3683 words | 10 minutes

4.2 Building a retrieval question-answer chain

The source files involved in the text can be obtained from the following path:

> - C3 builds a knowledge base

In `C3 搭建数据库` the previous section, we have introduced how to build a vector knowledge base based on your own local knowledge documents. In the following content, we will use the built vector database to recall query questions, and combine the recall results with the query to build prompts, which will be input into the large model for question and answering.

1. Load the vector database

First, we load the vector database that we built in the previous chapter. Note that you need to use the same Emedding as when building it.

python

```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

from langchain.vectorstores.chroma import Chroma
```

Load your API_KEY from the environment variable

python

```python
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']
```

Load the vector database, which contains the embeddings of multiple documents under ../../data_base/knowledge_db

python

```python
# 定义 Embeddings
embedding = ZhipuAIEmbeddings()

# 向量数据库持久化路径
persist_directory = '../C3 搭建知识库/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
)
```

python

```python
print(f"向量库中存储的数量：{vectordb._collection.count()}")
```

markup

```
向量库中存储的数量：20
```

We can test the loaded vector database and use a question query to perform vector retrieval. The following code will search the vector database based on similarity and return the top k most similar documents.

> ⚠️ **Before using similarity search, make sure you have installed the OpenAI open source fast word segmentation tool tiktoken package:** `pip install tiktoken`

python

```python
question = "什么是prompt engineering?"
docs = vectordb.similarity_search(question,k=3)
print(f"检索到的内容数：{len(docs)}")
```

markup

```
检索到的内容数：3
```

Print the retrieved content

python

```python
for i, doc in enumerate(docs):
    print(f"检索到的第{i}个内容: \n {doc.page_content}", end="\n------------
```

markup

```
检索到的第0个内容:
 相反，我们应通过 Prompt 指引语言模型进行深入思考。可以要求其先列出对问题的各种看法，说

综上所述，给予语言模型充足的推理时间，是 Prompt Engineering 中一个非常重要的设计原则

2.1 指定完成任务所需的步骤

接下来我们将通过给定一个复杂任务，给出完成该任务的一系列步骤，来展示这一策略的效果。

首先我们描述了杰克和吉尔的故事，并给出提示词执行以下操作：首先，用一句话概括三个反引号限
-----------------------------------------------------
检索到的第1个内容:
 第二章 提示原则

如何去使用 Prompt，以充分发挥 LLM 的性能？首先我们需要知道设计 Prompt 的原则，它们是

首先，Prompt 需要清晰明确地表达需求，提供充足上下文，使语言模型准确理解我们的意图，就像

其次，让语言模型有充足时间推理也极为关键。就像人类解题一样，匆忙得出的结论多有失误。因此

如果 Prompt 在这两点上都作了优化，语言模型就能够尽可能发挥潜力，完成复杂的推理和生成任

一、原则一 编写清晰、具体的指令
-----------------------------------------------------
检索到的第2个内容:
 一、原则一 编写清晰、具体的指令

亲爱的读者，在与语言模型交互时，您需要牢记一点:以清晰、具体的方式表达您的需求。假设您面前

并不是说 Prompt 就必须非常短小简洁。事实上，在许多情况下，更长、更复杂的 Prompt 反而会
```

```
所以，记住用清晰、详尽的语言表达 Prompt，就像在给外星人讲解人类世界一样，“Adding mor

从该原则出发，我们提供几个设计 Prompt 的技巧。

1.1 使用分隔符清晰地表示输入的不同部分
-----------------------------------------------------
```

2. Create an LLM

Here, we call OpenAI's API to create an LLM. Of course, you can also use other LLM APIs to create

```python
import os
OPENAI_API_KEY = os.environ["OPENAI_API_KEY"]
```

```python
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model_name = "gpt-3.5-turbo", temperature = 0)

llm.invoke("请你自我介绍一下自己！")
```

```markup
AIMessage(content='你好，我是一个智能助手，专门为用户提供各种服务和帮助。我可以回答问
```

3. Build a retrieval question-answer chain

```python
from langchain.prompts import PromptTemplate

template = """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
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)
```

Create another template-based search chain:

```python
from langchain.chains import RetrievalQA

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

The method RetrievalQA.from_chain_type() that creates a retrieval QA chain has the following parameters:

- llm: specifies the LLM to be used
- Specify chain type: RetrievalQA.from_chain_type(chain_type="map_reduce"), or use load_qa_chain() method to specify chain type.
- Custom prompt: By specifying the chain_type_kwargs parameter in the RetrievalQA.from_chain_type() method, the parameter: chain_type_kwargs = {"prompt": PROMPT}
- Return source documents: by specifying the return_source_documents=True parameter in the RetrievalQA.from_chain_type() method; you can also use the RetrievalQAWithSourceChain() method to return a reference to the source document (coordinates or primary keys, indexes)

4. Test the effect of retrieval question and answer chain

```python
question_1 = "什么是南瓜书？"
question_2 = "王阳明是谁？"
```

4.1 Prompt effect based on recall results and query

python

```python
result = qa_chain({"query": question_1})
print("大模型+知识库后回答 question_1 的结果：")
print(result["result"])
```

markup

```
d:\Miniconda\miniconda3\envs\llm2\lib\site-packages\langchain_core\_api\d
  warn_deprecated(


大模型+知识库后回答 question_1 的结果：
抱歉，我不知道南瓜书是什么。谢谢你的提问！
```

python

```python
result = qa_chain({"query": question_2})
print("大模型+知识库后回答 question_2 的结果：")
print(result["result"])
```

markup

```
大模型+知识库后回答 question_2 的结果：
我不知道王阳明是谁。

谢谢你的提问！
```

4.2 The effect of the large model answering itself

python

```python
prompt_template = """请回答下列问题：
                            {}""".format(question_1)

### 基于大模型的问答
llm.predict(prompt_template)
```

markup

```
d:\Miniconda\miniconda3\envs\llm2\lib\site-packages\langchain_core\_api\d
  warn_deprecated(
```

```
'南瓜书是指一种关于南瓜的书籍，通常是指介绍南瓜的种植、养护、烹饪等方面知识的书籍。南瓜
```

python

```
prompt_template = """请回答下列问题:
                            {}""".format(question_2)

### 基于大模型的问答
llm.predict(prompt_template)
```

markup

```
'王阳明（1472年-1529年），字宪，号阳明，浙江绍兴人，明代著名的哲学家、军事家、教育家、
```

> ⭐ **Through the above two questions, we found that LLM did not answer some recent knowledge and non-common sense professional questions very well. With our local knowledge, we can help LLM to give better answers. In addition, it also helps to alleviate the "illusion" problem of large models.**

5. Added the memory function of historical conversations

Now that we have achieved this by uploading local knowledge documents and then saving them to the vector knowledge base, by combining the query question with the recall results of the vector knowledge base and inputting it into the LLM, we get a much better result than asking the LLM to answer directly. When interacting with language models, you may have noticed a key problem - **they don't remember your previous communication content** . This poses a big challenge when we build some applications (such as chatbots), making the conversation seem to lack real continuity. How to solve this problem?

1. Memory

In this section we will introduce the storage module in LangChain, that is, how to embed previous conversations into the language model, giving it the ability to have continuous conversations. We will use `ConversationBufferMemory` it to save a list of chat message histories, which will be passed to the chatbot along with the questions when answering them, thus adding them to the context.

```python
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(
    memory_key="chat_history",  # 与 prompt 的输入变量保持一致。
    return_messages=True  # 将以消息列表的形式返回聊天记录，而不是单个字符串
)
```

For more information about the use of Memory, including retaining a specified number of conversation rounds, saving a specified number of tokens, saving summaries of historical conversations, etc., please refer to the relevant documentation of the Memory section of langchain.

2. ConversationalRetrievalChain

The Conversational Retrieval Chain adds the ability to process conversation history based on the retrieval QA chain.

Its workflow is:

1. Combine the previous conversation with the new question to generate a complete query statement.
2. Search the vector database for relevant documents for the query.
3. After getting the results, store all answers in the conversation memory area.
4. Users can view the complete conversation flow in the UI.

This chaining approach puts new questions in the context of previous conversations for retrieval, and can handle queries that rely on historical information. It also keeps all information in the conversation memory for easy tracking.

Next, let's test the effect of this conversation retrieval chain:

Use the vector database and LLM from the previous section! Start by asking a history-free question, "Can I learn something about hint engineering?" and see the answer.

python

```python
from langchain.chains import ConversationalRetrievalChain

retriever=vectordb.as_retriever()

qa = ConversationalRetrievalChain.from_llm(
    llm,
    retriever=retriever,
    memory=memory
)
question = "我可以学习到关于提示工程的知识吗？"
result = qa({"question": question})
print(result['answer'])
```

markup

```
是的，您可以学习到关于提示工程的知识。本模块内容基于吴恩达老师的《Prompt Engineering
```

Then based on the answer, proceed to the next question: "Why does this course need to teach this knowledge?"

python

```python
question = "为什么这门课需要教这方面的知识？"
result = qa({"question": question})
print(result['answer'])
```

markup

```
这门课程需要教授关于Prompt Engineering的知识，主要是为了帮助开发者更好地使用大型语言
```

As you can see, LLM accurately judges this aspect of knowledge, referring to the content as reinforcement learning knowledge, that is, we have successfully passed on historical information to it. This ability to continuously learn and associate previous and subsequent questions can greatly enhance the continuity and intelligence level of the question-answering system.

The source file acquisition path involved above is:

C3 builds a knowledge base

Previous chapter
1. LLM connected to LangChain

Next Chapter
3. Deploy the Knowledge Base Assistant