
3. John gives his apples to Jane.

Following these steps, we need to add the 2 apples John gave to Jane to
Jane's original number of apples. Therefore:

5 apples (Jane's original number) + 2 apples (given by John) = 7 apples

So, Jane now has 7 apples. My solution matches the proposed solution.
```

In some cases, when using GPT-4 or other generative AI models, you may not want the model's reasoning process to be visible to the user.

You can achieve this by asking an LLM to *generate an inner monologue*. This is particularly useful in tutoring applications, where revealing the model's reasoning might give away the solution prematurely.

The Inner Monologue Tactic

The *inner monologue tactic* instructs the model to structure parts of the output that should be hidden from the user in a specific format. This makes it easy to remove these parts before presenting the final output to the user.

Here's how you can utilize this tactic to answer user queries effectively:

Input:

```
Step 1: Solve the Problem

First, you need to work out your solution to the problem. Be sure not to
rely on the user's solution, as it might be incorrect. You can encapsulate
your workings within triple quotes (`"""`) to indicate that this section
should be hidden.

Step 2: Compare Solutions

Next, compare your solution with the user's solution. Evaluate whether the
user's solution is correct or not. Again, encapsulate this section within
triple quotes (`"""`).

Step 3: Prepare a Hint

If the user made a mistake, determine an appropriate hint that can help
guide them towards the correct answer, without giving it away outright.
This section should also be within triple quotes (`"""`).

Step 4: Deliver the Hint

If a mistake was made, present the hint from the previous step to the user.
Remember not to use triple quotes for this step. Instead of using a title
like "Step 4", use a heading such as `Hint:`

---

Problem Statement: A deck of cards has 52 cards: 13 hearts, 13 diamonds, 13
clubs, and 13 spades. If two cards are drawn from the deck without
replacement, what is the probability that one of the drawn cards is a heart
and the other one is a club?

User's Solution: (13/52)*(13/51) = 169/2652 = 13/204
```

Output:

```
"""
Step 1 - My Solution:

First, calculate the probability of drawing one heart and one club from a
deck of 52 cards without replacement.

In the first draw, the probability of drawing a heart is 13/52.

For the second draw, after removing one card (assuming it's a heart), the
total number of cards becomes 51. So, the probability of drawing a club now
is 13/51.

Multiplying these two probabilities gives the probability of both events
happening in sequence,which is P(Heart then Club) = (13/52)*(13/51).

Repeat the calculation assuming the first drawn card is a club and the
second drawn card is a heart. Remember, order matters in this case as
we're dealing with separate events.
"""
"""
Step 2 - Comparison with the User's Solution:

Compare your solution with the user's solution. The user's solution only
considers the sequence of drawing a heart first and then a club. They
have not accounted for the alternate sequence  (Club then Heart), and
thus, the solution is incomplete.
"""
"""
Step 3 - Preparation of Hint:

Given that the user hasn't considered both sequences (Heart then Club
and Club then Heart), a hint at the consideration of both possible
sequences will be helpful. This could nudge the user in the right
direction without revealing the solution outright.
"""
# Step 4 - Delivery of Hint:
# Hint: Remember that the two cards could be drawn in any order. Consider
both possible sequences: Heart then Club and Club then Heart.
```

Self-Eval LLM Responses

Another tactic you can use is to *critque a generated LLM output* and ask whether the LLM missed any information or important facts. You're essentially asking an LLM, to *evaluate itself* based on it's previous output.

First, create a prompt to generate a "Hello World" function:

Input:

```
Generate a hello world python function.
```

Output:

```
Sure, here's a very simple Python function that prints "Hello, World!" to
the console:

def print_hello_world():
    print("Hello, World!")

print_hello_world()

When you run this function, it will print the string "Hello, World!" to the
console.
```

The LLM returns a simple Python function called `print_hello_world()` which prints the traditional `"Hello, World!"` greeting.

Then make a second LLM request with the previous chat history, asking for the initial output to be improved:

Input:

```
...Previous Chat History...

Given the previous function, can you improve it again with anything else?
Perhaps defaults or type hints?
```

Output:

```
We can add type hints to make it clear that the function doesn't take any
arguments and doesn't return anything. We can also add a docstring to
describe what the function does.
```

```
Here's the improved code:

def print_hello_world() -> None:
    """
    Prints "Hello, World!" to the standard output.
    """
    print("Hello, World!")

If you wanted to make the function more flexible, you could add a default
argument that allows you to print a different message:

def print_message(message: str = "Hello, World!") -> None:
    """
    Prints the given message to the standard output. If no message is given
    , it defaults to "Hello, World!".

    Args:
        message (str, optional): The message to be printed.
        Defaults to "Hello, World!".
    """
    print(message)
```

Consequently GPT-4 returns an upgraded version of the function, now furnished with Python type hints and a default argument. This enriches the function with greater flexibility, allowing it to print not just `"Hello, World!"`, but any user-specified message.

These prompt-response exchanges illustrate how you can easily refine generated LLM outputs until you're satisfied with the final output.

NOTE

It's possible to critique an LLMs response multiple times, until no further refinement is provided by the LLM.

Classification with LLMs

Classifying, in the context of AI, refers to the process of predicting the class or category of a given data point or sample. It's a common task in machine learning where models are trained to assign pre-defined labels to unlabeled data based on learned patterns.

LLMs are powerful assets when it comes to classification, even with zero or only a small number of examples provided within a prompt. Why? That's because LLMs, like GPT-4, have been previously trained on an extensive dataset and now possess a degree of reasoning.

There are two overarching strategies in solving classification problems with LLMs: *zero-shot learning* and *few-shot learning*.

- *Zero-shot learning*: In this process, the LLM classifies data with exceptional accuracy, without the aid of any prior specific examples. It's akin to acing a project without any preparation – impressive, right?
- *Few-shot learning*: Here, you provide your LLM with a small number of examples. This strategy can significantly influence the structure of your output format and enhance the overall classification accuracy.

Why is this groundbreaking for you?

Leveraging LLMs lets you sidestep lengthy processes that traditional machine learning processes demand. Therefore you can quickly prototype a classification model, determine a base level accuracy and create immediate business value.

WARNING

Although an LLM can perform classification, depending upon your problem and training data you might find that using a traditional machine learning process could yield better results.

Building A Classification Model

Let's explore a few-shot learning example to determine the sentiment of text into either `'Compliment'`, `'Complaint'` or `'Neutral'`.

```
Given the statement, classify it as either "Compliment", "Complaint", or
"Neutral":
1. "The sun is shining." - Neutral
2. "Your support team is fantastic!" - Compliment
3. "I had a terrible experience with your software." - Complaint

You must follow the following principles:
```

```
- Only return the single classification word. The response should be either
"Compliment", "Complaint" or "Neutral".
- Perform the classification on the text enclosed within """ delimiters.

"""The user interface is intuitive."""

Classification:
```

```
Compliment
```

Several good use cases for LLM classification include:

1. Customer Reviews: Classify user reviews into categories like “Positive”, “Negative”, or “Neutral”. Dive deeper by further identifying sub-themes such as “Usability”, “Customer Support”, or “Price”.
2. Email Filtering: Detect the intent or purpose of emails and classify them as “Inquiry”, “Complaint”, “Feedback”, or “Spam”. This can help businesses prioritize responses and manage communications efficiently.
3. Social Media Sentiment Analysis: Monitor brand mentions and sentiment across social media platforms. Classify posts or comments as “Praise”, “Critic”, “Query”, or “Neutral”. Gain insights into public perception and adapt marketing or PR strategies accordingly.
4. News Article Categorization: Given the vast amount of news generated daily, LLMs can classify articles by themes or topics such as “Politics”, “Technology”, “Environment”, or “Entertainment”.
5. Resume Screening: For HR departments inundated with resumes, classify them based on predefined criteria like “Qualified”, “Overqualified”, “Underqualified”, or categorize by expertise areas such as “Software Development”, “Marketing”, or “Sales”.

WARNING

Be aware that exposing emails, resumes or sensitive data does run the risk of data being leaked into OpenAI’s future models as training data.

Majority Vote For Classification

Utilizing multiple LLM requests can help in reducing the variance of your classification labels. This process, known as *majority vote*, is somewhat like choosing the most common fruit out of a bunch. For instance, if you have 10 pieces of fruit and 6 out of them are apples, then apples are the majority. The same principle goes for choosing the majority vote in classification labels.

By soliciting several classifications and taking the *most frequent classification*, you're able to reduce the impact of potential outliers or unusual interpretations from a single model inference. However, do bear in mind that there can be significant downsides to this approach including the increased time required and cost for multiple API calls.

Let's classify the same piece of text three times, then take the majority vote:

```
from openai import OpenAI
import os

client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

base_template = """
Given the statement, classify it as either "Compliment", "Complaint", or
"Neutral":
1. "The sun is shining." - Neutral
2. "Your support team is fantastic!" - Compliment
3. "I had a terrible experience with your software." - Complaint

You must follow the following principles:
- Only return the single classification word. The response should be either
"Compliment", "Complaint" or "Neutral".
- Perform the classification on the text enclosed within ''' delimiters.

'''{content}'''

Classification:
"""

responses = []

for i in range(0, 3):
    response = client.chat.completions.create(
```

```
        model="gpt-4",
        messages=[{"role": "system",
            "content": base_template.format(content='''Outside is rainy,
            but I am having a great day, I just don't understand how people
            live, I'm so sad!'''),}],)
    responses.append(response.choices[0].message.content.strip())

def most_frequent_classification(responses):
    # Use a dictionary to count occurrences of each classification
    count_dict = {}
    for classification in responses:
        count_dict[classification] = count_dict.get(classification, 0) + 1

    # Return the classification with the maximum count
    return max(count_dict, key=count_dict.get)

print(most_frequent_classification(responses))  # Expected Output: Neutral
```

Calling the `most_frequent_classification(responses)` function should pinpoint `'Neutral'` as the dominant sentiment. You've now learned how to use the `OpenAI` package for majority vote classification.

Criteria Evaluation

In Chapter 1 a human-based evaluation system was used with a simple thumbs up / thumbs down rating system, to identify how often a response met our expectations. Rating manually can be expensive and tedious, requiring a qualified human to judge quality or identify errors. While this work can be outsourced to low cost raters on services such as Mechanical Turk, designing such a task in a way that gets valid results can itself be time-consuming and error prone. One increasingly common approach is to use a more sophisticated LLM to evaluate the responses of a smaller model.

The evidence is mixed on whether LLMs can act as effective evaluators, with some studies claiming LLMs are human-level evaluators, and others identifying incosistencies in how LLMs evaluate. In the author's experience, GPT-4 is a useful evaluator with consistent results across a diverse set of tasks. In particular, GPT-4 is effective and reliable in evaluating the responses from smaller, less sophisticated models like GPT-3.5-turbo. In the example that follows, we generate concise and verbose examples of answers to a question using GPT-3.5-turbo, ready for rating with GPT-4.

Input:

```
from openai import OpenAI
import os

client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

responses = []

for i in range(10):
    # concise if even, verbose if odd
    style = "concise" if i % 2 == 0 else "verbose"

    if style == "concise":
        prompt = f"""Return a {style} answer to the
        following question: What is the meaning of life?"""
    else:
        prompt = f"""Return an answer to the following
        question: What is the meaning of life?"""

    response = client.chat.completions.create(
        # using GPT-3.5 Turbo for this example
        model="gpt-3.5-turbo",
        messages=[{"role": "user",
            "content": prompt}])
    responses.append(
        response.choices[0].message.content.strip())

system_prompt = """You are assessing the conciseness of a
response from a chatbot.
You only respond with a 1 if the response is concise,
and a 0 if it is not.
"""

ratings = []

for idx, response in enumerate(responses):
    rating = client.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "system",
            "content": system_prompt},
            {"role": "system",
            "content": response}])
    ratings.append(
        rating.choices[0].message.content.strip())

for idx, rating in enumerate(ratings):
```

```
    style = "concise" if idx % 2 == 0 else "verbose"
    print(f"Style: {style}, ", f"Rating: {rating}")
```

Output:

```
Style: concise,  Rating: 1
Style: verbose,  Rating: 0
Style: concise,  Rating: 1
Style: verbose,  Rating: 0
Style: concise,  Rating: 1
Style: verbose,  Rating: 0
Style: concise,  Rating: 1
Style: verbose,  Rating: 0
Style: concise,  Rating: 1
Style: verbose,  Rating: 0
```

Here's what this code does step-by-step:

This script is a Python program that interacts with the OpenAI API to generate and evaluate responses based on their conciseness. Here's a step-by-step explanation:

1. `responses = []`: Creates an empty list named `responses` to store the responses generated by the OpenAI API.
2. The `for` loop runs 10 times, generating a response for each iteration.
3. Inside the loop, `style` is determined based on the current iteration number (`i`). It alternates between "concise" and "verbose" for even and odd iterations, respectively.
4. Depending on the `style`, a `prompt` string is formatted to ask "What is the meaning of life?" in either a concise or verbose manner.
5. `response = client.chat.completions.create(...)`: The script makes a request to the OpenAI API to generate a response based on the `prompt`. The model used here is specified as "gpt-3.5-turbo".
6. The generated response is then stripped of any leading or trailing whitespace and added to the `responses` list.
7. `system_prompt = """You are assessing..."""`: This sets up a prompt used for evaluating the conciseness of the generated responses.

8. `ratings = []`: Initializes an empty list to store the conciseness ratings.
9. Another `for` loop iterates over each response in `responses`.
10. For each response, the script sends it along with the `system_prompt` to the OpenAI API, requesting a conciseness evaluation. This time, the model used is "gpt-4".
11. The evaluation rating (either 1 for concise or 0 for not concise) is then stripped of whitespace and added to the `ratings` list.
12. The final `for` loop iterates over the `ratings` list. For each rating, it prints the `style` of the response (either "concise" or "verbose") and its corresponding conciseness `rating`.

For simple ratings like conciseness, GPT-4 performs with near 100% accuracy, however for more complex ratings it's important to spend some time evaluating the evaluator. For example, by setting test cases that contain an issue, as well as test cases that do not contain an issue, you can identify the accuracy of your evaluation metric. An evaluator can itself be evaluated by counting the number of false positives (when the LLM hallucinates an issue in a test case that is known not to contain an issue), as well as the number of false negatives (when the LLM misses an issue in a test case that is known to contain an issue). In our example we generated the concise and verbose examples, so we can easily check the rating accuracy, but in more complex examples you may need human evaluators to validate the ratings.

EVALUATE QUALITY

Using GPT-4 to evaluate the responses of less sophisticated models is an emerging standard practice, but care must be taken that the results are reliable and consistent.

Compared to human-based evaluation, LLM-based or synthetic evaluation typically costs an order of magnitude less and completes in a few minutes rather than taking days or weeks. Even in important or sensitive cases where a final manual review by a human is necessary, rapid iteration and A/B testing of the prompt through synthetic reviews can save significant time and improve results considerably. However, the cost of running many tests at scale can add up, and the latency or rate limits of GPT-4 can be a blocker. If at all possible, a prompt engineer should first test using programmatic techniques that don't require a call to an LLM, such as simply measuring the length of the response, which runs near instantly for close to zero cost.

Meta Prompting

Meta prompting is a technique that involves the creation of text prompts that, in turn, generate other text prompts. These text prompts are then used to generate new assets in many mediums such as images, videos and more text.

To better understand meta prompting, let's take the example of authoring a children's book with the assistance of GPT-4. First, you direct the LLM to generate the text for your children's book. Afterward, you invoke meta prompting by instructing GPT-4 to produce prompts that are suitable for image-generation models. This could mean creating situational descriptions or specific scenes based on the storyline of your book, which then can be given to AI models like Midjourney or Stable Diffusion. These image-generation models can, therefore, deliver images in harmony with your AI-crafted children's story.

Figure 3-8 below visually describes the process of meta prompting in the context of crafting a children's book:

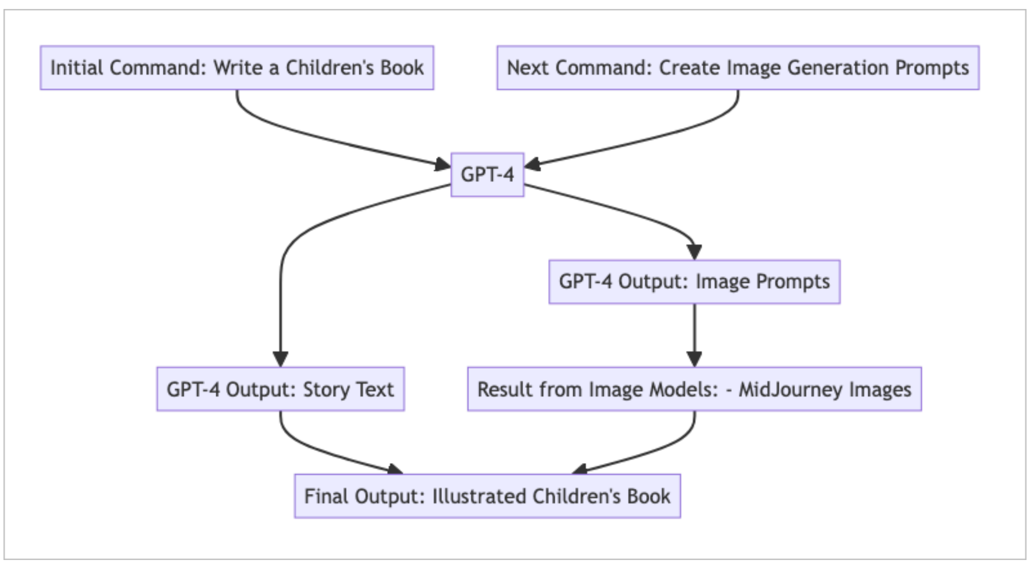

Figure 3-8. Utilizing an LLM to generate image prompts for MidJourney's image creation in the process of crafting a children's book

Meta prompts offer a multitude of benefits for a variety of applications:

- *Image generation from product descriptions*: Meta prompts can be employed to derive an image generation prompt for image models like Midjourney, effectively creating a visual representation of product descriptions.

- *Generating style/feature prompts*: Let's consider you are a copywriter needing to develop a unique style guide prompt from a couple of blog posts. Given each client has a distinctive tone and style, it's beneficial to utilize a *meta prompt* that encapsulates all the varied features, rather than producing a single prompt output.
- *Optimizing prompts to achieve specific goals*: A common approach is to ask ChatGPT or another language model to refine or improve `Prompt A` in order to attain `Goal 1`, given `Prompt A` and `Goal 1`. This method aids in identifying any missed input features, that by adding could assist the language model in generating a more detailed and desirable response.

Let's experiment with creating a meta prompt for the copywriting example:

Input:

```
Act as a GPT Prompt Engineer, you are solely responsible for producing
highly effective large language model text prompts given a Task.

You must follow the following principles:
- Return only a single prompt, never return the output of a prompt.
- You must follow the Task properly.
- You must either refine the prompt or create a new prompt depending upon
the Task.
- Bullet point out any important features so that another Prompt Engineer
can easily copy inputs into the prompt.

Task: You must create a copywriting guide ChatGPT several blog posts.
Include relevant features within the ChatGPT prompt that will allow ChatGPT
to imitate the writer. You must decide what the most important features are
to mimic the writer's style etc. You will only be applying this style to
new blog posts.

Blog post text: You need a variety of soft skills in addition to technical
skills to succeed in the technology sector. Soft skills are used by
software professionals to collaborate with their peers...
```

Output:

```
Prompt:

Based on the extensive text provided, your task is to create a series of
blog posts following the same style and structure of the writer. As a guide
you should follow these features:

1. Clear, direct language: Your writing should be in clear and direct
language, breaking down complex subjects into easily understandable
concepts.
2. Use of subheadings: To provide structure to your blog posts, break them
up with relevant subheadings. This helps to guide the reader and make your
text more readable.
...
9. Inclusion of technical terminology: When discussing technical topics,
use appropriate technical language to ensure accuracy and credibility.
However, be sure to provide explanations or definitions for more complex
terms.
```

As previously mentioned, it can be beneficial to *critique the model's output* and to ask for a refined prompt either via the interface or API.

ChatGPT can perform web search and can make use of the content of web pages, along with the previously generated copywriting style guide. Let's provide 2 URLs that ChatGPT will read to enhance the output.

Input:

```
Given the following style guide and these 2 blog posts. I want you to
improve the copywriting style prompt so an LLM or copywriter can write
effectively for my client.

...Previously Generated Copywriting guide...

URL 1: https://understandingdata.com/posts/how-to-clean-and-process-data
URL 2: https://understandingdata.com/posts/how-to-scrape-twitter-data
```

The output is shown in Figure 3-9.

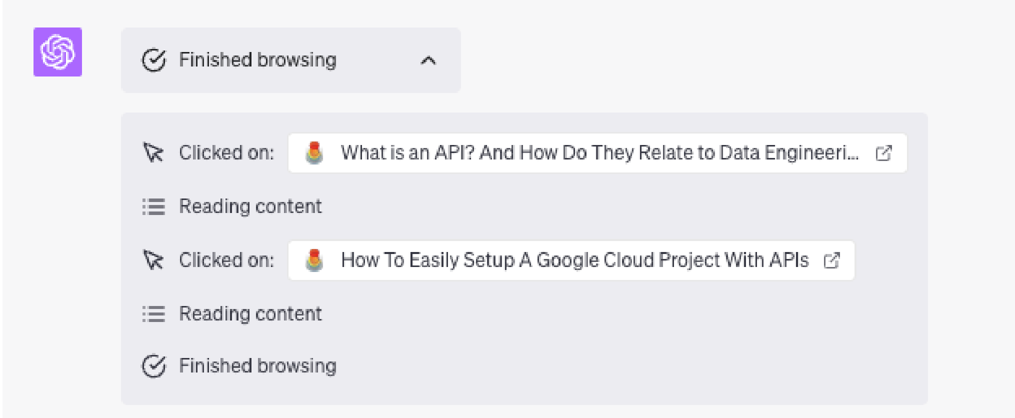

Figure 3-9. ChatGPT refining a meta prompt by 2 URL web pages

```
Based on the style guide and the two blog posts, here's an improved
copywriting style prompt:

To create a blog post for the client, adhere to the following guidelines:

1. Clarity and Directness: Use clear, direct language to explain complex
topics. Present information in an easy-to-understand manner. Aim for your
audience to grasp concepts without needing prior knowledge.
...
8. Use of Technical Terminology: Use the appropriate technical language
when discussing complex topics. However, make sure to provide definitions
or explanations for these terms to keep the text accessible.

Model your writing after these blog post examples:

"What is an API? And How Do They Relate to Data Engineering?": This blog
post uses clear and direct language to explain the concept of an API. It
provides detailed explanations and uses practical examples of commonly used
APIs. It links the technical concept of APIs to business growth and
productivity. The information is well-categorized using subheadings like
'Types of API's'.

"How To Easily Setup A Google Cloud Project With APIs": This blog post is a
practical guide on setting up a Google Cloud Project. It's broken down into
numbered steps, each with a clear subheading. The post uses detailed
explanations and includes practical examples such as screenshots. It also
uses clear, direct language to guide the reader through the process.
```

Meta prompting offers a dynamic and innovative way to harness the power of generative AI models, fostering the creation of complex, multifaceted prompts and even prompts that generate other prompts. It broadens the application scope, from text and image generation to style and feature prompts, and optimization towards specific goals. As you continue to refine and explore the potential of meta prompting, it promises to be a game-changer in how you utilize, interact with, and benefit from using LLMs.

Summary

You're now aware of how crucial it is to give clear directions and examples to generate desired outputs.

Also, you have hands-on experience extracting structured data from a hierarchical list using regular expressions in Python and you've learned to utilize nested data structures like JSON and YAML to produce robust, parsable outputs.

You've learned several best practices and effective prompt engineering techniques including:

- List generation.
- Explain it like I'm five.
- Asking for Context.
- Chunking text.
- Sentiment analysis.
- Least to most.
- Role prompting.
- Using reference text.
- Giving GPT time to think.
- Critiquing an LLMs output.
- LLM based classification.
- Meta prompting.

In the next chapter, you will learn how to use a popular LLM package called LangChain that'll help you to create even more advanced prompt engineering workflows.