LangChain2 提示工程

提示工程

在Open AI的官方文档 GPT 最佳实践中，给出了提示工程的原则：

1. 写清晰的指示
2. 给模型提供参考（也就是示例）
3. 将复杂任务拆分成子任务
4. 给GPT时间思考
5. 使用外部工具
6. 反复迭代问题

提示的结构

- 指令instruction：告诉模型要干什么，以及提示模型如何使用外部信息、如何处理查询以及如何构造输出。一个常见用例是告诉模型“你是一个有用的XX助手”。
- 上下文context：模型的额外知识来源。一个常见的用例时是把从向量数据库查询到的知识作为上下文传递给模型。
- 输入input：具体的问题或者需要大模型做的具体事情，这个部分和“指令”部分其实也可以合二为一。但是拆分出来成为一个独立的组件，就更加结构化，便于复用模板。这通常是作为变量，在调用模型之前传递给提示模板，以形成具体的提示。
- 输出指示器Output Indicator：标记要生成的文本的开始，LangChain中的代理在构建提示模板时，经常性的会用一个“Thought：”（思考）作为引导词，指示模型开始输出自己的推理（Reasoning）。

LangChain中提供了以下提示模板：

序号	提示模板	含义
1	PromptTemplate	这是最常用的 String 提示模板，我们已经使用过 PromptTemplate 语句导入这个模板。
2	ChatPromptTemplate	常用的 Chat 提示模板，用于组合各种角色的消息模板，传入聊天模型（Chat Model），具体消息模板包括 ChatMessagePromptTemplate、HumanMessagePromptTemplate、AIMessagePromptTemplate 和 SystemMessagePromptTemplate。
3	FewShotPromptTemplate	少样本提示模板，通过示例的展示来“教”模型如何回答。
4	PipelinePrompt	用于把几个提示组合在一起使用。
5	自定义模板	LangChain 还允许你基于其他模板类来定制自己的提示模板。

- PromptTemplate

通过PromptTemplate的from_template方法创建提示模板对象，再利用prompt.format方法将参数实例化

```
from langchain import PromptTemplate
template = """\
你是业务咨询顾问。
你给一个销售{product}的电商公司，起一个好的名字?
"""
prompt = PromptTemplate.from_template(template)
print(prompt.format(product="鲜花"))
```

也可以通过提示模板类的构造函数，在创建模板时指定input_variables:

```
prompt = PromptTemplate(
    input_variables=["product", "market"],
    template="你是业务咨询顾问。对于一个面向{market}市场的，专注于销售{product}的公司，你会推荐哪个名字？"
)
print(prompt.format(product="鲜花", market="高端"))
```

- ChatPromptTemplate

围绕系统、用户、助理三个角色设计。消息必须是消息对象的数组，其中每个对象都有一个角色（系统、用户或助理）和内容。对话首先由系统消息格式化，然后是交替的用户消息和助理消息。

```
# 导入聊天消息类模板
from langchain.prompts import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)
# 模板的构建
template="你是一位专业顾问，负责为专注于{product}的公司起名。"
system_message_prompt = SystemMessagePromptTemplate.from_template(template)
human_template="公司主打产品是{product_detail}。"
human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)
prompt_template = ChatPromptTemplate.from_messages([system_message_prompt, human_message_prompt])

# 格式化提示消息生成提示
prompt = prompt_template.format_prompt(product="鲜花装饰", product_detail="创新的鲜花设计。").to_messages()
```

- FewShot

首先需要创建一些实例，每个示例都是一个字典，其中键是输入变量，值是这些输入变量的值。

```
# 1. 创建一些示例
samples = [
  {
    "flower_type": "玫瑰",
    "occasion": "爱情",
    "ad_copy": "玫瑰，浪漫的象征，是你向心爱的人表达爱意的最佳选择。"
  },
  {
    "flower_type": "康乃馨",
    "occasion": "母亲节",
    "ad_copy": "康乃馨代表着母爱的纯洁与伟大，是母亲节赠送给母亲的完美礼物。"
  },
  {
    "flower_type": "百合",
    "occasion": "庆祝",
    "ad_copy": "百合象征着纯洁与高雅，是你庆祝特殊时刻的理想选择。"
  },
  {
    "flower_type": "向日葵",
    "occasion": "鼓励",
    "ad_copy": "向日葵象征着坚韧和乐观，是你鼓励亲朋好友的最好方式。"
  }
]
```

然后需要创建PromptTemplate提示模板

```
# 2. 创建一个提示模板
from langchain.prompts.prompt import PromptTemplate
template="鲜花类型: {flower_type}\n场合: {occasion}\n文案: {ad_copy}"
prompt_sample = PromptTemplate(input_variables=["flower_type", "occasion",
"ad_copy"],
                               template=template)
print(prompt_sample.format(**samples[0]))
```

之后创建FewShotPromptTemplate 对象，包含了多个示例和一个提示，使用多个示例指导模型对新的prompt生成输出。

```python
# 3. 创建一个FewShotPromptTemplate对象
from langchain.prompts.few_shot import FewShotPromptTemplate
prompt = FewShotPromptTemplate(
    examples=samples,
    example_prompt=prompt_sample,
    suffix="鲜花类型: {flower_type}\n场合: {occasion}",
    input_variables=["flower_type", "occasion"]
)
print(prompt.format(flower_type="野玫瑰", occasion="爱情"))
```

当示例很多时，一次性全给模型会浪费token数，可以通过示例选择器选择向量相似度最高的样本。这里使用Chroma向量数据库

```python
# 5. 使用示例选择器
from langchain.prompts.example_selector import
SemanticSimilarityExampleSelector
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings

# 初始化示例选择器
example_selector = SemanticSimilarityExampleSelector.from_examples(
    samples,
    OpenAIEmbeddings(),
    Chroma,
    k=1
)

# 创建一个使用示例选择器的FewShotPromptTemplate对象
prompt = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=prompt_sample,
    suffix="鲜花类型: {flower_type}\n场合: {occasion}",
    input_variables=["flower_type", "occasion"]
)
print(prompt.format(flower_type="红玫瑰", occasion="爱情"))
```

SemanticSimilarityExampleSelector可以根据语义相似性选择最相关的示例。然后，它创建了一个新的FewShotPromptTemplate对象，这个对象使用了上一步创建的选择器来选择最相关的示例生成提示。

COT

Few-Shot CoT

在prompt的示例中写出推导过程，在算数、常识和推理任务都提高了性能。推理步骤示例：

1. 问题理解：首先理解用户的需求，可以使用提示模板，告诉模型回答的整体流程，例如：“遇到XX问题，我先看自己有没有相关知识，有的话，就提供答案；没有，就调用工具搜索，有了知识后再试图解决”。
2. 信息搜索：模型搜索相关信息
3. 决策制定：基于检索到的信息，通过COT指导模型进行决策的具体流程。示例中应该描述清楚解决问题的具体流程，例如：“遇到生日派对送花的情况，我先考虑用户的需求，然后查看鲜花的库存，最后决定推荐一些玫瑰和百合，因为这些花通常适合生日派对。”最后模型进行决策并生成答案。

Zero-Shot CoT

在prompt中加入“让我们一步步的思考”或者“你是一个很有经验的XX专家”之类的话，要求模型根据事实逐步思考。

COT实战代码：

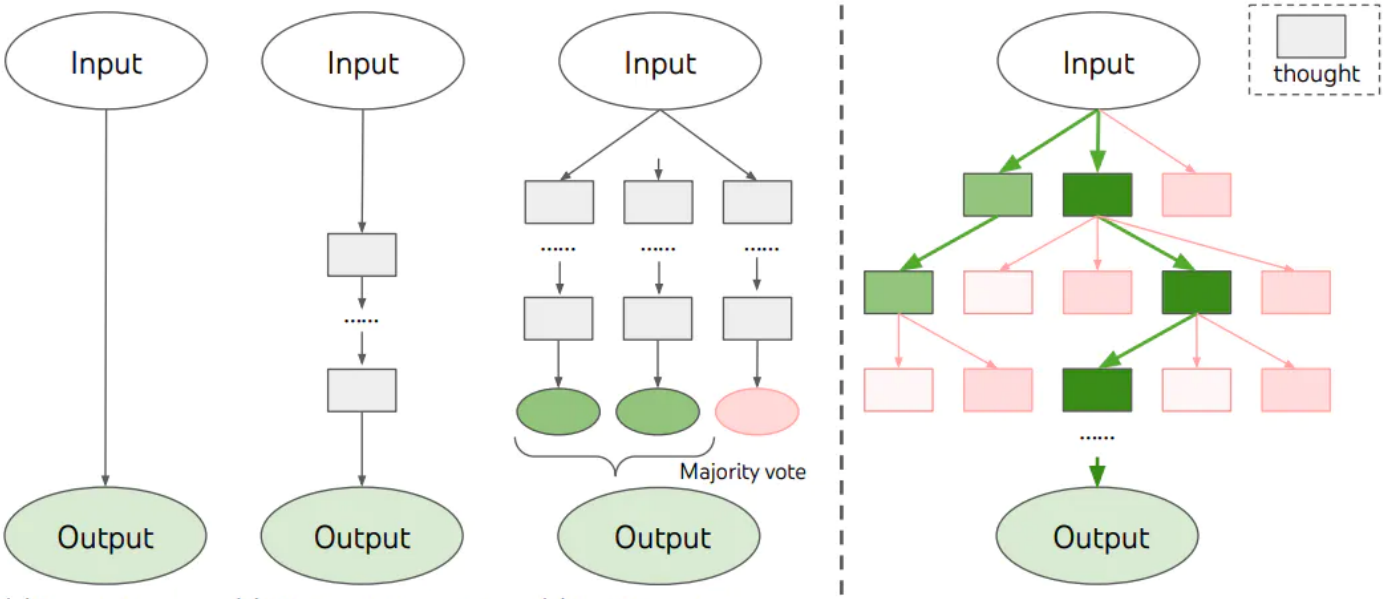
与之前的ChatPromptTemplate类似，额外添加了COT模板，其中包括了AI的角色和目标描述、思考链条以及遵循思考链条的一些示例，显示了AI如何理解问题，并给出建议。

```
# 设置环境变量和API密钥
import os
os.environ["OPENAI_API_KEY"] = '你的OpenAI API Key'

# 创建聊天模型
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(temperature=0)

# 设定 AI 的角色和目标
role_template = "你是一个为花店电商公司工作的AI助手，你的目标是帮助客户根据他们的喜好做出明智的决定"

# CoT 的关键部分，AI 解释推理过程，并加入一些先前的对话示例（Few-Shot Learning）
cot_template = """
作为一个为花店电商公司工作的AI助手，我的目标是帮助客户根据他们的喜好做出明智的决定。

我会按部就班的思考，先理解客户的需求，然后考虑各种鲜花的涵义，最后根据这个需求，给出我的推荐。
同时，我也会向客户解释我这样推荐的原因。

示例 1:
  人类：我想找一种象征爱情的花。
  AI：首先，我理解你正在寻找一种可以象征爱情的花。在许多文化中，红玫瑰被视为爱情的象征，这是因为它们的红色通常与热情和浓烈的感情联系在一起。因此，考虑到这一点，我会推荐红玫瑰。红玫瑰不仅能够象征爱情，同时也可以传达出强烈的感情，这是你在寻找的。
```

```
示例 2:
   人类：我想要一些独特和奇特的花。
   AI：从你的需求中，我理解你想要的是独一无二和引人注目的花朵。兰花是一种非常独特并且颜色鲜艳的花，它们在世界上的许多地方都被视为奢侈品和美的象征。因此，我建议你考虑兰花。选择兰花可以满足你对独特和奇特的要求，而且，兰花的美丽和它们所代表的力量和奢侈也可能会吸引你。
"""
from langchain.prompts import ChatPromptTemplate, HumanMessagePromptTemplate, SystemMessagePromptTemplate
system_prompt_role = SystemMessagePromptTemplate.from_template(role_template)
system_prompt_cot = SystemMessagePromptTemplate.from_template(cot_template)

# 用户的询问
human_template = "{human_input}"
human_prompt = HumanMessagePromptTemplate.from_template(human_template)

# 将以上所有信息结合为一个聊天提示
chat_prompt = ChatPromptTemplate.from_messages([system_prompt_role, system_prompt_cot, human_prompt])

prompt = chat_prompt.format_prompt(human_input="我想为我的女朋友购买一些花。她喜欢粉色和紫色。你有什么建议吗?").to_messages()

# 接收用户的询问，返回回答结果
response = llm(prompt)
print(response)
```

Tree of Thought

在需要多步骤推理的任务中，引导语言模型搜索一棵由连贯的语言序列（解决问题的中间步骤）组成的思维树，而不是简单地生成一个答案。ToT框架的核心思想是：让模型生成和评估其思维的能力，并将其与搜索算法（如广度优先搜索和深度优先搜索）结合起来，进行系统性地探索和验证。对于每个任务，将其分解为多个步骤，为每个步骤提出多个方案，在多条思维路径中搜寻最优的方案。



(a) Input-Output Prompting (IO)

(c) Chain of Thought Prompting (CoT)

(c) Self Consistency with CoT (CoT-SC)

(d) Tree of Thoughts (ToT)