Evaluate and optimize the build part

Note: The source code corresponding to this article is in the Github open source project LLM Universe . Readers are welcome to download and run it. You are welcome to give us a Star~

In the previous chapter, we talked about how to evaluate the overall performance of a large model application based on the RAG framework. By constructing a validation set in a targeted manner, we can use a variety of methods to evaluate system performance from multiple dimensions. However, the purpose of the evaluation is to better optimize the application effect. To optimize the application performance, we need to combine the evaluation results, split the Bad Cases evaluated, and evaluate and optimize each part separately.

RAG stands for Retrieval Enhanced Generation, so it has two core parts: the retrieval part and the generation part. The core function of the retrieval part is to ensure that the system can find the corresponding answer fragment according to the user's query, while the core function of the generation part is to ensure that after the system obtains the correct answer fragment, it can give full play to the large model capability to generate a correct answer that meets the user's requirements.

To optimize a large model application, we often need to start from both parts at the same time, evaluate the performance of the retrieval part and the optimization part respectively, find out the bad cases and optimize the performance in a targeted manner. As for the generation part, in the case of a limited large model base, we often optimize the generated answers by optimizing Prompt Engineering. In this chapter, we will first combine the large model application example we just built - Personal Knowledge Base Assistant, to explain how to evaluate and analyze the performance of the generation part, find out the bad cases in a targeted manner, and optimize the generation part by optimizing Prompt Engineering.

Before we start, let's load our vector database and retrieval chain:

python

```
import sys
sys.path.append("../C3 搭建知识库") # 将父目录放入系统路径中

# 使用智谱 Embedding API，注意，需要将上一章实现的封装代码下载到本地
from zhipuai_embedding import ZhipuAIEmbeddings
```

```python
from langchain.vectorstores.chroma import Chroma
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv, find_dotenv
import os

_ = load_dotenv(find_dotenv())    # read local .env file
zhipuai_api_key = os.environ['ZHIPUAI_API_KEY']
OPENAI_API_KEY = os.environ["OPENAI_API_KEY"]

# 定义 Embeddings
embedding = ZhipuAIEmbeddings()

# 向量数据库持久化路径
persist_directory = '../../data_base/vector_db/chroma'

# 加载数据库
vectordb = Chroma(
    persist_directory=persist_directory,  # 允许我们将persist_directory目录
    embedding_function=embedding
)

# 使用 OpenAI GPT-3.5 模型
llm = ChatOpenAI(model_name = "gpt-3.5-turbo", temperature = 0)

os.environ['HTTPS_PROXY'] = 'http://127.0.0.1:7890'
os.environ["HTTP_PROXY"] = 'http://127.0.0.1:7890'
```

We first use the initialized Prompt to create a template-based search chain:

python

```python
from langchain.prompts import PromptTemplate
from langchain.chains import RetrievalQA


template_v1 = """使用以下上下文来回答最后的问题。如果你不知道答案，就说你不知道，不要
案。最多使用三句话。尽量使答案简明扼要。总是在回答的最后说“谢谢你的提问！”。
{context}
问题: {question}
"""

QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context","question"],
                                 template=template_v1)
```

```
qa_chain = RetrievalQA.from_chain_type(llm,
                                       retriever=vectordb.as_retriever(),
                                       return_source_documents=True,
                                       chain_type_kwargs={"prompt":QA_CHA
```

Test the effect first:

python

```
question = "什么是南瓜书"
result = qa_chain({"query": question})
print(result["result"])
```

markup

```
南瓜书是对《机器学习》（西瓜书）中比较难理解的公式进行解析和补充推导细节的书籍。南瓜书的
```

1. Improve the quality of intuitive answers

There are many ways to find Bad Cases. The most intuitive and simplest way is to evaluate the quality of intuitive answers and determine where there are deficiencies based on the original data. For example, we can construct a Bad Case for the above test:

markup

```
问题：什么是南瓜书
初始回答：南瓜书是对《机器学习》（西瓜书）中难以理解的公式进行解析和补充推导细节的一本书
存在不足：回答太简略，需要回答更具体；谢谢你的提问感觉比较死板，可以去掉
```

We then modify the Prompt template specifically, adding a requirement for a specific answer and removing the "Thank you for your question" part:

python

```
template_v2 = """使用以下上下文来回答最后的问题。如果你不知道答案，就说你不知道，不要
案。你应该使答案尽可能详细具体，但不要偏题。如果答案比较长，请酌情进行分段，以提高答案的
{context}
```

```
问题：{question}
有用的回答："""

QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context","question"],
                                 template=template_v2)
qa_chain = RetrievalQA.from_chain_type(llm,
                                       retriever=vectordb.as_retriever(),
                                       return_source_documents=True,
                                       chain_type_kwargs={"prompt":QA_CHA

question = "什么是南瓜书"
result = qa_chain({"query": question})
print(result["result"])
```

markup

```
南瓜书是一本针对周志华老师的《机器学习》（西瓜书）的补充解析书籍。它旨在对西瓜书中比较难
```

As you can see, the improved v2 version can give more specific and detailed answers, solving the previous problem. But we can think further: will requiring the model to give specific and detailed answers lead to unclear and unfocused answers to some key points? We test the following questions:

python

```
question = "使用大模型时，构造 Prompt 的原则有哪些"
result = qa_chain({"query": question})
print(result["result"])
```

markup

```
在使用大型语言模型时，构造Prompt的原则主要包括编写清晰、具体的指令和给予模型充足的思考时

其次，给予语言模型充足的推理时间也是至关重要的。类似于人类解决问题时需要思考的时间，模型

通过遵循这两个原则，设计优化的Prompt可以帮助语言模型充分发挥潜力，完成复杂的推理和生成任
```

As you can see, the model's answer to our question about the LLM course is indeed detailed and specific, and it also fully refers to the course content. However, the answer begins with words

such as "first" and "second", and the overall answer is divided into 4 paragraphs, which makes the answer unclear and difficult to read. Therefore, we construct the following Bad Case:

markup

```
问题：使用大模型时，构造 Prompt 的原则有哪些
初始回答：略
存在不足：没有重点，模糊不清
```

For this bad case, we can improve the prompt and require it to mark the points of the answers with several points to make the answers clear and specific:

python

```
template_v3 = """使用以下上下文来回答最后的问题。如果你不知道答案，就说你不知道，不要
案。你应该使答案尽可能详细具体，但不要偏题。如果答案比较长，请酌情进行分段，以提高答案的
如果答案有几点，你应该分点标号回答，让答案清晰具体
{context}
问题: {question}
有用的回答:"""

QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context","question"],
                                 template=template_v3)
qa_chain = RetrievalQA.from_chain_type(llm,
                                       retriever=vectordb.as_retriever(),
                                       return_source_documents=True,
                                       chain_type_kwargs={"prompt":QA_CHA

question = "使用大模型时，构造 Prompt 的原则有哪些"
result = qa_chain({"query": question})
print(result["result"])
```

markup

```
1. 编写清晰、具体的指令是构造 Prompt 的第一原则。Prompt需要明确表达需求，提供充足上下

2. 给予模型充足思考时间是构造Prompt的第二原则。语言模型需要时间推理和解决复杂问题，匆忙

3. 在设计Prompt时，要指定完成任务所需的步骤。通过给定一个复杂任务，给出完成任务的一系列

4. 迭代优化是构造Prompt的常用策略。通过不断尝试、分析结果、改进Prompt的过程，逐步逼近

5. 添加表格描述是优化Prompt的一种方法。要求模型抽取信息并组织成表格，指定表格的列、表名
```

```
总之，构造Prompt的原则包括清晰具体的指令、给予模型充足思考时间、指定完成任务所需的步骤、
```

There are many ways to improve the quality of answers. The key is to think around specific business, find out the unsatisfactory points in the initial answers, and make targeted improvements. I will not go into details here.

2. Indicate the source of knowledge to increase credibility

Due to the hallucination problem of large models, we sometimes suspect that the model's answer is not derived from the existing knowledge base content. This is particularly important in some scenarios where authenticity needs to be guaranteed, such as:

python

```
question = "强化学习的定义是什么"
result = qa_chain({"query": question})
print(result["result"])
```

markup

```
强化学习是一种机器学习方法，旨在让智能体通过与环境的交互学习如何做出一系列好的决策。在强
```

We can ask the model to indicate the source of knowledge when generating answers. This can prevent the model from fabricating knowledge that does not exist in the given data. At the same time, it can also improve our credibility in the answers generated by the model:

python

```
template_v4 = """使用以下上下文来回答最后的问题。如果你不知道答案，就说你不知道，不要
案。你应该使答案尽可能详细具体，但不要偏题。如果答案比较长，请酌情进行分段，以提高答案的
如果答案有几点，你应该分点标号回答，让答案清晰具体。
请你附上回答的来源原文，以保证回答的正确性。
{context}
问题: {question}
有用的回答:"""

QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context","question"],
```

```
                                        template=template_v4)
qa_chain = RetrievalQA.from_chain_type(llm,
                                       retriever=vectordb.as_retriever(),
                                       return_source_documents=True,
                                       chain_type_kwargs={"prompt":QA_CHA

question = "强化学习的定义是什么"
result = qa_chain({"query": question})
print(result["result"])
```

markup

```
强化学习是一种机器学习方法，旨在让智能体通过与环境的交互学习如何做出一系列好的决策。在这
```

However, attaching the original source often leads to an increase in context and a decrease in response speed. We need to consider whether to require the original text to be attached based on the business scenario.

3. Construct a chain of thoughts

Large models can often understand and execute instructions well, but the models themselves still have some limitations, such as hallucinations of large models, inability to understand more complex instructions, inability to execute complex steps, etc. We can minimize its limitations by constructing thought chains and structuring Prompt into a series of steps. For example, we can construct a two-step thought chain and require the model to reflect in the second step to eliminate the hallucination problem of large models as much as possible.

First we have a Bad Case:

markup

```
问题：我们应该如何去构造一个 LLM 项目
初始回答：略
存在不足：事实上，知识库中中关于如何构造LLM项目的内容是使用 LLM API 去搭建一个应用，核
```

python

```
question = "我们应该如何去构造一个LLM项目"
result = qa_chain({"query": question})
```

```
print(result["result"])
```

markup

```
构建一个LLM项目需要考虑以下几个步骤：

1. 确定项目目标和需求：首先要明确你的项目是为了解决什么问题或实现什么目标，确定需要使用

2. 收集和准备数据：根据项目需求，收集和准备适合的数据集，确保数据的质量和多样性，以提高

3. 设计Prompt和指令微调：根据项目需求设计合适的Prompt，确保指令清晰明确，可以引导LLM

4. 进行模型训练和微调：使用基础LLM或指令微调LLM对数据进行训练和微调，以提高模型在特定任

5. 测试和评估模型：在训练完成后，对模型进行测试和评估，检查其在不同场景下的表现和效果，

6. 部署和应用模型：将训练好的LLM模型部署到实际应用中，确保其能够正常运行并实现预期的效

来源：根据提供的上下文内容进行总结。
```

To this end, we can optimize the prompt and turn the previous prompt into two steps, requiring the model to reflect in the second step:

python

```
template_v4 = """
请你依次执行以下步骤：
① 使用以下上下文来回答最后的问题。如果你不知道答案，就说你不知道，不要试图编造答案。
你应该使答案尽可能详细具体，但不要偏题。如果答案比较长，请酌情进行分段，以提高答案的阅读
如果答案有几点，你应该分点标号回答，让答案清晰具体。
上下文：
{context}
问题：
{question}
有用的回答：
② 基于提供的上下文，反思回答中有没有不正确或不是基于上下文得到的内容，如果有，回答你不知
确保你执行了每一个步骤，不要跳过任意一个步骤。
"""

QA_CHAIN_PROMPT = PromptTemplate(input_variables=["context","question"],
                                 template=template_v4)
qa_chain = RetrievalQA.from_chain_type(llm,
```

```
                                    retriever=vectordb.as_retriever(),
                                    return_source_documents=True,
                                    chain_type_kwargs={"prompt":QA_CHA
```

```python
question = "我们应该如何去构造一个LLM项目"
result = qa_chain({"query": question})
print(result["result"])
```

markup

根据上下文中提供的信息，构造一个LLM项目需要考虑以下几个步骤：

1. 确定项目目标：首先要明确你的项目目标是什么，是要进行文本摘要、情感分析、实体提取还是

2. 设计Prompt：根据项目目标设计合适的Prompt，Prompt应该清晰明确，指导LLM生成符合预期

3. 调用API接口：根据设计好的Prompt，通过编程调用LLM的API接口来生成结果。确保API接口的

4. 分析结果：获取LLM生成的结果后，进行结果分析，确保结果符合项目目标和预期。如果结果不

5. 优化和改进：根据分析结果的反馈，不断优化和改进LLM项目，提高项目的效率和准确性。可以

通过以上步骤，可以构建一个有效的LLM项目，利用LLM的强大功能来实现文本摘要、情感分析、实

It can be seen that after asking the model to reflect on itself, the model repaired its illusion and gave the correct answer. We can also complete more functions by constructing thought chains, which will not be described here. Readers are welcome to try.

4. Add a command parsing

We often face a requirement that we need the model to output in the format we specify. However, since we use Prompt Template to fill in user questions, the format requirements in user questions are often ignored, for example:

python

```python
question = "LLM的分类是什么？给我返回一个 Python List"
result = qa_chain({"query": question})
print(result["result"])
```

```markup
根据上下文提供的信息，LLM（Large Language Model）的分类可以分为两种类型，即基础LLM和

根据上下文，可以返回一个Python List，其中包含LLM的两种分类：["基础LLM"， "指令微调LL
```

As you can see, although we require the model to return a Python List, this output requirement is wrapped in the Template and ignored by the model. To address this issue, we can construct a Bad Case:

```markup
问题：LLM的分类是什么？给我返回一个 Python List
初始回答：根据提供的上下文，LLM的分类可以分为基础LLM和指令微调LLM。
存在不足：没有按照指令中的要求输出
```

One solution to this problem is to add a layer of LLM before our retrieval LLM to implement instruction parsing, and separate the format requirements of user questions from the content of the questions. This idea is actually the prototype of the currently popular Agent mechanism, that is, for user instructions, set up an LLM (ie Agent) to understand the instructions, determine what tools need to be executed for the instructions, and then call the tools that need to be executed in a targeted manner. Each tool can be an LLM based on different Prompt Engineering, or it can be a database, API, etc. In fact, there is an Agent mechanism designed in LangChain, but we will not go into details in this tutorial. Here we only simply implement this function based on OpenAI's native interface:

```python
# 使用第二章讲过的 OpenAI 原生接口

from openai import OpenAI

client = OpenAI(
    # This is the default and can be omitted
    api_key=os.environ.get("OPENAI_API_KEY"),
)


def gen_gpt_messages(prompt):
    '''
    构造 GPT 模型请求参数 messages

    请求参数：
```

```
        prompt: 对应的用户提示词
    '''
    messages = [{"role": "user", "content": prompt}]
    return messages


def get_completion(prompt, model="gpt-3.5-turbo", temperature = 0):
    '''
    获取 GPT 模型调用结果

    请求参数:
        prompt: 对应的提示词
        model: 调用的模型，默认为 gpt-3.5-turbo，也可以按需选择 gpt-4 等其他模型
        temperature: 模型输出的温度系数，控制输出的随机程度，取值范围是 0~2。温度系
    '''
    response = client.chat.completions.create(
        model=model,
        messages=gen_gpt_messages(prompt),
        temperature=temperature,
    )
    if len(response.choices) > 0:
        return response.choices[0].message.content
    return "generate answer error"

prompt_input = '''
请判断以下问题中是否包含对输出的格式要求，并按以下要求输出：
请返回给我一个可解析的Python列表，列表第一个元素是对输出的格式要求，应该是一个指令；第二
如果没有格式要求，请将第一个元素置为空
需要判断的问题：
~~~
{}
~~~
不要输出任何其他内容或格式，确保返回结果可解析。
'''
```

Let's test the LLM's ability to decompose the format requirements:

python

```
response = get_completion(prompt_input.format(question))
response
```

markup

```
'```\n["给我返回一个 Python List", "LLM的分类是什么？"]\n```'
```

As you can see, through the above prompt, LLM can well implement the output format parsing. Next, we can set up another LLM to parse the output content according to the output format requirements:

python

```
prompt_output = '''
请根据回答文本和输出格式要求，按照给定的格式要求对问题做出回答
需要回答的问题：
~~~
{}
~~~
回答文本：
~~~
{}
~~~
输出格式要求：
~~~
{}
~~~
'''
```

We can then chain the two LLMs together with the search chain:

python

```
question = 'LLM的分类是什么？给我返回一个 Python List'
# 首先将格式要求与问题拆分
input_lst_s = get_completion(prompt_input.format(question))
# 找到拆分之后列表的起始和结束字符
start_loc = input_lst_s.find('[')
end_loc = input_lst_s.find(']')
rule, new_question = eval(input_lst_s[start_loc:end_loc+1])
# 接着使用拆分后的问题调用检索链
result = qa_chain({"query": new_question})
result_context = result["result"]
# 接着调用输出格式解析
```

```
response = get_completion(prompt_output.format(new_question, result_conte
response
```

```
markup
"['基础LLM', '指令微调LLM']"
```

As you can see, after the above steps, we have successfully achieved the output format limitation. Of course, in the above code, the core is to introduce the Agent idea. In fact, whether it is the Agent mechanism or the Parser mechanism (that is, limiting the output format), LangChain provides a mature tool chain for use. Interested readers are welcome to explore it in depth, so I will not explain it here.

Through the ideas explained above, combined with actual business situations, we can continuously discover bad cases and optimize prompts in a targeted manner, thereby improving the performance of the generation part. However, the premise of the above optimization is that the retrieval part can retrieve the correct answer fragment, that is, the accuracy and recall of the retrieval are as high as possible. So, how can we evaluate and optimize the performance of the retrieval part? We will explore this issue in depth in the next chapter.

Note: The source code corresponding to this article is in the Github open source project LLM Universe . Readers are welcome to download and run it. You are welcome to give us a Star~

Previous chapter
1. How to evaluate LLM applications

Next Chapter
3. Evaluate and optimize the search part