Types of Prompts

Download the required Dependencies

[18]:
```
!pip install -q --upgrade openai langchain langchain-community langchain_openai␣
↪openai
```

[4]:
```
OPENAI_API_KEY="Enter your api key"
```

[5]:
```
import os
os.environ["OPENAI_API_KEY"] = OPENAI_API_KEY
```

[6]:
```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(
    temperature=0,
    model_name="gpt-3.5-turbo"
)
```

1 Zero Short Prompt

This involves giving the AI a task without any prior examples. You describe what you want in detail, assuming the AI has no prior knowledge of the task.

[7]:
```
from langchain import PromptTemplate

template="what is mean by large language model ?"

# Prompt template (simple instruction)
template = PromptTemplate(template=template)

# Send prompt and get response
response = llm.invoke(template.format())

# Print response
print(response.content)
```

```
A large language model is a type of artificial intelligence system that is
trained on vast amounts of text data in order to understand and generate human
```

language. These models are capable of processing and generating text in a way that is very similar to how humans communicate. They are typically used for tasks such as language translation, text generation, and natural language understanding. Examples of large language models include GPT-3 (Generative Pre-trained Transformer 3) and BERT (Bidirectional Encoder Representations from Transformers).

2 One Short Prompt

You provide one example along with your prompt. This helps the AI understand the context or format you're expecting.

[8]:
```
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate


# Create a one-shot prompt
template = """
A Foundation Model in AI refers to a model like GPT-3,
which is trained on a large dataset and can be adapted to various tasks.
Explain what BERT is in this context.
"""

# Define the prompt template
prompt_template = PromptTemplate(
    input_variables=[],
    template=template
)

# Generate the response
response = llm.invoke(prompt_template.format())

print(response.content)
```

BERT (Bidirectional Encoder Representations from Transformers) is another example of a Foundation Model in AI. It is a pre-trained model developed by Google that is trained on a large corpus of text data. BERT is designed to understand the context of words in a sentence by considering the words that come before and after each word. This bidirectional approach allows BERT to capture more complex relationships and nuances in language compared to traditional models. Like GPT-3, BERT can be fine-tuned for specific tasks such as text classification, question answering, and named entity recognition.

3 Few Short prompt

It is a technique where you provide a small set of examples or instructions to guide the model towards a specific task or response style

[9]:
```
from langchain import PromptTemplate, FewShotPromptTemplate

# First, create the list of few shot examples.
examples = [
    {"word": "happy", "antonym": "sad"},
    {"word": "tall", "antonym": "short"},
]

# Next, we specify the template to format the examples we have provided.
# We use the `PromptTemplate` class for this.
example_formatter_template = """Word: {word}
Antonym: {antonym}
"""

example_prompt = PromptTemplate(
    input_variables=["word", "antonym"],
    template=example_formatter_template,
)
```

[10]:
```
# Finally, we create the `FewShotPromptTemplate` object.
few_shot_prompt = FewShotPromptTemplate(
    # These are the examples we want to insert into the prompt.
    examples=examples,
    # This is how we want to format the examples when we insert them into the␣
 ↪prompt.
    example_prompt=example_prompt,
    # The prefix is some text that goes before the examples in the prompt.
    # Usually, this consists of intructions.
    prefix="Give the antonym of every input\n",
    # The suffix is some text that goes after the examples in the prompt.
    # Usually, this is where the user input will go
    suffix="Word: {input}\nAntonym: ",
    # The input variables are the variables that the overall prompt expects.
    input_variables=["input"],
    # The example_separator is the string we will use to join the prefix,␣
 ↪examples, and suffix together with.
    example_separator="\n",
)
```

[11]:
```
print(few_shot_prompt.format(input="big"))
```

```
Give the antonym of every input

Word: happy
Antonym: sad

Word: tall
```

Antonym: short

Word: big
Antonym:

[12]:
```
from langchain.chains import LLMChain
chain=LLMChain(llm=llm,prompt=few_shot_prompt)
chain({"input":"big"})
```

```
/usr/local/lib/python3.10/dist-packages/langchain_core/_api/deprecation.py:139:
LangChainDeprecationWarning: The class `LLMChain` was deprecated in LangChain
0.1.17 and will be removed in 0.3.0. Use RunnableSequence, e.g., `prompt | llm`
instead.
  warn_deprecated(
/usr/local/lib/python3.10/dist-packages/langchain_core/_api/deprecation.py:139:
LangChainDeprecationWarning: The method `Chain._call_` was deprecated in
langchain 0.1.0 and will be removed in 0.3.0. Use invoke instead.
  warn_deprecated(
```

[12]:
```
{'input': 'big', 'text': 'small'}
```

4 Chain-of-Thought Prompt

Chain-of-Thought (CoT) prompting is a technique that guides LLMs to follow a reasoning process when dealing with hard problems. This is done by showing the model a few examples where the step-by-step reasoning is clearly laid out. The model is then expected to follow that "chain of thought" reasoning and get to the correct answer.

[ ]:
```
from langchain.prompts import PromptTemplate
from langchain_community.chains import SimpleChain
# Define the initial prompt as a question
prompt = "What is the sum of 5 and 3?"

# Chain of Thought (CoT) steps explained as text
cot_explanation = """
1. Let's add the first number, 5.
2. Then, add the second number, 3, to the previously obtained sum.
3. The answer is the final sum.
"""

# Combine the prompt and CoT explanation for clarity
prompt_with_cot = f"{prompt}\n\n{cot_explanation}"

# Define a prompt template without any input variables
prompt_template = PromptTemplate(input_variables=[], template=prompt_with_cot)

# Initialize the SimpleChain with the prompt template
```

```
chain = SimpleChain(prompt_template=prompt_template)

# Generate the response using the chain
response = chain.run()

# Print the content of the response (assuming it's text)
print(response.content)
```

5 Tree of thoughts

[14]:
```
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

def tree_of_thoughts(prompt, depth=2, breadth=3, llm=None):
    """
    This function generates a tree of thoughts using an LLM.

    Args:
        prompt: The initial prompt to start the exploration.
        depth: Maximum depth of the tree (number of levels).
        breadth: Number of branches to explore at each level.
        llm: The LLM object to use for generating responses (optional).

    Returns:
        A list of branches, where each branch is a tuple containing the prompt
        and a list of responses generated by the LLM.
    """

    branches = []

    def generate_branches(prompt, current_depth):
        if current_depth == depth:
            return

        # Create an instance of PromptTemplate
        prompt_template = PromptTemplate(input_variables=[], template=prompt)

        # Check if LLM is provided, otherwise raise an error
        if not llm:
            raise ValueError("LLM object is required for generating responses.")

        chain = LLMChain(llm=llm, prompt=prompt_template)
        # Use 'run' method to generate a single response and replicate it for␣
↪breadth
        response = chain.run({})
        responses = [response] * breadth
```

```
        branches.append((prompt, responses))

        for response in responses:
            new_prompt = prompt + " " + response
            generate_branches(new_prompt, current_depth + 1)

    generate_branches(prompt, 0)
    return branches

# Define the initial prompt
initial_prompt = "Find the sum of 5 and 3"

# Assuming you have an LLM object defined as 'my_llm'
thought_tree = tree_of_thoughts(initial_prompt, llm=llm) # Uncomment and replace my_llm with your actual LLM object
thought_tree
```

[14]:
```
[('Find the sum of 5 and 3',
  ['The sum of 5 and 3 is 8.',
   'The sum of 5 and 3 is 8.',
   'The sum of 5 and 3 is 8.']),
 ('Find the sum of 5 and 3 The sum of 5 and 3 is 8.',
  ['The sum of 5 and 3 is 8.',
   'The sum of 5 and 3 is 8.',
   'The sum of 5 and 3 is 8.']),
 ('Find the sum of 5 and 3 The sum of 5 and 3 is 8.',
  ['The sum of 5 and 3 is 8.',
   'The sum of 5 and 3 is 8.',
   'The sum of 5 and 3 is 8.']),
 ('Find the sum of 5 and 3 The sum of 5 and 3 is 8.',
  ['The sum of 5 and 3 is 8.',
   'The sum of 5 and 3 is 8.',
   'The sum of 5 and 3 is 8.'])]
```

6 self consisitency prompt

[22]:
```
import openai
from langchain.llms import OpenAI
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from collections import Counter


def tree_of_thoughts(prompt, depth=2, breadth=3, llm=None):  # Add llm parameter here
```

```
    """
    This function generates a tree of thoughts using an LLM.

    Args:
        prompt: The initial prompt to start the exploration.
        depth: Maximum depth of the tree (number of levels).
        breadth: Number of branches to explore at each level.
        llm: The LLM object to use for generating responses.

    Returns:
        A list of branches, where each branch is a tuple containing the prompt
        and a list of responses generated by the LLM.
    """
    if llm is None:
        raise ValueError("An LLM instance must be provided.")  # Raise error if no LLM is provided

    branches = []

    def generate_branches(prompt, current_depth):
        if current_depth == depth:
            return

        # Pass the prompt as the 'template' argument
        prompt_template = PromptTemplate(input_variables=[], template=prompt)
        chain = LLMChain(llm=llm, prompt=prompt_template)  # Use 'prompt' instead of 'prompt_template' here
        # Use 'run' to generate responses as it returns a list of strings
        responses = [chain.run({}) for _ in range(breadth)]

        branches.append((prompt, responses))

        for response in responses:
            new_prompt = prompt + " " + response  # Directly use response as it's already a string
            generate_branches(new_prompt, current_depth + 1)

    generate_branches(prompt, 0)
    return branches


def evaluate_branches_with_self_consistency(branches):
    """
    This function evaluates branches using self-consistency (most common response).

    Args:
```

```
        branches: A list of branches generated by the tree_of_thoughts function.

    Returns:
        The most common response across all branches (or None if no clear winner).
    """

    all_responses = []
    for prompt, responses in branches:
        # Responses are now strings, so no need for 'get'
        all_responses.extend(responses)

    response_counter = Counter(all_responses)
    most_common_response, count = response_counter.most_common(1)[0]

    # Consider responses with a minimum threshold count for better self-consistency
    threshold = 2  # Minimum occurrence to consider a response "common"
    if count >= threshold:
        return most_common_response
    else:
        return None


# Define the initial prompt (replace with your OpenAI API key)
initial_prompt = "How can we improve urban transportation?"

# Initialize your LLM (replace with your actual LLM object)
llm = OpenAI(temperature=0) # Replace with your actual LLM object

# Generate tree of thoughts
thought_tree = tree_of_thoughts(initial_prompt, llm=llm)  # Pass the LLM object

# Evaluate and select the best branch with self-consistency
best_thought = evaluate_branches_with_self_consistency(thought_tree)
print("Best Thought Path with Self-Consistency:", best_thought)
```

Best Thought Path with Self-Consistency: that takes into account the needs of all residents, including those with disabilities, and considers the impact on the environment.

8. Encourage remote work and flexible schedules: With the rise of remote work, cities can encourage companies to offer flexible work schedules to reduce rush hour traffic and ease the strain on transportation systems.

9. Promote electric and alternative fuel vehicles: Governments can provide incentives for people to switch to electric or alternative fuel vehicles,

reducing air pollution and dependence on fossil fuels.

10. Involve the community: It is important to involve the community in the planning and decision-making process for urban transportation. This can help identify specific needs and concerns and ensure that the solutions implemented are effective and beneficial for all residents.

7 Self Prompting

[ ]:
```
import openai

# Function to generate a prompt based on a previous response
def generate_self_prompt(response):
    # You can customize this logic to fit your specific use case
    prompt = f"Based on the following text, generate a continuation or a related question:\n\n{response}\n\nContinuation or related question:"
    return prompt

# Function to get a response from the GPT model
def get_gpt_response(prompt):
    # Note: Replace 'your_api_key' with your actual OpenAI API key
    openai.api_key = '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'
    response = openai.Completion.create(
        engine="gpt-3.5-turbo",
        prompt=prompt,
        max_tokens=100,
        n=1,
        stop=None,
        temperature=0.7,
    )
    return response.choices[0].text.strip()

# Initial user-provided prompt
initial_prompt = "Once upon a time, in a land far away, there was a village surrounded by mountains."

# Get the first response from the GPT model
response = get_gpt_response(initial_prompt)
print("GPT-3 Response:", response)

# Generate self-prompts and responses iteratively
for _ in range(5):  # Adjust the range for more or fewer iterations
    # Generate a new prompt based on the last response
    new_prompt = generate_self_prompt(response)

    # Get a new response from the GPT model
```

```
    response = get_gpt_response(new_prompt)

    # Print the response
    print("GPT-3 Response:", response)
```