10.

## 10.1 集成的背景与LLM集成

学习任何新的技术框架或工具，往往需要对其背后的原理和历史背景有所了解，这样可以更好地掌握它的应用方式和最佳实践。在探讨为什么学习LangChain的集成项目之前，我们先看看Apache Camel和Spring Cloud Data Flow的集成技术历史与现状。Apache Camel和Spring Cloud Data Flow都是集成领域的佼佼者，它们各自拥有丰富的生态系统和社区支持。这两个框架已经解决了很多集成的常见问题，提供了大量的最佳实践。LangChain作为一个新的集成框架，其设计思想和实现方式无疑是受到这两个框架的启发。学习它们可以帮助我们更好地理解LangChain的设计哲学和技术选型。

当我们谈及集成，首先要了解的是：为什么Apache Camel会成为这个领域的佼佼者呢？答案很简单。Camel针对不同的协议和数据类型，提供了特定的API实现。这是因为，集成的真正挑战在于处理来自不同系统、协议和数据格式的信息。Camel已经解决了这个问题。它目前支持超过80个协议和数据类型，包括但不限于RESTful服务、消息队列和数据库。并且，得益于其模块化和可扩展的架构，Camel为开发者带来了巨大的便利性和灵活性。

而当我们进入到云计算时代，数据的整合和流动变得更加关键。Spring Cloud Data Flow（SCDF）应运而生，成为了云原生环境中的数据流处理利器。它不仅仅是一个数据流处理工具，更是一个完整的微服务集成框架。SCDF允许开发者轻松创建、部署和监控数据流处理管道，更重要的是，它支持与多种云平台集成，为云原生应用的开发提供了极大的灵活性。

如今面对大语言模型开发(LLM应用开发)的复杂性，LLM应用开发和部署需要一个专为其量身打造的集成框架。LangChain正是这样的解决方案。

LangChain继承了Apache Camel和SCDF的经验，结合LLM的特性，它提供了一套完整的工具和技术，简化了LLM应用的开发、部署和管理。

### 10.1.1 LangChain的集成生态

集成的真正挑战在于处理来自不同系统、协议和数据格式的信息，而LangChain的Integrations库正是基于这一核心思想而构建的。以下是其核心分类及相关描述，集成项目的数量在2023年8月统计：

- Callbacks：LangChain理解到，在某些特定事件触发时，开发者可能需要与其他系统或服务互动。正如Camel针对不同协议和数据类型提供了特定的API实现，Callbacks功能也为开发者创造了类似的桥梁，已实现六个此类项目的集成。

- 聊天模型包装器：面对多样的对话场景，LangChain提供了10种聊天模型来满足从简单问答到高级交互的需求。这如同Camel为不同的协议和数据类型提供了解决方案。

- Document loaders：文档加载与处理是集成中的基础工作。LangChain提供了127种文档加载工具，确保了各种应用场景的需求都能被满足。

- Document transformers：针对文档的处理和转换，LangChain提供了7种转换器，这可以看作是LangChain为多种数据格式提供支持的一个延伸。

- LLM模型包装器：针对LLM应用开发，LangChain准备了57种LLM模型，满足了与不同协议和数据格式交互的需要。

- Memory：LangChain提供了12种记忆存储解决方案，满足了各种持久化需求。

- Retrievers：信息检索是集成的关键，LangChain为此准备了22种集成工具，无论是本地文档还是网络上的信息都能被有效地检索。

-嵌入模型包装器：文本的向量化处理是与各种协议和数据类型交互的关键。LangChain为此提供了31种文本嵌入模型包装器。

- Agent toolkit：为了帮助开发者创建智能代理，LangChain提供了21种集成工具套件，确保与各种系统和服务的交互能够流畅进行。

- Tools：为了满足开发、测试和优化的多样需求，LangChain提供了37种集成工具。

- Vector store：机器学习和深度学习需要专门的向量数据存储，LangChain为此提供了45种解决方案。

- Grouped by provider：展现了LangChain与各大供应商和平台的集成实力，这也反映了它在处理不同系统、协议和数据格式信息方面的广泛适应性。

LangChain虽然提供了广泛的集成库，但这也为开发者带来了一系列的挑战。以下是开发者在使用LangChain的集成库时可能会遇到的问题：

（1）选择的困惑：面对多种相似的工具，开发者可能会在选择上犹豫不决，不知道哪种更适合他们的需求。（2）学习曲线：不同的工具有其特有的功能和操作方式，这意味着开发者需要为每一种工具投入学习的时间。（3）维护的挑战：随着技术的迅速发展，一些工具可能会过时，需要定期更新或替换。（4）性能与兼容性：使用不同的集成工具时，可能会出现性能瓶颈或兼容性问题，这有可能影响整个LLM应用的稳定性和效率。

为了帮助开发者更有效地选择和使用集成工具，我们选择了以下几类在开发中比较受关注的集成类型，并为每个类型选取了几个典型的集成项目，结合代码示例进行说明。

## 10.2 LLM集成指南

LLM集成是实现了各个模型平台的LLM模型包装器，我们主要介绍HuggingFace 和AzureOpenAI。Huggingface为我们提供了一个平台，能够及时追踪当前热门的新型语言模型。它为BERT、XLNet、GPT等多种模型提供了统一且高效的编码实践。更为出色的是，它设有一个模型库，其中涵盖了众多常用的预训练模型以及为各种任务进行微调（Fine-tuning）的模型，使得模型的下载变得简单快捷。HuggingFace有很多模型支持本地下载和部署，为开发者提供了多样的功能和模块，能够快速响应各种需求，大大提高开发效率。

另外，若你希望利用免费的模型，可以前往Hugging Face平台，比如搜索google/flan-t5-xl 来获取免费的模型。

AzureOpenAI是Langchain与Azure之间的集成项目，旨在让开发者能够在Langchain框架中，轻松地通过模型包装器在Azure平台上调用OpenAI的GPT系列模型。此外，Azure OpenAI不仅是Azure平台上的标志性工具，它也是LLM应用中的核心组件，确保与Azure平台之间的稳固协同作业，从而为LLM应用提供坚实的基础。

### 10.2.1 Azure OpenAI 集成

AzureOpenAI的Langchain实现的在Azure平台调用OpenAI能力的LLM模型包装器。在LLM应用中起到了至关重要的作用。举个例子，如果开发者想要在Azure平台中利用OpenAI的GPT系列模型，就可以使用这个集成项目。更为重要的是，它来源于Azure平台，一个在云计算领域具有权威地位的平台，这确保了其稳定性和高效性，为LLM应用提供了有力的支持。

集成步骤

1. 从哪里导入类

在终端中先设置以下参数：

# Set this to `azure`

export OPENAI\_API\_TYPE=azure

# The API version you want to use: set this to `2023-05-15` for the released version.

export OPENAI\_API\_VERSION=2023-05-15

# The base URL for your Azure OpenAI resource. You can find this in the Azure portal under your Azure OpenAI resource.

export OPENAI\_API\_BASE=https://your-resource-name.openai.azure.com

# The API key for your Azure OpenAI resource. You can find this in the Azure portal under your Azure OpenAI resource.

export OPENAI\_API\_KEY=<your Azure OpenAI API key>

对于Azure OpenAI集成项目，从 langchain.llms 中导入了 AzureOpenAI 类。具体代码如下：

安装Langchain的Python库后，我们就可以导入AzureOpenAI 类。在langchain.llms中，我们已经集成了各大模型平台的API封装。当你在VSCode中编辑并输入点号后，它会自动列出所有可用的封装集成。为了方便识别，这些集成通常都会以其模型平台的名称作为类名的前缀，如AzureOpenAI。

在 langchain.chat\_models 中，我们针对聊天模型专门实现了各大模型平台的API封装。当你在VSCode环境中编程并输入点号后，系统会自动列举所有的聊天模型封装集成选项。为了方便开发者迅速识别，这些封装通常以其对应的模型平台名称作为类名的前缀，例如AzureOpenAI。如果你想使用Azure的聊天模型，可以直接通过from langchain.chat\_models.azure\_openai import AzureChatOpenAI来导入AzureChatOpenAI类。

from langchain.llms import AzureOpenAI

2. 使用方法

Azure OpenAI在使用上提供了一个简洁明了的接口。主要包含以下两个步骤：

- 初始化 ：首先创建一个AzureOpenAI的实例。在这个过程中，需要指定部署名称和模型名称。

llm = AzureOpenAI(

deployment\_name="td2",

model\_name="text-davinci-002",

)

- 执行 ：一旦初始化完成，可以轻松地运行LLM应用并获得结果。例如，要请求一个笑话，只需调用此实例并提供相应的提示：

llm("Tell me a joke")

此外，Azure OpenAI API的配置可以通过环境变量或直接在Python环境中进行，与标准OpenAI API的使用方式略有不同。

效果展示

在运行Azure OpenAI LLM应用后，得到的响应如下：

"\n\nWhy couldn't the bicycle stand up by itself? Because it was...two tired!"

此外，通过调用 print 方法，开发者可以查看LLM应用的自定义输出：

print(llm)

# 打印结果：AzureOpenAI

Params: {'deployment\_name': 'text-davinci-002', 'model\_name': 'text-davinci-002', 'temperature': 0.7, 'max\_tokens': 256, 'top\_p': 1, 'frequency\_penalty': 0, 'presence\_penalty': 0, 'n': 1, 'best\_of': 1}

### 10.2.2 Hugging Face Hub 集成

Hugging Face Hub集成项目是一个旨在提高机器学习和深度学习模型可访问性和协作性的创新平台。该项目专注于提供一个中心化的位置，供开发者和研究者分享、发布和协作开发各种NLP模型。除了存储预训练模型，Hugging Face Hub还支持多种框架和库，确保开发者可以轻松地集成和部署这些模型到他们的LLM应用中。此外，通过API和其他工具的集成，Hugging Face Hub使得与其他平台的交互更为流畅，为机器学习社区带来了巨大价值。对于Hugging Face Hub集成项目，相关的类从 langchain 中导入。具体代码如下：

集成步骤

from langchain import HuggingFaceHub, PromptTemplate, LLMChain

集成Hugging Face Hub主要涉及以下步骤：

- 安装和设置 ：首先确保安装了 huggingface\_hub Python包。

pip install huggingface\_hub

接下来，获取API令牌并为其设置环境变量。

from getpass import getpass

HUGGINGFACEHUB\_API\_TOKEN = getpass()

os.environ["HUGGINGFACEHUB\_API\_TOKEN"] = HUGGINGFACEHUB\_API\_TOKEN

- 准备示例 ：构建一个问题和相应的模板，然后使用 PromptTemplate 生成相应的提示。

question = "Who won the FIFA World Cup in the year 1994? "

template = """Question: {question}\nAnswer: Let's think step by step."""

prompt = PromptTemplate(template=template, input\_variables=["question"])

- 模型示例 ：使用不同的模型库和repo\_id来实例化 HuggingFaceHub 并运行LLMChain。

repo\_id = "<specific\_repo\_id>"

llm = HuggingFaceHub(

repo\_id=repo\_id, model\_kwargs={"temperature": 0.5, "max\_length": 64}

)

llm\_chain = LLMChain(prompt=prompt, llm=llm)

print(llm\_chain.run(question))

其中， <specific\_repo\_id> 可以是Flan, Dolly, Camel, XGen, Falcon等模型的特定ID。

效果展示

使用Flan模型：

repo\_id = "google/flan-t5-xxl"

llm = HuggingFaceHub(

repo\_id=repo\_id, model\_kwargs={"temperature": 0.5, "max\_length": 64}

)

llm\_chain = LLMChain(prompt=prompt, llm=llm)

print(llm\_chain.run(question))

- 使用Flan模型得到的回答是：

The FIFA World Cup was held in the year 1994. West Germany won the FIFA World Cup in 1994.

- 使用Dolly模型：

repo\_id = "databricks/dolly-v2-3b"

llm = HuggingFaceHub(

repo\_id=repo\_id, model\_kwargs={"temperature": 0.5, "max\_length": 64}

)

llm\_chain = LLMChain(prompt=prompt, llm=llm)

print(llm\_chain.run(question))

使用Dolly模型，得到的回答是：

First of all, the world cup was won by the Germany. Then the Argentina won the world cup in 2022. So, the Argentina won the world cup in 1994.

- 使用Camel模型、XGen模型和Falcon模型也会得到类似的输出，具体取决于所选的模型和参数配置。

## 10.3 聊天模型集成指南

Anthropic、Google PaLM Chat API和ChatOpenAI，这三个平台在2023年的大语言模型赛道上，锋芒毕现，都已成为语言模型和聊天机器人领域的重量级选手。

随着GPT-4 等大语言模型的突破，聊天机器人已经不仅仅是简单的问答工具，它们现在广泛应用于客服、企业咨询、电子商务等多种场景，为用户提供准确、快速的反馈。在这样的背景下，开发者们急需一套可以轻松切换、集成不同平台的工具。

正是基于这样的需求，Anthropic API、Google PaLM Chat API和ChatOpenAI的API封装应运而生。这些封装不仅为开发者提供了与上述三大平台的稳定交互能力，而且确保了在开发和部署聊天机器人的过程中，无论是从哪个平台切换到哪个平台，都能够做到高效和快速。对于开发者而言，这无疑大大降低了开发难度和时间成本。

### 10.3.1 Anthropic集成

集成步骤

1. 从哪里导入类。Anthropic聊天模型的集成涉及从 langchain.chat\_models 、 langchain.prompts.chat 和 langchain.schema 中导入相关的类。

在 langchain.chat\_models中，我们针对聊天模型专门实现了各大模型平台的API封装。当你在VSCode环境中编程并输入点号后，系统会自动列举所有的聊天模型封装集成选项。为了方便开发者迅速识别，这些封装通常以“Chat”加上其对应的模型平台名称作为类名的前缀，例如ChatAnthropic。如果你想使用Anthropic的聊天模型，可以直接通过 from langchain.chat\_models import ChatAnthropic来导入ChatAnthropic类。具体导入代码为：

from langchain.chat\_models import ChatAnthropic

同时也要引入了langchain.schema中定义的三种消息类型。这些工具主要用于格式化和处理聊天模型的输入数据。具体来说，聊天模型包装器期望的输入是一个消息列表，而不是单一的字符串。因此，当开发者利用此包装器与模型进行交互时，他们需要确保按照特定的结构组织数据，满足模型的输入要求。

from langchain.prompts.chat import (

ChatPromptTemplate,

SystemMessagePromptTemplate,

AIMessagePromptTemplate,

HumanMessagePromptTemplate,

)

from langchain.schema import AIMessage, HumanMessage, SystemMessage

2. 使用方法。Anthropic聊天模型的集成可以总结为以下几个步骤：

- 基本使用 ：首先，我们创建一个 ChatAnthropic 实例，然后使用它来处理消息。

chat = ChatAnthropic()

messages = [HumanMessage(content="Translate this sentence from English to French. I love programming.")]

chat(messages)

这会得到输出：

AIMessage(content=" J'aime la programmation.", additional\_kwargs={}, example=False)

Anthropic聊天模型不仅提供了基本的聊天功能，还进一步支持异步和流功能，为开发者提供更为灵活和高效的交互方式。

在代码中，我们可以看到从langchain.callbacks.manager导入的CallbackManager和从langchain.callbacks.streaming\_stdout导入的StreamingStdOutCallbackHandler。CallbackManager用于管理各种回调操作，确保异步任务的顺利执行。

而StreamingStdOutCallbackHandler则专门用于处理流输出，即实时将模型的响应输出到标准输出流。结合这两个工具，开发者可以更加轻松地利用Anthropic聊天模型的高级功能，确保数据处理的实时性和流畅性。

from langchain.callbacks.manager import CallbackManager

from langchain.callbacks.streaming\_stdout import StreamingStdOutCallbackHandler

await chat.agenerate([messages])

这将返回以下输出：

LLMResult(generations=[[ChatGeneration(text=" J'aime programmer.", generation\_info=None, message=AIMessage(content=" J'aime programmer.", additional\_kwargs={}, example=False))]], llm\_output={}, run=[RunInfo(run\_id=UUID('8cc8fb68-1c35-439c-96a0-695036a93652'))])

同样，我们可以设置 streaming 和 callback\_manager 参数来启用流功能：

chat = ChatAnthropic(

streaming=True,

verbose=True,

callback\_manager=CallbackManager([StreamingStdOutCallbackHandler()]),

)

chat(messages)

这会得到输出：

J'aime la programmation.

AIMessage(content=" J'aime la programmation.", additional\_kwargs={}, example=False)

效果展示

Anthropic聊天模型能够根据提供的人类消息进行响应。例如，在上述示例中，模型成功地将英文句子“I love programming.”翻译成法文句子“J'aime la programmation.”。

此外，通过异步和流功能，开发者可以更加灵活地使用模型，使其更加适应各种实时交互的场景。

### 10.3.2 Google PaLM Chat集成

集成步骤

1. 导入所需的类和方法

为了使用Vertex AI聊天模型，您需要首先安装 google-cloud-aiplatform 。

#!pip install google-cloud-aiplatform

然后，从 langchain.chat\_models 、 langchain.prompts.chat 和 langchain.schema 导入相关的类和方法。这是为了支持模型交互、聊天提示和消息架构。

from langchain.chat\_models import ChatVertexAI

from langchain.prompts.chat import (

ChatPromptTemplate,

SystemMessagePromptTemplate,

HumanMessagePromptTemplate,

)

from langchain.schema import HumanMessage, SystemMessage

2. 使用方法

Vertex AI聊天模型的使用可以分为以下几个步骤：

- 基本使用 ：您可以创建一个 ChatVertexAI 实例，并用它来处理消息。

chat = ChatVertexAI()

messages = [

SystemMessage(

content="You are a helpful assistant that translates English to French."

),

HumanMessage(

content="Translate this sentence from English to French. I love programming."

),

]

chat(messages)

例如，您可以给模型发送一个系统消息，告诉它它是一个帮助将英语翻译成法语的助手，然后发送一个人类消息，要求翻译一个句子。

这将返回如下输出：

AIMessage(content='Sure, here is the translation of the sentence "I love programming" from English to French: J\'aime programmer.', additional\_kwargs={}, example=False)

为了开发者可以在不同的情境创建不同的提示词模板，然后在实际对话中动态地填充这些模板，参照以下代码进行配置提示词模板。先定义一个提示词模板，明确表达助手的功能——即从{input\_language}翻译至{output\_language}。基于这个定义，我们使用SystemMessagePromptTemplate.from\_template方法创建了一个系统消息的提示词模板对象。

template = (

"You are a helpful assistant that translates {input\_language} to {output\_language}."

)

system\_message\_prompt = SystemMessagePromptTemplate.from\_template(template)

human\_template = "{text}"

human\_message\_prompt=HumanMessagePromptTemplate.from\_template(human\_template)

接下来，代码同样为人类输入定义了一个简洁的模板，并进一步利用HumanMessagePromptTemplate.from\_template实例化了人类消息的提示词模板对象。

为了方便地组合多个角色的模板消息，如系统、AI和人类等，代码利用了ChatPromptTemplate.from\_messages方法。这个方法接收一系列的模板对象，并将它们合成为一个完整的聊天提示词模板。

最后，我们使用format\_prompt方法，将具体的参数——如输入语言、输出语言和用户文本——绑定进预定义的模板中。这样，经过格式化的提示词可以反映出助手的功能（从英语翻译至法语）和用户的原始输入（"I love programming."），并将其整合为一个完整的、为语言模型准备的提示词，从而引导模型提供相关的回复。

chat\_prompt = ChatPromptTemplate.from\_messages(

[system\_message\_prompt, human\_message\_prompt]

)

# get a chat completion from the formatted messages

chat(

chat\_prompt.format\_prompt(

input\_language="English", output\_language="French", text="I love programming."

).to\_messages()

)

这将返回如下输出：

AIMessage(content='Sure, here is the translation of "I love programming" in French: J\'aime programmer.', additional\_kwargs={}, example=False)

Vertex AI还提供了Codey API，更改模型型号为“codechat-bison”，专门用于代码帮助。例如，当您询问如何创建一个Python函数来识别所有的质数时，它可以提供相关的代码建议。

chat = ChatVertexAI(model\_name="codechat-bison")

messages = [

HumanMessage(

content="How do I create a python function to identify all prime numbers?"

)

]

chat(messages)

这将返回如下输出：

AIMessage(content='The following Python function can be used to identify all prime numbers up to a given integer: ...', additional\_kwargs={}, example=False)

Vertex AI提供了强大的聊天功能，使开发者能够与模型进行直观的交互。通过使用不同的消息类型，如 HumanMessage 和 SystemMessage ，开发者可以更好地引导模型的行为。

使用 MessagePromptTemplate 可以进一步增强这种交互，因为开发者可以为特定的任务或场景创建定制的模板，而不是每次都手动构建完整的消息。

Codey API是一个强大的工具，特别是对于需要代码帮助的开发者。它可以根据用户的问题提供相关的代码建议，从而帮助他们更快地解决问题。

### 10.3.3 Azure上的OpenAI端点集成

集成步骤

1. 导入所需的类和方法

若要使用Azure上托管的OpenAI端点，您需要从 langchain.chat\_models 和 langchain.schema 导入相关的类和方法，以支持模型交互和消息架构。

from langchain.chat\_models import AzureChatOpenAI

from langchain.schema import HumanMessage

1. 使用方法

BASE\_URL = "https://${TODO}.openai.azure.com"

API\_KEY = "..."

DEPLOYMENT\_NAME = "chat"

model = AzureChatOpenAI(

openai\_api\_base=BASE\_URL,

openai\_api\_version="2023-05-15",

deployment\_name=DEPLOYMENT\_NAME,

openai\_api\_key=API\_KEY,

openai\_api\_type="azure",

)

与Azure上的OpenAI端点交互主要涉及以下步骤：首先，您需要配置必要的基本信息，包括Azure上的OpenAI API的基本URL（BASE\_URL）、API密钥（API\_KEY）用于身份验证及访问服务，以及代表Azure部署的名称（DEPLOYMENT\_NAME）。

特别注意，${TODO} 部分应替换为您在Azure上的OpenAI服务的真实URL部分。有了这些信息，您可以创建一个名为AzureChatOpenAI的模型实例，其中openai\_api\_type的值已经被设定为 "azure" ，确保API请求会被重定向到Azure托管的OpenAI端点。最后，与其他聊天模型的交互方式相同，您可以向该模型发送一个HumanMessage，例如请求将英语句子翻译成法语，并从模型中获取对应的回复。

model(

[

HumanMessage(

content="Translate this sentence from English to French. I love programming."

)

]

)

这将返回如下输出：

AIMessage(content="J'aime programmer.", additional\_kwargs={})

这是一个简单且直接的方法来与Azure上托管的OpenAI端点进行交互。一旦您已经在Azure上配置好了OpenAI服务，并获取了相关的API密钥和URL，这个过程就变得相对简单。

Azure提供了一个可靠且安全的环境来托管和运行OpenAI模型，这为企业和开发者提供了一个在云中快速部署和扩展AI解决方案的方法。

需要注意的是，每当您与Azure上的服务进行交互时，都应确保保护好您的API密钥，以防止任何未授权的访问或潜在的滥用。

## 10.4 向量库集成指南

[向量库](https://www.pinecone.io/learn/vector-database/)是一种索引和存储向量嵌入以实现高效管理和快速检索的数据库。与单独的[向量索引](https://www.pinecone.io/learn/vector-indexes/)不同，像Pinecone这样的向量数据库提供了额外的功能，例如索引管理、数据管理、元数据存储和过滤以及水平扩展。

向量库在多种应用场景中发挥着关键作用，特别是在处理大数据和复杂查询时。其中，语义文本搜索是一个典型的应用，用户可以通过NLP转换器和句子嵌入模型将文本数据转化为向量嵌入，再利用Pinecone这类工具进行索引和搜索。此外，它还可以支持生成问答系统，即从Pinecone检索与特定查询相关的上下文，然后传递给如OpenAI这样的生成模型，从而产生基于真实数据的答案。

不仅如此，向量库的应用还扩展到了图像和电商领域。例如，通过将图像数据转化为向量嵌入，再使用Pinecone之类的工具构建索引，我们可以轻松地执行图像的相似性搜索。同时，基于代表用户兴趣和行为的向量，向量库可以为电子商务平台生成产品推荐，从而实现个性化的用户体验。

Pinecone, Chroma, Milvus : 这三种存储解决方案为机器学习和深度学习中的向量数据存储提供了优质的选择，确保向量数据的稳定、高效存储。

### 10.4.1 Chroma集成

集成步骤

在这个基础示例中，我们首先加载了最近的国情咨文，将其分割成几部分，使用开源嵌入模型进行嵌入，加载到Chroma中，然后对其进行查询。

1. 导入所需的类和方法，先安装向量库chromadb。

pip install chromadb

从 langchain.embeddings.sentence\_transformer ， langchain.text\_splitter ， langchain.vectorstores 和 langchain.document\_loaders 导入相关的类和方法来支持文档加载、文本分割、嵌入和向量存储。

# import

from langchain.embeddings.sentence\_transformer import SentenceTransformerEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import Chroma

from langchain.document\_loaders import TextLoader

1. 使用方法。

为了有效地处理美国国情咨文，首先需要使用TextLoader进行文档的加载。在加载后，借助CharacterTextSplitter，我们将文档分割成大小为1000字符的各个块，这些块之间存在0字符的重叠。在文档准备完成后，我们采用SentenceTransformerEmbeddings来创建一个名为all-MiniLM-L6-v2的嵌入模型型号。

接着，为了实现向量化搜索和相似度检测，我们使用Chroma.from\_documents方法将这些嵌入后的文档加载到Chroma中。一旦数据加载完毕，便可以利用db.similarity\_search方法查询文档，快速找到与特定查询内容最相关的文档部分。

# load the document and split it into chunks

loader = TextLoader("../../../state\_of\_the\_union.txt")

documents = loader.load()

# split it into chunks

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

docs = text\_splitter.split\_documents(documents)

# create the open-source embedding function

embedding\_function = SentenceTransformerEmbeddings(model\_name="all-MiniLM-L6-v2")

# load it into Chroma

db = Chroma.from\_documents(docs, embedding\_function)

# query it

query = "What did the president say about Ketanji Brown Jackson"

docs = db.similarity\_search(query)

# print results

print(docs[0].page\_content)

打印获得的结果是：

/Users/jeff/.pyenv/versions/3.10.10/lib/python3.10/site-packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user\_install.html

from .autonotebook import tqdm as notebook\_tqdm

Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Voting Rights Act. And while you’re at it, pass the Disclose Act so Americans can know who is funding our elections.

Tonight, I’d like to honor someone who has dedicated his life to serve this country: Justice Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the United States Supreme Court. Justice Breyer, thank you for your service.

One of the most serious constitutional responsibilities a President has is nominating someone to serve on the United States Supreme Court.

And I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brown Jackson. One of our nation’s top legal minds, who will continue Justice Breyer’s legacy of excellence.

3. 分析。 这个示例演示了如何使用 langchain 库处理、嵌入和查询文档的过程。在这里，我们对美国国情咨文进行了查询，具体查找总统关于美国Ketanji Brown Jackson的言论，并成功找到了相关内容。

这种查询可以用于许多不同的应用场景，如新闻文章分析、法律文档查询、学术研究等。使用开源嵌入模型和Chroma这样的向量存储工具，可以有效地搜索大量的文档，并快速找到与特定查询相关的部分。

需要注意的是，在实际应用中，您可能需要调整文档分割的大小和嵌入模型的选择，以适应特定的需求和数据集。

### 10.4.2 Pinecone集成

Pinecone是一个高效的向量搜索服务，特别针对那些需要处理大量数据的应用而设计。高速性能是其核心特点之一，即使在数十亿个条目的数据集上，它都能确保超低的查询延迟，从而为用户提供即时的搜索反馈。

除了高速查询，Pinecone还具备实时更新的能力。这意味着当你添加、编辑或删除数据时，其索引会被实时地更新，确保数据的实时性和准确性。这为动态变化的数据环境提供了极大的便利。

最后，Pinecone融合了向量搜索与元数据过滤的功能。这使得它不仅可以根据向量相似性搜索，还可以结合元数据进行过滤，从而提供更为精准的搜索结果。而作为一个完全托管的服务，Pinecone使得用户无需担心后端的复杂性和安全性，可以专注于实现其业务需求。这个示例提供了如何与 Pinecone 向量数据库交互的步骤。

1. 初始化和设置。首先，用户需要通过 pip 安装 pinecone-client ， openai ， tiktoken 和 langchain 。tiktoken 是一个文档计数工具，用于计算文档中的词数或标记数，而无需进行实际的模型转换。这在评估模型所需的令牌数或预估模型调用的成本时尤为有用。简单来说，tiktoken 提供了一种高效的方式来了解文档的大小和复杂性。

pip install pinecone-client openai tiktoken langchain

为了与 Pinecone 互动，用户需要输入 Pinecone API 密钥和 Pinecone 环境。此外，由于我们要使用 OpenAIEmbeddings，所以也需要 OpenAI 的 API 密钥。

import os

import getpass

os.environ["PINECONE\_API\_KEY"] = getpass.getpass("Pinecone API Key:")

os.environ["OPENAI\_API\_KEY"] = getpass.getpass("OpenAI API Key:")

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import Pinecone

from langchain.document\_loaders import TextLoader

2. 加载和分割文档。与之前Chroma的示例类似，这里使用 TextLoader 加载文档，并使用 CharacterTextSplitter 将其分割。

from langchain.document\_loaders import TextLoader

loader = TextLoader("../../../state\_of\_the\_union.txt")

documents = loader.load()

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

docs = text\_splitter.split\_documents(documents)

1. 嵌入文档。使用嵌入模型包装器OpenAIEmbeddings进行文档嵌入。

embeddings = OpenAIEmbeddings()

4. 与 Pinecone 互动。使用 Pinecone 的 API 初始化 Pinecone，然后检查我们的索引是否已存在。如果不存在，我们创建一个新的索引。OpenAI 的 text-embedding-ada-002 模型使用 1536 维，所以我们需要设置维度为 1536。

1536维在这里指的是由OpenAI的`text-embedding-ada-002`模型生成的每个文本嵌入（或称为向量表示）所具有的维度或特征数。简单地说，当这个模型接受一个文本输入并为其生成嵌入时，输出的嵌入向量将有1536个数值或坐标。这1536个值捕获了文本的语义信息，使得具有相似意义的文本具有相近的向量表示。因此，当我们创建一个索引来存储由此模型生成的嵌入时，需要确保该索引能够容纳1536维的数据，从而确保每一个维度的信息都被完整地保存下来。

import pinecone

# initialize pinecone

pinecone.init(

api\_key=os.getenv("PINECONE\_API\_KEY"), # find at app.pinecone.io

environment=os.getenv("PINECONE\_ENV"), # next to api key in console

)

index\_name = "langchain-demo"

# First, check if our index already exists. If it doesn't, we create it

if index\_name not in pinecone.list\_indexes():

# we create a new index

pinecone.create\_index(

name=index\_name,

metric='cosine',

dimension=1536

)

# The OpenAI embedding model `text-embedding-ada-002 uses 1536 dimensions`

docsearch = Pinecone.from\_documents(docs, embeddings, index\_name=index\_name)

# if you already have an index, you can load it like this

# docsearch = Pinecone.from\_existing\_index(index\_name, embeddings)

query = "What did the president say about Ketanji Brown Jackson"

docs = docsearch.similarity\_search(query)

5. 文档搜索。使用similarity\_search 方法查询文档，并输出查询结果。当使用similarity\_search方法查询文档时，首先通过嵌入模型将每个文档和查询都转换为向量。然后，计算查询向量与文档向量间的相似度，通常基于余弦相似度。最后，根据相似度排序并返回与查询最相关的文档。

print(docs[0].page\_content)

6. 向现有索引添加更多文本。使用add\_texts函数将更多的文本嵌入到现有的 Pinecone 索引中。首先，初始化一个代表该索引的对象。然后，它创建一个Pinecone向量存储实例，该实例将使用指定的嵌入函数将文本转化为向量。最后，使用add\_texts函数，它将字符串 "More text!" 的向量表示形式加入到这个索引中，以便于后续的相似度查询。

index = pinecone.Index("langchain-demo")

vectorstore = Pinecone(index, embeddings.embed\_query, "text")

vectorstore.add\_texts("More text!")

7. 最大边际相关性搜索。除了使用 similarity\_search ，用户还可以使用 mmr 作为检索器。这为用户提供了一种新的、可能更加相关的方法来查询文档。使用了最大边际相关性(Maximum Margin Relevance, MMR)搜索方法来查询文档，从而可能提供更相关的搜索结果。首先，它通过as\_retriever函数将文档搜索器设置为使用 "mmr" 作为其检索类型。然后，它用指定的查询query来获取相关的文档。在获取的文档中，代码遍历每一份匹配的文档，并打印其内容。

retriever = docsearch.as\_retriever(search\_type="mmr")

matched\_docs = retriever.get\_relevant\_documents(query)

for i, d in enumerate(matched\_docs):

print(f"\n## Document {i}\n")

print(d.page\_content)

这个示例为我们提供了一个使用 Pinecone 向量数据库的全面指南。从初始化到查询，用户可以根据自己的需求进行各种操作。

### 10.4.2 Milvus集成

Milvus是一个专门的向量数据库，于2019年首次亮相，旨在为由深度神经网络和其他机器学习模型生成的大规模嵌入向量提供存储、索引和管理。这不仅仅是一个常规的数据库；它是特别为处理向量数据而设计的，能够轻松管理万亿级别的向量索引。

与传统的关系型数据库不同，它们通常用于存储和查询结构化数据，Milvus从其核心设计上就是为了处理从非结构化数据生成的嵌入向量。这是因为在当前的互联网时代，非结构化数据，如电子邮件、论文、物联网传感器数据和社交媒体图片，越来越普遍。

为了使这些非结构化数据对机器有意义，科研人员和工程师经常使用嵌入技术将它们转化为数值向量。这些向量捕获了原始数据的关键特征和信息。Milvus的主要任务是存储这些嵌入向量，并为之提供高效查询功能。

此外，Milvus还能够衡量向量之间的相似性。它通过计算向量间的距离来评估相似度，因此，如果两个向量很接近，则它们表示的原始非结构化数据也很相似。这一特点使Milvus在很多领域，如推荐系统、图像搜索和自然语言处理，都成为了一个强大的工具。

1. 准备工作。先确保已经运行了一个 Milvus 实例，并通过 pip 安装了 pymilvus 。

%pip install pymilvus

2. 设置 OpenAI API 密钥

由于我们想使用 OpenAIEmbeddings，所以需要获取 OpenAI 的 API 密钥。通过设置环境变量实现这一点。

import os

import getpass

os.environ["OPENAI\_API\_KEY"] = getpass.getpass("OpenAI API Key:")

3. 加载和分割文档。这部分与之前的 Azure 和 Pinecone 示例类似。首先使用 TextLoader 从给定的路径加载文档。然后，使用 CharacterTextSplitter 根据给定的大小分割这些文档。

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.text\_splitter import CharacterTextSplitter

from langchain.vectorstores import Milvus

from langchain.document\_loaders import TextLoader

4. 嵌入文档

使用 OpenAI 的模型来为这些文档生成嵌入向量。

from langchain.document\_loaders import TextLoader

loader = TextLoader("../../../state\_of\_the\_union.txt")

documents = loader.load()

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

docs = text\_splitter.split\_documents(documents)

embeddings = OpenAIEmbeddings()

5. 与 Milvus 互动。在与Milvus互动的步骤中，首先根据给定的参数（如主机名和端口号）建立与Milvus实例的连接。一旦连接建立，用户便可以将之前处理过并转换为向量形式的文档加载到Milvus数据库中，为后续的查询和分析做好准备。

vector\_db = Milvus.from\_documents(

docs,

embeddings,

connection\_args={"host": "127.0.0.1", "port": "19530"},

)

6. 文档搜索

和前面的Pinecone 示例一样，可以使用 similarity\_search 方法查询与输入查询相似的文档。在这个例子中，查询的是 "What did the president say about Ketanji Brown Jackson"，并且返回了相关的段落。

query = "What did the president say about Ketanji Brown Jackson"

docs = vector\_db.similarity\_search(query)

打印搜索的相关文档结果。

docs[0].page\_content

# 'Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Voting Rights Act. And while you’re at it, pass the Disclose Act so Americans can know who is funding our elections. \n\nTonight, I’d like to honor someone who has dedicated his life to serve this country: Justice Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the United States Supreme Court. Justice Breyer, thank you for your service. \n\nOne of the most serious constitutional responsibilities a President has is nominating someone to serve on the United States Supreme Court. \n\nAnd I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brown Jackson. One of our nation’s top legal minds, who will continue Justice Breyer’s legacy of excellence.'

此示例提供了如何使用 Milvus 向量数据库的指南。从初始化到查询，提供了完整的步骤，使用户能够轻松地使用 Milvus 进行大规模文档嵌入和查询。与 OpenAI 和 langchain 结合使用，Milvus 可以为用户提供高效的向量搜索和查询能力。

## 10.5 嵌入模型集成指南

Cohere Embeddings提供了与Cohere平台的无缝对接，确保文本嵌入过程既高效又精确。而HuggingFaceHub Embeddings和llamacpp embeddings则代表了另外两种强大的文本嵌入集成方法。它们都经过严格的测试，以确保与HuggingFaceHub和llamacpp平台的稳定和高效交互，使得开发者可以更轻松地在其LLM应用中利用这些先进的嵌入技术。

### 10.5.1 HuggingFaceEmbeddings嵌入集成

Sentence Transformers Embeddings为开发者提供了一种高效、简洁的方式来为文本生成向量嵌入。这种嵌入通常基于深度学习模型，专为捕捉文本之间的复杂语义关系而设计。利用这种技术，开发者能够实现更加精确的文本匹配和更深入的内容分析，为LLM应用带来了优化的效果。

1. SentenceTransformerEmbeddings 简介。实际上，Sentence Transformers Embeddings是通过HuggingFaceEmbeddings集成进行调用的。对于那些已经熟悉sentence\_transformers包的开发者，为了使其更容易上手和进行整合，Langchain提供了SentenceTransformerEmbeddings的别名，这样开发者可以在代码中使用熟悉的命名方式。

2. 首先需要对开发环境进行配置。确保在开始前已经正确地安装了sentence\_transformers包，这是为了确保文本嵌入的流程可以顺利进行。如未安装，可以通过提供的安装命令轻松完成设置。

pip install sentence\_transformers > /dev/null

注意，为了保证环境的稳定性，请及时更新 pip 到其最新版本。

[notice] A new release of pip is available: 23.0.1 -> 23.1.1

[notice] To update, run: pip install --upgrade pip

- 引入必要模块 ：

开发者需要从 langchain.embeddings 模块导入HuggingFaceEmbeddings和SentenceTransformerEmbeddings。

from langchain.embeddings import HuggingFaceEmbeddings,SentenceTransformerEmbeddings

3. 文本嵌入实践。首要步骤是初始化嵌入模型。通过HuggingFaceEmbeddings并指定相关的模型名称 all-MiniLM-L6-v2 来设定所需的嵌入模型包装器，从而为后续的文本转换工作做好准备。

embeddings = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")

对于更熟悉 SentenceTransformer 的开发者，上述初始化等同于：

embeddings = SentenceTransformerEmbeddings(model\_name="all-MiniLM-L6-v2")

在LLM应用中，当开发者遇到文本数据，如"This is a test document."，他们可以使用先前初始化的嵌入模型，直接将此文本转换为相应的向量嵌入，为后续分析或其他操作提供机器可理解的格式。

text = "This is a test document."

query\_result = embeddings.embed\_query(text)

如果需要处理多个文档的嵌入时，例如同时嵌入 "This is a test document." 和 "This is not a test document."，可以将这些文本作为列表传递给embeddings.embed\_documents方法，从而一次性得到这些文档的对应嵌入结果，存储在doc\_result中。

doc\_result = embeddings.embed\_documents([text, "This is not a test document."])

通过上述步骤，开发者不仅可以轻松集成和使用Sentence Transformers Embeddings，还可以确保其在LLM应用中的高效性和准确性。

### 10.5.2 Llama-cpp嵌入集成

llama.cpp 主要目标是在MacBook上使用4位整数量化运行LLaMA模型。它是一个纯粹的C/C++实现，不依赖任何外部库。尽管该程序优先考虑Apple芯片并通过ARM NEON, Accelerate和Metal框架进行优化，但它也为x86架构提供了AVX、AVX2和AVX512的支持。此外，该程序在计算精度上支持混合的F16 / F32，并能够支持4位、5位和8位的整数量化。对于BLAS操作，它支持各种库，如OpenBLAS、Apple BLAS、ARM Performance Lib、ATLAS、BLIS、Intel MKL、NVHPC、ACML、SCSL、SGIMATH等。另外，也支持cuBLAS和CLBlast。

在准备集成和使用Llama-cpp之前，开发者首先需要设置其开发环境。具体来说，必须确保已经安装了llama-cpp-python库。这可以通过简单地运行特定的安装命令来实现。

pip install llama-cpp-python

为了简化开发过程，Llama-cpp的嵌入模块已被预先集成。因此，开发者可以直接从langchain.embeddings模块中导入LlamaCppEmbeddings来使用这个功能。

from langchain.embeddings import LlamaCppEmbeddings

在实际应用中，要利用Llama-cpp进行文本嵌入，首先需要初始化模型。

开发者可以通过LlamaCppEmbeddings类并为其提供特定的模型路径（例如“/path/to/model/ggml-model-q4\_0.bin”）来完成这个步骤，从而为后续操作创建一个Llama-cpp嵌入模型实例。

llama = LlamaCppEmbeddings(model\_path="/path/to/model/ggml-model-q4\_0.bin")

为了从LLM应用中的文本数据中生成嵌入，开发者只需将所需文本传递给已初始化的LlamaCppEmbeddings实例。例如，对于文本 "This is a test document."，通过调用Llama-cpp模型的嵌入方法，开发者可以轻松获得该文本的向量嵌入表示。

text = "This is a test document."

query\_result = llama.embed\_query(text)

对于要嵌入的一组文档，开发者可以简单地传递一个包含所有文档的列表给llama.embed\_documents方法。例如，将文本列表[text]传入方法后，它将返回这些文档的向量嵌入表示，并存储在doc\_result变量中。这使得对多个文档的批量嵌入变得简单高效。

doc\_result = llama.embed\_documents([text])

4. 深入应用

一旦成功集成Llama-cpp到LangChain中，开发者就可以充分利用其高效的文本嵌入能力，进一步为LLM应用带来高准确率和低延迟的体验。通过不断地测试、优化和调整，开发者可以确保Llama-cpp嵌入在各种场景下都能稳定发挥其最大潜能。

### 10.5.3 Cohere嵌入集成

随着各种文本嵌入技术的发展，Cohere成为了开发者在LLM应用中的又一选择。其凭借稳定性和高效性受到许多开发者的欢迎。接下来，我们将探讨如何在LangChain中集成和使用Cohere嵌入。

Cohere是一个先进的文本嵌入工具，旨在为文本数据生成精确的表示。与其他嵌入方法相比，Cohere在某些特定任务上可能具有更高的准确率和更好的性能。

Cohere利用如BERT和GPT等Transformer架构的深度学习模型为文本生成嵌入。这些嵌入不仅仅反映文本的表面结构，而是深入捕捉其语义含义，确保即使两段文本的字面表述不同，但只要它们的意思或概念相似，生成的嵌入也会是相似的。

在LangChain中，为了简化开发者的工作流并提供更便捷的Cohere嵌入使用体验，开发团队预置了与Cohere相关的嵌入模块。开发者只需通过简单地从langchain.embeddings模块中导入CohereEmbeddings类，就能轻松地在其应用中集成Cohere的功能。

from langchain.embeddings import CohereEmbeddings

在LangChain中，当开发者想要实际应用Cohere的嵌入功能时，首先需要拥有一个有效的Cohere API密钥，这是为了确保与Cohere服务的通信。一旦获得密钥，开发者可以简单地使用CohereEmbeddings类并通过cohere\_api\_key参数来初始化它，从而在应用中轻松地生成文本嵌入。

embeddings = CohereEmbeddings(cohere\_api\_key=cohere\_api\_key)

在LLM应用中，当开发者想要为特定的文本数据，例如"This is a test document."，生成嵌入表示时，他们可以直接利用Cohere的嵌入方法。这一方法将该文本转换为一个数值向量，该向量捕获了文本的语义含义，从而为后续的分析或操作提供了基础。

query\_result = embeddings.embed\_query(text)

如果需要为一组文档生成嵌入，可以使用以下方法：

doc\_result = embeddings.embed\_documents([text])

集成Cohere嵌入到LangChain中，开发者可以充分发挥其特有的文本嵌入优势，为LLM应用提供更准确的文本表示。当处理复杂的语言任务时，Cohere可能会带来更好的性能和稳定性。

为了满足LLM应用的开发者对高效、稳定的文本嵌入的需求，LangChain和Cohere之间的合作确保了最佳的集成体验和高效的嵌入生成。

## 10.6 Agent toolkits 集成指南

Agent toolkits的集成旨在简化并增强LLM应用中的数据处理和分析功能。CSV Agent提供了一个专门的工具，允许开发者无缝地处理CSV数据，降低了对复杂编码的依赖。Pandas Agent则集成了Pandas框架，赋予了开发者在应用中进行高效数据操作的能力。另外，为了满足先进的数据可视化需求，PowerBI Agent与Microsoft PowerBI紧密结合，为开发者带来了丰富的、直观的数据可视化工具。这些工具套件确保了LLM应用的数据处理、分析和可视化都既简单又高效。

### 10.6.1 CSV Agent的集成

LangChain为开发者提供了多种与CSV文件互动的方式，特别是针对问题回答任务。本文将详细讨论如何使用CSV Agent以及一些相关的安全注意事项。

CSV Agent主要用于与CSV文件交互，特别是当需要查询或检索信息时。需要注意的是，CSV Agent内部调用了Pandas DataFrame agent和Python agent。这意味着，当LLM生成的Python代码可能存在问题时，执行这些代码可能会导致意外的后果。因此，使用时应保持谨慎。

CSV Agent是LLM应用中用于处理CSV数据的工具，而在编程中，为了实现这一功能，开发者需要引用一系列特定的API。create\_csv\_agent是创建和管理CSV Agent的核心方法，位于langchain.agents模块中。而OpenAI来自langchain.llms模块，可能是用于模型管理或与OpenAI平台的交互。接下来，ChatOpenAI从langchain.chat\_models导入，可能是一个针对聊天模型的包装器。最后，AgentType定义了LLM应用中可以使用的各种Agent的类型，助于开发者明确各个Agent的角色和功能。通过这些API的结合，开发者可以在LLM应用中轻松创建和使用CSV Agent。

from langchain.agents import create\_csv\_agent

from langchain.llms import OpenAI

from langchain.chat\_models import ChatOpenAI

from langchain.agents.agent\_types import AgentType

初始化方法中，create\_csv\_agent函数被用于构建一个新的CSV Agent。此代理的特点是使用ZERO\_SHOT\_REACT\_DESCRIPTION类型，意味着该代理可以在没有预先训练的情况下对数据进行描述或反应。在示例代码中，代理被配置为处理名为titanic.csv的文件，使用OpenAI模型并设置其温度参数为0，以获得更确定性的输出。同时，通过verbose=True参数，代理在运行时会显示更多的详细信息。总之，此代码片段展示了如何快速设置并初始化一个能够对CSV数据进行零射击描述的代理。

agent = create\_csv\_agent(

OpenAI(temperature=0),

"titanic.csv",

verbose=True,

agent\_type=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION,

)

另外还可以使用OPENAI\_FUNCTIONS类型进行初始化。

这是一种不同于ZERO\_SHOT\_REACT\_DESCRIPTION的代理构建方法。在这种方法中，代理是基于特定的OpenAI功能进行操作的，特别是那些与ChatOpenAI模型相关的功能。例如，在给出的代码中，我们使用了模型gpt-3.5-turbo-0613来创建一个代理，该模型的温度参数被设置为0以获得确定性输出。这个代理专门为处理名为titanic.csv的文件而设，并通过verbose=True参数提供额外的运行时详细信息。

OPENAI\_FUNCTIONS类型是一个代理初始化选项，专门为高级的OpenAI模型设计。这些模型，例如gpt-3.5-turbo-0613，是在2023年6月13日之后发布的。之前发布的型号并不能使用OPENAI\_FUNCTIONS类型进行初始化。

agent = create\_csv\_agent(

ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613"),

"titanic.csv",

verbose=True,

agent\_type=AgentType.OPENAI\_FUNCTIONS,

)

使用CSV Agent查询数据是一个直接的方式，允许开发者对已加载的CSV文件进行交互式查询。在CSV Agent被初始化后，它会在后台读取和理解CSV文件内容。例如，通过调用agent.run()方法并提供相应的文本查询，如“how many rows are there?”，代理会检索文件并返回文件中行的数量。这种方法为开发者提供了一个简洁、直观的接口，使他们能够无缝地与CSV数据互动，而无需编写复杂的查询或处理逻辑。

agent.run("how many rows are there?")

或者询问具有超过3名兄弟姐妹的人数：

agent.run("how many people have more than 3 siblings")

CSV Agent的设计不仅仅局限于单个CSV文件的查询，它同样能够处理并与多个CSV文件交互。这意味着开发者可以轻松比较和分析多个数据集之间的差异。例如，通过传递两个CSV文件“titanic.csv”和“titanic\_age\_fillna.csv”给代理，您可以询问这两个数据框（DataFrames）在年龄列上的不同之处。这样的功能大大增强了CSV Agent的灵活性和实用性，为开发者提供了一个高效的工具来分析和对比不同的数据源。

agent = create\_csv\_agent(

ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613"),

["titanic.csv", "titanic\_age\_fillna.csv"],

verbose=True,

agent\_type=AgentType.OPENAI\_FUNCTIONS,

)

agent.run("how many rows in the age column are different between the two dfs?")

对于需要与CSV文件进行交互的LLM应用开发者，LangChain中的CSV Agent提供了一种高效且灵活的方法。然而，考虑到安全性问题，开发者应该谨慎使用，并确保不执行可能有害的LLM生成的Python代码。

### 10.6.2 Pandas Agent的集成

LangChain不仅提供了与CSV文件互动的方法，还为开发者提供了与pandas数据帧互动的方式。此功能特别适用于问题回答任务。本文将详细讨论如何使用Pandas Dataframe Agent以及与之相关的一些安全注意事项。

Pandas Dataframe Agent主要用于与pandas数据帧交互，特别是在需要查询或检索信息时。需要注意的是，此代理在后台调用Python代理，执行LLM生成的Python代码。如果LLM生成的Python代码可能是有害的，则执行此代码可能会导致意外的后果。因此，使用时应谨慎。

首先导入了一系列的模块和工具来处理和互动Pandas数据帧。使用create\_pandas\_dataframe\_agent，他们可以创建一个代理来与数据帧交互。同时，ChatOpenAI和OpenAI提供了与LLM模型的连接，可以为该代理提供理解和输出自然语言的回答。AgentType则定义了可能的代理类型。接着，他们使用Python的pandas库导入了一个CSV文件，名为"titanic.csv"，并将其读取为一个数据帧df。这样，开发者即可以利用LangChain中的工具和代理与这个数据帧进行互动。

from langchain.agents import create\_pandas\_dataframe\_agent

from langchain.chat\_models import ChatOpenAI

from langchain.agents.agent\_types import AgentType

from langchain.llms import OpenAI

import pandas as pd

df = pd.read\_csv("titanic.csv")

这种方法使用了ZERO\_SHOT\_REACT\_DESCRIPTION类型作为代理类型。如下是初始化代理的例子：

agent = create\_pandas\_dataframe\_agent(OpenAI(temperature=0), df, verbose=True)

另外还可以使用OPENAI\_FUNCTIONS类型进行初始化。

这是一种不同于ZERO\_SHOT\_REACT\_DESCRIPTION的代理构建方法。在这种方法中，代理是基于特定的OpenAI功能进行操作的，特别是那些与ChatOpenAI模型相关的功能。例如，在给出的代码中，我们使用了模型gpt-3.5-turbo-0613来创建一个代理，该模型的温度参数被设置为0以获得确定性输出。这个代理专门为处理名为titanic.csv的文件而设，并通过verbose=True参数提供额外的运行时详细信息。

OPENAI\_FUNCTIONS类型是一个代理初始化选项，专门为高级的OpenAI模型设计。这些模型，例如gpt-3.5-turbo-0613，是在2023年6月13日之后发布的。之前发布的型号并不能使用OPENAI\_FUNCTIONS类型进行初始化。

agent = create\_pandas\_dataframe\_agent(

ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613"),

df,

verbose=True,

agent\_type=AgentType.OPENAI\_FUNCTIONS,

)

与CSV Agent类似，一旦初始化Pandas Dataframe Agent，您可以运行查询以检索数据帧中的数据。

例如，如果您想知道数据帧中有多少行，只需使用agent.run("how many rows are there?")，代理将计算数据帧中的行数，并返回相应的结果。这为那些不熟悉Pandas语法的人提供了一个简单直接的方式来查询和分析数据。

agent.run("how many rows are there?")

或者询问具有超过3名兄弟姐妹的人数：

agent.run("how many people have more than 3 siblings")

除了单个数据帧，Pandas Dataframe Agent还支持与多个数据帧互动。例如，您可以将多个数据帧传递给代理，并询问两个数据帧之间年龄列的差异行数：

df1 = df.copy()

df1["Age"] = df1["Age"].fillna(df1["Age"].mean())

agent = create\_pandas\_dataframe\_agent(OpenAI(temperature=0), [df, df1], verbose=True)

agent.run("how many rows in the age column are different?")

# 输出结果：

> Entering new AgentExecutor chain...

Thought: I need to compare the age columns in both dataframes

Action: python\_repl\_ast

Action Input: len(df1[df1['Age'] != df2['Age']])

Observation: 177

Thought: I now know the final answer

Final Answer: 177 rows in the age column are different.

> Finished chain.

'177 rows in the age column are different.'

对于需要与pandas数据帧进行交互的LLM应用开发者，LangChain中的Pandas Dataframe Agent提供了一种高效且灵活的方法。然而，考虑到安全性问题，开发者应该在使用时谨慎，并确保不执行可能有害的LLM生成的Python代码。

### 10.6.3 PowerBI Dataset Agent的集成

Power BI是一个用于数据可视化和报告的工具，但当需要通过编程方式查询和分析Power BI数据集时，LangChain的PowerBI Dataset Agent为您提供了一个方便的途径。这使得查询变得更自然和人性化，而不是依赖于DAX（数据分析表达式）查询语言。

PowerBI Dataset Agent设计用于与Power BI数据集交互。您可以使用此代理查询数据集，例如描述数据表，查询表中的记录数，或对数据进行多维度的分析。

初始化PowerBI Dataset Agent所用的类： create\_pbi\_agent 和 PowerBIToolkit 来自 langchain.agents.agent\_toolkits。

PowerBIDataset来自 langchain.utilities.powerbi。ChatOpenAI来自langchain.chat\_models。AgentExecutor 来自langchain.agents。

通过执行 pip install azure-identity 命令，安装了 azure.identity 包，以支持Azure的身份验证。代理需要Azure的认证来访问PowerBI数据集。这里使用了 DefaultAzureCredential() 函数，它是 Azure 提供的默认方法，用于获取适当的认证凭据，以便代理可以访问相关数据。

from langchain.agents.agent\_toolkits import create\_pbi\_agent

from langchain.agents.agent\_toolkits import PowerBIToolkit

from langchain.utilities.powerbi import PowerBIDataset

from langchain.chat\_models import ChatOpenAI

from langchain.agents import AgentExecutor

from azure.identity import DefaultAzureCredential

创建 PowerBI Dataset Agent 涉及以下步骤：首先，通过创建一个或多个LLM（如fast\_llm和smart\_llm）来获取文本的嵌入表示。然后，通过实例PowerBIToolkit 来创建一个 Power BI 工具集，用于处理 PowerBI 数据集的任务。接着，通过调用 create\_pbi\_agent 方法来创建 PowerBI Dataset Agent，该代理可以与 PowerBI 数据集进行交互。

在数据集查询方面，有多种操作可用。

首先，通过调用agent\_executor.run("Describe table1")，代理可以提供关于数据表的描述信息。此外，代理还可以在数据表上执行简单的查询操作，比如计算数据表中的记录数，通过调用 agent\_executor.run("How many records are in table1?") 完成。同时，代理也能够执行更复杂的查询操作，比如按维度进行分组计算记录数，例如调用 agent\_executor.run("How many records are there by dimension1 in table2?") 将在 table2 中按 dimension1 维度计算记录数。

fast\_llm = ChatOpenAI(

temperature=0.5, max\_tokens=1000, model\_name="gpt-3.5-turbo", verbose=True

)

smart\_llm = ChatOpenAI(temperature=0, max\_tokens=100, model\_name="gpt-4", verbose=True)

toolkit = PowerBIToolkit(

powerbi=PowerBIDataset(

dataset\_id="<dataset\_id>",

table\_names=["table1", "table2"],

credential=DefaultAzureCredential(),

),

llm=smart\_llm,

)

agent\_executor = create\_pbi\_agent(

llm=fast\_llm,

toolkit=toolkit,

verbose=True,

)

例如，代理可以计算数据表中的记录数：

agent\_executor.run("How many records are in table1?")

我们还可以提供一些自定义的Few-shot Prompts，使模型更容易理解与Power BI数据集相关的问题和回答。提供这些提示可以帮助模型生成更准确的DAX查询。需要注意的是当与Power BI数据集进行互动时，请确保您有适当的权限和凭证。

LangChain的PowerBI Dataset Agent目前仍在积极开发中，可能存在不完善的地方，因此在生产环境中使用时应进行适当的测试。

# fictional example

few\_shots = """

Question: How many rows are in the table revenue?

DAX: EVALUATE ROW("Number of rows", COUNTROWS(revenue\_details))

----

Question: How many rows are in the table revenue where year is not empty?

DAX: EVALUATE ROW("Number of rows", COUNTROWS(FILTER(revenue\_details, revenue\_details[year] <> "")))

----

Question: What was the average of value in revenue in dollars?

DAX: EVALUATE ROW("Average", AVERAGE(revenue\_details[dollar\_value]))

----

"""

toolkit = PowerBIToolkit(

powerbi=PowerBIDataset(

dataset\_id="<dataset\_id>",

table\_names=["table1", "table2"],

credential=DefaultAzureCredential(),

),

llm=smart\_llm,

examples=few\_shots,

)

agent\_executor = create\_pbi\_agent(

llm=fast\_llm,

toolkit=toolkit,

verbose=True,

)

执行查询语句：

agent\_executor.run("What was the maximum of value in revenue in dollars in 2022?")

使用LangChain的PowerBI Dataset Agent与Power BI数据集进行互动为开发者提供了一种新的、自然的方式来查询和分析数据。通过这种方式，开发者可以更加灵活和直观地与数据互动，而不需要深入了解DAX查询语言的复杂性。

## 10.7 Retrievers 集成指南

Retrievers的集成重点在于为开发者提供方便、高效的信息检索工具。首先，Arxiv API Wrapper为那些需要访问和检索学术文献的开发者提供了专门的解决方案，确保他们能够无缝地从Arxiv数据库中获取所需的研究资料。其次，Azure Cognitive Search Wrapper为开发者提供了与Azure平台的深度集成，使其能够高效、准确地从Azure中检索各种信息。最后，Wikipedia API Wrapper则简化了从维基百科中提取内容的流程，让开发者无需深入了解其背后的技术细节，即可轻松获取所需的公开信息。

### 10.7.1 WikipediaRetriever集成

Wikipedia作为一个庞大的在线百科全书，为用户提供了丰富的知识和信息。使用LangChain的WikipediaRetriever，您可以轻松地从Wikipedia获取相关文档并对其进行查询。

首先，为了使用相关功能，安装 wikipedia Python包。

pip install wikipedia

接下来，让我们详细介绍 WikipediaRetriever 的参数配置。默认情况下，lang 参数设置为 "en"，它用于在特定语言的 Wikipedia 部分中进行搜索。load\_max\_docs 参数默认为100，它限制了下载文档的数量。在实验阶段，建议使用较小的数字，因为目前的硬上限是300。此外，load\_all\_available\_meta 参数默认为False，这意味着只下载最重要的字段，包括发布日期、标题和摘要。如果将其设置为True，则会下载其他字段。

运行 WikipediaRetriever 很简单，只需使用 get\_relevant\_documents() 方法，并输入自由文本查询作为参数。例如，如果要查询关于 "HUNTER X HUNTER" 的相关文档，您可以执行以下操作：

from langchain.retrievers import WikipediaRetriever

retriever = WikipediaRetriever()

docs = retriever.get\_relevant\_documents(query="HUNTER X HUNTER")

您可以查看文档的元数据或内容:

print(docs[0].metadata)

print(docs[0].page\_content[:400])

为了使用LangChain进行问题回答，您需要一个OpenAI API密钥。之后，您可以使用 ConversationalRetrievalChain 结合 ChatOpenAI 和WikipediaRetriever进行交互式的问答。例如：

from getpass import getpass

import os

from langchain.chat\_models import ChatOpenAI

from langchain.chains import ConversationalRetrievalChain

OPENAI\_API\_KEY = getpass()

os.environ["OPENAI\_API\_KEY"] = OPENAI\_API\_KEY

model = ChatOpenAI(model\_name="gpt-3.5-turbo")

qa = ConversationalRetrievalChain.from\_llm(model, retriever=retriever)

questions = ["What is Apify?", ...]

对于每个问题，您可以通过适当的代码获取并打印出答案。请注意以下几点事项：首先，在使用API密钥时，务必确保不要在公开的代码中暴露它，以保护您的安全。其次，与Wikipedia进行交互时，可能会受到频率限制或其他限制，因此要注意控制API请求的速率，以避免触发限制。

LangChain的WikipediaRetriever为开发者提供了一个简单而有效的方式，使其可以轻松地与Wikipedia互动，获取相关的文档并进行问题回答。这为开发者带来了巨大的便利，使他们可以更容易地从Wikipedia获取知识并将其应用到自己的应用中。

### 10.7.2 ArxivRetriever集成

arXiv是一个免费的分发服务和开放获取档案，截止2023年8月16日数据，该站点收录了2309934篇学术论文，涵盖了物理、数学、计算机科学、量化生物学、量化金融、统计学、电气工程与系统科学以及经济学等领域。ArxivRetriever是Langchain框架与arXiv的集成工具，可以用于从arXiv检索相关文档。

首先，要使用arXiv集成工具，需要安装arxiv Python包。可以使用命令“!pip install arxiv”来进行安装。

pip install arxiv

ArxivRetriever是Langchain框架提供的一个类，用于与arXiv进行交互。它有一些参数需要设置。其中，load\_max\_docs参数用于限制下载的文档数量。默认情况下是100，但是考虑到下载较多文档可能带来的问题，建议在测试时使用较小的数字。此外，还有一个load\_all\_available\_meta参数，默认为False。当设为False时，只会下载最重要的字段，包括发布日期、标题、作者和摘要。如果将其设为True，还会下载其他字段。

在使用示例中，可以看到如何实例化一个ArxivRetriever对象。在此例中，load\_max\_docs被设置为2，以限制下载的文档数量。然后，通过调用get\_relevant\_documents方法，输入一个查询关键字，可以从arXiv中检索相关的文档。

from langchain.retrievers import ArxivRetriever

retriever = ArxivRetriever(load\_max\_docs=2)

docs = retriever.get\_relevant\_documents(query="1605.08386")

以上代码展示了如何从Arxiv.org检索与query相关的文档。

当集成上述工具到LLM应用中时，开发者可能会遇到多种问题。例如，如何有效地将查询结果从ArxivRetriever传递给OpenAI Embeddings进行嵌入。答案是：通过先使用ArxivRetriever检索文档，再使用OpenAI Embeddings为其生成嵌入，从而实现流畅的工作流程。

### 10.7.3 Azure Cognitive Search集成

Azure Cognitive Search作为一个强大的云搜索服务，能够为开发者带来丰富的搜索体验。本文将详细指导开发者如何在LangChain中集成和使用Azure Cognitive Search。

Azure Cognitive Search（之前称为Azure Search）是一个云搜索服务，为开发者提供了建设丰富搜索体验的基础设施、APIs和工具。无论是文档搜索、在线零售应用还是私有内容的数据探索，搜索都是任何向用户展示文本的应用的基础。

在集成Azure Cognitive Search到LangChain之前，需要根据Azure的官方指南进行相应的配置，同时确保您已经获取了三个重要信息：ACS服务名称、ACS索引名称和API密钥。

一旦完成了配置和信息获取，您可以在LangChain中使用Azure Cognitive Search。首先，从langchain.retrievers模块中导入AzureCognitiveSearchRetriever类，这样您就能够在LangChain中利用Azure Cognitive Search的检索功能。

from langchain.retrievers import AzureCognitiveSearchRetriever

可以将ACS服务名称、索引名称和API密钥设置为环境变量。这样在创建检索器时，可以直接读取这些环境变量。

import os

os.environ["AZURE\_COGNITIVE\_SEARCH\_SERVICE\_NAME"] = "<YOUR\_ACS\_SERVICE\_NAME>"

os.environ["AZURE\_COGNITIVE\_SEARCH\_INDEX\_NAME"] = "<YOUR\_ACS\_INDEX\_NAME>"

os.environ["AZURE\_COGNITIVE\_SEARCH\_API\_KEY"] = "<YOUR\_API\_KEY>"

现在，可以使用上述环境变量创建Azure Cognitive Search检索器，并按需检索相关文档。

retriever = AzureCognitiveSearchRetriever(content\_key="content", top\_k=10)

documents = retriever.get\_relevant\_documents("what is langchain")

其中， top\_k 参数指定了返回的结果数量，可以根据实际需求进行调整。

集成Azure Cognitive Search到LangChain不仅提升了LLM应用的搜索功能，还使开发者能够更加便捷地利用Azure提供的强大搜索能力。希望本指南能够帮助开发者更轻松地在LangChain中实现这一集成。

## 10.8 本章小结

第十章详细探讨了LangChain的集成生态，展现了LangChain如何与各种流行的技术和工具相结合，为用户提供一个更加完整和高效的开发体验。在章节的初步，我们了解了集成的背景，特别是LangChain与LLM之间的集成方式和LangChain的整体集成生态。

深入到具体集成，LLM集成指南为我们展示了如何与Azure OpenAI和Hugging Face Hub相结合，突出了LLM在这两个平台上的功能和优势。随后，聊天模型集成指南讲述了如何与Anthropic、Google PaLM Chat以及Azure上的OpenAI端点进行集成，使得LangChain能够与多种聊天模型无缝对接。

而在向量库集成部分，我们被引导了解了如何将LangChain与Chroma、Pinecone和Milvus这三大主流向量库结合，从而优化和加速向量搜索和操作。进一步，嵌入模型集成指南为我们呈现了与HuggingFaceEmbeddings、Llama-cpp和Cohere的集成方法，确保LangChain能够有效地生成和管理文本嵌入。

在Agent toolkits集成部分，读者可以学习到如何将CSV、Pandas以及PowerBI Dataset Agent与LangChain集成，使数据处理和可视化变得更为轻松。最后，Retrievers集成指南涵盖了WikipediaRetriever、ArxivRetriever和Azure Cognitive Search的集成过程，确保LangChain可以有效地从各种来源检索信息。

总体而言，这一章为我们揭示了LangChain强大的集成能力，无论是与聊天模型、向量库、嵌入模型还是数据处理工具，LangChain都能够提供流畅和高效的集成解决方案，极大地提高了开发者的工作效率。