个人知识库助手项目

一、引言

1、项目背景介绍

在当今信息爆炸的时代，人们面临着海量数据的挑战，如何快速、准确地获取所需信息成为了一个迫切的需求。为了解决这一问题，本项目应运而生，它是一个基于大型语言模型应用开发教程的个人知识库助手。该项目通过精心设计和开发，实现了对大量复杂信息的有效管理和检索，为用户提供了一个强大的信息获取工具。

本项目的开发初衷是利用大型语言模型的强大处理能力，结合用户的实际需求，打造一个能够理解自然语言查询并提供精确答案的智能助手。在这一过程中，开发团队对现有的大模型应用进行了深入分析和研究，进而进行了一系列的封装和完善工作，以确保项目的稳定性和易用性。

2、目标与意义

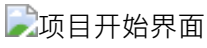
本项目的目的是为了提供一个高效、智能的解决方案，帮助用户在面对海量信息时能够快速定位和获取所需知识，从而提高工作效率和决策质量。通过构建一个个人知识库助手，项目旨在简化信息检索过程，使得用户能够通过自然语言查询，轻松访问和整合分散在不同数据源中的信息。

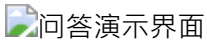
意义方面，该项目具有以下几个关键点：

- **提升信息获取效率**：通过智能检索和问答系统，用户可以迅速找到相关信息，减少了在多个平台或数据库中手动搜索的时间。
- **增强知识管理能力**：项目支持用户构建和维护个人知识库，有助于积累和组织专业知识，形成个人的知识资产。
- **促进决策支持**：通过提供准确、及时的信息，项目能够辅助用户做出更加明智的决策，特别是在需要快速响应的情况下。
- **支持个性化定制**：项目允许用户根据自己的需求和偏好定制知识库，使得信息检索更加个性化和精准。
- **推动技术创新**：项目的开发和应用展示了大型语言模型在信息管理和检索领域的潜力，为未来的技术创新提供了实践案例和灵感。
- **普及智能助手概念**：通过易于使用的界面和部署方式，项目降低了智能助手技术的门槛，使其更加普及和易于接受。

3、主要功能

本项目可以实现基于 Datawhale 的现有项目 README 的知识问答，使用户可以快速了解 Datawhale 现有项目情况。

项目开始界面 

问答演示界面 

二、技术实现

1、环境依赖

1.1 技术资源要求

- **CPU**: Intel 5代处理器（云CPU方面，建议选择 2 核以上的云CPU服务）
- **内存（RAM）**：至少 4 GB
- **操作系统**：Windows、macOS、Linux均可

1.2 项目设置

克隆储存库

```
git clone https://github.com/datawhalechina/llm-universe.git
cd llm-universe/project
```

创建 Conda 环境并安装依赖项

- python>=3.9
- pytorch>=2.0.0

```
# 创建 Conda 环境
conda create -n llm-universe python==3.9.0
# 激活 Conda 环境
conda activate llm-universe
# 安装依赖项
pip install -r requirements.txt
```

1.3 项目运行

- 启动服务为本地 API

```
# Linux 系统
cd project/serve
uvicorn api:app --reload
```

```
# Windows 系统
cd project/serve
python api.py
```

- 运行项目

```
cd llm-universe/project/serve
python run_gradio.py -model_name='chatglm_std' -embedding_model='m3e' -
db_path='../../data_base/knowledge_db' -persist_path='../../data_base/vector_db'
```

2、开发流程简述

2.1 当前的项目版本及未来规划

- **当前版本：**0.2.0(更新于2024.3.17)
    - **更新内容**
        - ☑ 新增 m3e embedding
        - ☑ 新增知识库内容
        - ☑ 新增 Datawhale 的所有 Md 的总结
        - ☑ 修复 gradio 显示错误
    - **目前支持的模型**
        - OpenAi
            - ☑ gpt-3.5-turbo
            - ☑ gpt-3.5-turbo-16k-0613
            - ☑ gpt-3.5-turbo-0613
            - ☑ gpt-4
            - ☑ gpt-4-32k
        - 文心一言
            - ☑ ERNIE-Bot
            - ☑ ERNIE-Bot-4
            - ☑ ERNIE-Bot-turbo
        - 讯飞星火
            - ☑ Spark-1.5
            - ☑ Spark-2.0
        - 智谱 AI
            - ☑ chatglm_pro
            - ☑ chatglm_std
            - ☑ chatglm_lite
- **未来规划**
    - ☐ 更新 智谱Ai embedding

2.2 核心Idea

核心是针对四种大模型 API 实现了底层封装，基于 Langchain 搭建了可切换模型的检索问答链，并实现 API 以及 Gradio 部署的个人轻量大模型应用。

2.3 使用的技术栈

本项目为一个基于大模型的个人知识库助手，基于 LangChain 框架搭建，核心技术包括 LLM API 调用、向量数据库、检索问答链等。项目整体架构如下：

 整体框架

如上，本项目从底向上依次分为 LLM 层、数据层、数据库层、应用层与服务层。

① LLM 层主要基于四种流行 LLM API 进行了 LLM 调用封装，支持用户以统一的入口、方式来访问不同的模型，支持随时进行模型的切换；

② 数据层主要包括个人知识库的源数据以及 Embedding API，源数据经过 Embedding 处理可以被向量数据库使用；

③ 数据库层主要为基于个人知识库源数据搭建的向量数据库，在本项目中我们选择了 Chroma；

④ 应用层为核心功能的最顶层封装，我们基于 LangChain 提供的检索问答链基类进行了进一步封装，从而支持不同模型切换以及便捷实现基于数据库的检索问答；

⑤ 最顶层为服务层，我们分别实现了 Gradio 搭建 Demo 与 FastAPI 组建 API 两种方式来支持本项目的服务访问。

三、应用详解

1、核心架构

llm-universe 个人知识库助手地址：

https://github.com/datawhalechina/llm-universe/tree/main

该项目是个典型的RAG项目，通过langchain+LLM实现本地知识库问答，建立了全流程可使用开源模型实现的本地知识库对话应用。目前已经支持使用 ***ChatGPT***，***星火spark模型***，***文心大模型***，***智谱GLM*** 等大语言模型的接入。该项目实现原理和一般 RAG 项目一样，如前文和下图所示：

整个 RAG 过程包括如下操作：

1.用户提出问题 Query

2.加载和读取知识库文档

3.对知识库文档进行分割

4.对分割后的知识库文本向量化并存入向量库建立索引

5.对问句 Query 向量化

6.在知识库文档向量中匹配出与问句 Query 向量最相似的 top k 个

7.匹配出的知识库文本文本作为上下文 Context 和问题一起添加到 prompt 中

8.提交给 LLM 生成回答 Answer

可以大致分为索引，检索和生成三个阶段，这三个阶段将在下面小节配合该 llm-universe 知识库助手项目进行拆解。

2、索引-indexing

本节讲述该项目 llm-universe 个人知识库助手：创建知识库并加载文件-读取文件-**文本分割**(Text splitter) ，知识库**文本向量化**(embedding)以及存储到**向量数据库**的实现，

其中**加载文件**：这是读取存储在本地的知识库文件的步骤。**读取文件**：读取加载的文件内容，通常是将其转化为文本格式 。**文本分割(Text splitter)**：按照一定的规则(例如段落、句子、词语等)将文本分割。**文本向量化：**这通常涉及到 NLP 的特征抽取，该项目通过本地 m3e 文本嵌入模型，openai，zhipuai 开源 api 等方法将分割好的文本转化为数值向量并存储到向量数据库

2.1 知识库搭建-加载和读取

该项目llm-universe个人知识库助手选用 Datawhale 一些经典开源课程、视频（部分）作为示例，具体包括：

- 《机器学习公式详解》PDF版本
- 《面向开发者的 LLM 入门教程 第一部分 Prompt Engineering》md版本
- 《强化学习入门指南》MP4版本
- 以及datawhale总仓库所有开源项目的readme https://github.com/datawhalechina

这些知识库源数据放置在 **../../data_base/knowledge_db** 目录下，用户也可以自己存放自己其他的文件。

1.下面讲一下如何获取 DataWhale 总仓库的所有开源项目的 readme ，用户可以通过先运行 **project/database/test_get_all_repo.py** 文件，用来获取 Datawhale 总仓库所有开源项目的 readme，代码如下：

```
import json
import requests
import os
import base64
import loguru
from dotenv import load_dotenv
# 加载环境变量
load_dotenv()
# 从环境变量中获取TOKEN
TOKEN = os.getenv('TOKEN')
# 定义获取组织仓库的函数
def get_repos(org_name, token, export_dir):
    headers = {
        'Authorization': f'token {token}',
    }
    url = f'https://api.github.com/orgs/{org_name}/repos'
    response = requests.get(url, headers=headers, params={'per_page': 200, 'page':
0})
    if response.status_code == 200:
        repos = response.json()
        loguru.logger.info(f'Fetched {len(repos)} repositories for {org_name}.')
        # 使用 export_dir 确定保存仓库名的文件路径
        repositories_path = os.path.join(export_dir, 'repositories.txt')
        with open(repositories_path, 'w', encoding='utf-8') as file:
            for repo in repos:
                file.write(repo['name'] + '\n')
```

```
        return repos
    else:
        loguru.logger.error(f"Error fetching repositories:
{response.status_code}")
        loguru.logger.error(response.text)
        return []
# 定义拉取仓库README文件的函数
def fetch_repo_readme(org_name, repo_name, token, export_dir):
    headers = {
        'Authorization': f'token {token}',
    }
    url = f'https://api.github.com/repos/{org_name}/{repo_name}/readme'
    response = requests.get(url, headers=headers)
    if response.status_code == 200:
        readme_content = response.json()['content']
        # 解码base64内容
        readme_content = base64.b64decode(readme_content).decode('utf-8')
        # 使用 export_dir 确定保存 README 的文件路径
        repo_dir = os.path.join(export_dir, repo_name)
        if not os.path.exists(repo_dir):
            os.makedirs(repo_dir)
        readme_path = os.path.join(repo_dir, 'README.md')
        with open(readme_path, 'w', encoding='utf-8') as file:
            file.write(readme_content)
    else:
        loguru.logger.error(f"Error fetching README for {repo_name}:
{response.status_code}")
        loguru.logger.error(response.text)
# 主函数
if __name__ == '__main__':
    # 配置组织名称
    org_name = 'datawhalechina'
    # 配置 export_dir
    export_dir = "../../database/readme_db"  # 请替换为实际的目录路径
    # 获取仓库列表
    repos = get_repos(org_name, TOKEN, export_dir)
    # 打印仓库名称
    if repos:
        for repo in repos:
            repo_name = repo['name']
            # 拉取每个仓库的README
            fetch_repo_readme(org_name, repo_name, TOKEN, export_dir)
    # 清理临时文件夹
    # if os.path.exists('temp'):
    #     shutil.rmtree('temp')
```

默认会把这些readme文件放在同目录database下的readme_db文件。其中这些readme文件含有不少无关信息，即再运行**project/database/text_summary_readme.py文件**可以调用大模型生成每个readme文件的摘要并保存到上述知识库目录../../data_base/knowledge_db /readme_summary文件夹中，****。代码如下：

```
import os
from dotenv import load_dotenv
import openai
from test_get_all_repo import get_repos
from bs4 import BeautifulSoup
import markdown
import re
import time
# Load environment variables
load_dotenv()
TOKEN = os.getenv('TOKEN')
# Set up the OpenAI API client
openai_api_key = os.environ["OPENAI_API_KEY"]

# 过滤文本中链接防止大语言模型风控
def remove_urls(text):
    # 正则表达式模式，用于匹配URL
    url_pattern = re.compile(r'https?://[^\s]*')
    # 替换所有匹配的URL为空字符串
    text = re.sub(url_pattern, '', text)
    # 正则表达式模式，用于匹配特定的文本
    specific_text_pattern = re.compile(r'扫描下方二维码关注公众号|提取码|关注|科学上网|回复关键词|侵权|版权|致谢|引用|LICENSE'
                                       r'|组队打卡|任务打卡|组队学习的那些事|学习周期|开源内容|打卡|组队学习|链接')
    # 替换所有匹配的特定文本为空字符串
    text = re.sub(specific_text_pattern, '', text)
    return text

# 抽取md中的文本
def extract_text_from_md(md_content):
    # Convert Markdown to HTML
    html = markdown.markdown(md_content)
    # Use BeautifulSoup to extract text
    soup = BeautifulSoup(html, 'html.parser')

    return remove_urls(soup.get_text())

def generate_llm_summary(repo_name, readme_content,model):
    prompt = f"1：这个仓库名是 {repo_name}. 此仓库的readme全部内容是: {readme_content}\
                2:请用约200以内的中文概括这个仓库readme的内容,返回的概括格式要求：这个仓库名是...,这仓库内容主要是..."
    openai.api_key = openai_api_key
    # 具体调用
    messages = [{"role": "system", "content": "你是一个人工智能助手"},
                {"role": "user", "content": prompt}]
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
    )
    return response.choices[0].message["content"]
```

```
def main(org_name,export_dir,summary_dir,model):
    repos = get_repos(org_name, TOKEN, export_dir)

    # Create a directory to save summaries
    os.makedirs(summary_dir, exist_ok=True)

    for id, repo in enumerate(repos):
        repo_name = repo['name']
        readme_path = os.path.join(export_dir, repo_name, 'README.md')
        print(repo_name)
        if os.path.exists(readme_path):
            with open(readme_path, 'r', encoding='utf-8') as file:
                readme_content = file.read()
            # Extract text from the README
            readme_text = extract_text_from_md(readme_content)
            # Generate a summary for the README
            # 访问受限，每min一次
            time.sleep(60)
            print('第' + str(id) + '条' + 'summary开始')
            try:
                summary = generate_llm_summary(repo_name, readme_text,model)
                print(summary)
                # Write summary to a Markdown file in the summary directory
                summary_file_path = os.path.join(summary_dir, f"
{repo_name}_summary.md")
                with open(summary_file_path, 'w', encoding='utf-8') as
summary_file:
                    summary_file.write(f"# {repo_name} Summary\n\n")
                    summary_file.write(summary)
            except openai.OpenAIError as e:
                summary_file_path = os.path.join(summary_dir, f"
{repo_name}_summary风控.md")
                with open(summary_file_path, 'w', encoding='utf-8') as
summary_file:
                    summary_file.write(f"# {repo_name} Summary风控\n\n")
                    summary_file.write("README内容风控。\n")
                print(f"Error generating summary for {repo_name}: {e}")
                # print(readme_text)
        else:
            print(f"文件不存在: {readme_path}")
            # If README doesn't exist, create an empty Markdown file
            summary_file_path = os.path.join(summary_dir, f"{repo_name}_summary不
存在.md")
            with open(summary_file_path, 'w', encoding='utf-8') as summary_file:
                summary_file.write(f"# {repo_name} Summary不存在\n\n")
                summary_file.write("README文件不存在。\n")
if __name__ == '__main__':
    # 配置组织名称
    org_name = 'datawhalechina'
    # 配置 export_dir
    export_dir = "../database/readme_db"  # 请替换为实际readme的目录路径
    summary_dir="../../data_base/knowledge_db/readme_summary"# 请替换为实际readme的
概括的目录路径
    model="gpt-3.5-turbo"  #deepseek-chat,gpt-3.5-turbo,moonshot-v1-8k
```

```
    main(org_name,export_dir,summary_dir,model)
```

其中 **extract_text_from_md()** 函数用来抽取 md 文件中的文本， **remove_urls()** 函数过滤了 readme 文本中的一些网页链接以及过滤了可能引起大模型风控一些词汇。接着调用 generate_llm_summary() 让大模型生成每个 readme 的概括。

2.在上述知识库构建完毕之后，**../../data_base/knowledge_db** 目录下就有了 Datawhale 开源的所有项目的 readme 概括的 md 文件，以及《机器学习公式详解》PDF版本，《面向开发者的 LLM 入门教程 第一部分 Prompt Engineering》md版本，《强化学习入门指南》MP4版本等文件。

其中有 mp4 格式，md 格式，以及 pdf 格式，对这些文件的加载方式，该项目将代码放在了 **project/database/create_db.py文件** 下，部分代码如下。其中 pdf 格式文件用 PyMuPDFLoader 加载器，md 格式文件用UnstructuredMarkdownLoader加载器。要注意的是其实数据处理是一件非常复杂和业务个性化的事，如pdf文件中包含图表，图片和文字以及不同层次标题，这些都需要根据业务进行精细化处理。具体操作可以关注**第二部分的高阶RAG教程技术**进行自行摸索：

```
from langchain.document_loaders import UnstructuredFileLoader
from langchain.document_loaders import UnstructuredMarkdownLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.document_loaders import PyMuPDFLoader
from langchain.vectorstores import Chroma
# 首先实现基本配置

DEFAULT_DB_PATH = "../../data_base/knowledge_db"
DEFAULT_PERSIST_PATH = "../../data_base/vector_db"
...
...
...
def file_loader(file, loaders):
    if isinstance(file, tempfile._TemporaryFileWrapper):
        file = file.name
    if not os.path.isfile(file):
        [file_loader(os.path.join(file, f), loaders) for f in  os.listdir(file)]
        return
    file_type = file.split('.')[-1]
    if file_type == 'pdf':
        loaders.append(PyMuPDFLoader(file))
    elif file_type == 'md':
        pattern = r"不存在|风控"
        match = re.search(pattern, file)
        if not match:
            loaders.append(UnstructuredMarkdownLoader(file))
    elif file_type == 'txt':
        loaders.append(UnstructuredFileLoader(file))
    return
...
...
```

2.2 文本分割和向量化

文本分割和向量化操作，在整个 RAG 流程中是必不可少的。需要将上述载入的知识库分本或进行 token 长度进行分割，或者进行语义模型进行分割。该项目利用 Langchain 中的文本分割器根据 chunk_size (块大小)和 chunk_overlap (块与块之间的重叠大小)进行分割。

- chunk_size 指每个块包含的字符或 Token（如单词、句子等）的数量
- chunk_overlap 指两个块之间共享的字符数量，用于保持上下文的连贯性，避免分割丢失上下文信息

1. 可以设置一个最大的 Token 长度，然后根据这个最大的 Token 长度来切分文档。这样切分出来的文档片段是一个一个均匀长度的文档片段。而片段与片段之间的一些重叠的内容，能保证检索的时候能够检索到相关的文档片段。这部分文本分割代码也在 **project/database/create_db.py** 文件，该项目采用了 langchain 中 RecursiveCharacterTextSplitter 文本分割器进行分割。代码如下：

```
......
def create_db(files=DEFAULT_DB_PATH, persist_directory=DEFAULT_PERSIST_PATH,
embeddings="openai"):
    """
    该函数用于加载 PDF 文件，切分文档，生成文档的嵌入向量，创建向量数据库。

    参数:
    file: 存放文件的路径。
    embeddings: 用于生产 Embedding 的模型

    返回:
    vectordb: 创建的数据库。
    """
    if files == None:
        return "can't load empty file"
    if type(files) != list:
        files = [files]
    loaders = []
    [file_loader(file, loaders) for file in files]
    docs = []
    for loader in loaders:
        if loader is not None:
            docs.extend(loader.load())
    # 切分文档
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=500, chunk_overlap=150)
    split_docs = text_splitter.split_documents(docs)
    ....
    ....
    ....此处省略了其他代码
    ....
    return vectordb
...........
```

2. 而在切分好知识库文本之后，需要对文本进行 **向量化** 。该项目在**project/embedding/call_embedding.py**，文本嵌入方式可选本地 m3e 模型，以及调用 openai 和 zhipuai 的 api 的方式进行文本嵌入。代码如下：

```
import os
import sys

sys.path.append(os.path.dirname(os.path.dirname(__file__)))
sys.path.append(r"../../")
from embedding.zhipuai_embedding import ZhipuAIEmbeddings
from langchain.embeddings.huggingface import HuggingFaceEmbeddings
from langchain.embeddings.openai import OpenAIEmbeddings
from llm.call_llm import parse_llm_api_key


def get_embedding(embedding: str, embedding_key: str = None, env_file: str =
None):
    if embedding == 'm3e':
        return HuggingFaceEmbeddings(model_name="moka-ai/m3e-base")
    if embedding_key == None:
        embedding_key = parse_llm_api_key(embedding)
    if embedding == "openai":
        return OpenAIEmbeddings(openai_api_key=embedding_key)
    elif embedding == "zhipuai":
        return ZhipuAIEmbeddings(zhipuai_api_key=embedding_key)
    else:
        raise ValueError(f"embedding {embedding} not support ")
```

2.3 向量数据库

在对知识库文本进行分割和向量化后，就需要定义一个向量数据库用来存放文档片段和对应的向量表示了，在向量数据库中，数据被表示为向量形式，每个向量代表一个数据项。这些向量可以是数字、文本、图像或其他类型的数据。

向量数据库使用高效的索引和查询算法来加速向量数据的存储和检索过程。该项目选择 chromadb 向量数据库（类似的向量数据库还有 faiss 等)。定义向量库对应的代码也在 **project/database/create_db.py** 文件中，persist_directory 即为本地持久化地址，vectordb.persist() 操作可以持久化向量数据库到本地，后续可以再次载入本地已有的向量库。完整的文本分割，获取向量化，并且定义向量数据库代码如下：

```
def create_db(files=DEFAULT_DB_PATH, persist_directory=DEFAULT_PERSIST_PATH,
embeddings="openai"):
    """
    该函数用于加载 PDF 文件，切分文档，生成文档的嵌入向量，创建向量数据库。

    参数:
    file: 存放文件的路径。
    embeddings: 用于生产 Embedding 的模型

    返回:
    vectordb: 创建的数据库。
    """
    if files == None:
        return "can't load empty file"
```

```
    if type(files) != list:
        files = [files]
    loaders = []
    [file_loader(file, loaders) for file in files]
    docs = []
    for loader in loaders:
        if loader is not None:
            docs.extend(loader.load())
    # 切分文档
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=500, chunk_overlap=150)
    split_docs = text_splitter.split_documents(docs)
    if type(embeddings) == str:
        embeddings = get_embedding(embedding=embeddings)
    # 定义持久化路径
    persist_directory = '../../data_base/vector_db/chroma'
    # 加载数据库
    vectordb = Chroma.from_documents(
    documents=split_docs,
    embedding=embeddings,
    persist_directory=persist_directory  # 允许我们将persist_directory目录保存到磁盘
上
    )

    vectordb.persist()
    return vectordb
```

3、检索-Retriver和生成-Generator

本节进入了 RAG 的检索和生成阶段，即对问句 Query 向量化后在知识库文档向量中匹配出与问句 Query 向量最相似的 top k 个片段，匹配出的知识库文本文本作为上下文 Context 和问题一起添加到 prompt 中，然后提交给 LLM 生成回答 Answer。下面将根据 llm_universe 个人知识库助手进行讲解。

3.1 向量数据库检索

通过上一章节文本的分割向量化以及构建向量数据库索引，接下去就可以利用向量数据库来进行高效的检索。向量数据库是一种用于有效搜索大规模高维向量空间中相似度的库，能够在大规模数据集中快速找到与给定 query 向量最相似的向量。如下面示例所示：

```
question="什么是机器学习"
Copy to clipboardErrorCopied
sim_docs = vectordb.similarity_search(question,k=3)
print(f"检索到的内容数：{len(sim_docs)}")
```

```
检索到的内容数：3
```

```
for i, sim_doc in enumerate(sim_docs):
    print(f"检索到的第{i}个内容: \n{sim_doc.page_content[:200]}", end="\n----------
----\n")
```

```
检索到的第0个内容:
导，同时也能体会到这三门数学课在机器学习上碰撞产生的“数学之美”。
1.1
引言
本节以概念理解为主，在此对“算法”和“模型”作补充说明。“算法”是指从数据中学得“模型”的具
体方法，例如后续章节中将会讲述的线性回归、对数几率回归、决策树等。“算法”产出的结果称为“模
型”，
通常是具体的函数或者可抽象地看作为函数，例如一元线性回归算法产出的模型即为形如 f(x) = wx
+ b

的一元一次函数。
--------------

检索到的第1个内容:
模型：机器学习的一般流程如下：首先收集若干样本（假设此时有 100 个），然后将其分为训练样本
（80 个）和测试样本（20 个），其中 80 个训练样本构成的集合称为“训练集”，20 个测试样本构
成的集合
称为“测试集”，接着选用某个机器学习算法，让其在训练集上进行“学习”（或称为“训练”），然后产
出

得到“模型”（或称为“学习器”），最后用测试集来测试模型的效果。执行以上流程时，表示我们已经
默
--------------

检索到的第2个内容:
→_→
欢迎去各大电商平台选购纸质版南瓜书《机器学习公式详解》
←_←
第 1 章
绪论
本章作为“西瓜书”的开篇，主要讲解什么是机器学习以及机器学习的相关数学符号，为后续内容作
铺垫，并未涉及复杂的算法理论，因此阅读本章时只需耐心梳理清楚所有概念和数学符号即可。此外，
在
阅读本章前建议先阅读西瓜书目录前页的《主要符号表》，它能解答在阅读“西瓜书”过程中产生的大部
分对数学符号的疑惑。
本章也作为
```

3.2 大模型llm的调用

这里以该项目 **project/qa_chain/model_to_llm.py** 代码为例，在 **project/llm/** 的目录文件夹下分别定义了 ***星火spark***，***智谱glm***，***文心llm***等开源模型api调用的封装，并在 **project/qa_chain/model_to_llm.py** 文件中导入了这些模块，可以根据用户传入的模型名字进行调用 llm。代码如下：

```
def model_to_llm(model:str=None, temperature:float=0.0, appid:str=None,
api_key:str=None,Spark_api_secret:str=None,Wenxin_secret_key:str=None):
        """
        星火：model,temperature,appid,api_key,api_secret
        百度问心：model,temperature,api_key,api_secret
        智谱：model,temperature,api_key
        OpenAI：model,temperature,api_key
        """
        if model in ["gpt-3.5-turbo", "gpt-3.5-turbo-16k-0613", "gpt-3.5-turbo-
0613", "gpt-4", "gpt-4-32k"]:
            if api_key == None:
                api_key = parse_llm_api_key("openai")
            llm = ChatOpenAI(model_name = model, temperature = temperature ,
openai_api_key = api_key)
        elif model in ["ERNIE-Bot", "ERNIE-Bot-4", "ERNIE-Bot-turbo"]:
            if api_key == None or Wenxin_secret_key == None:
                api_key, Wenxin_secret_key = parse_llm_api_key("wenxin")
            llm = Wenxin_LLM(model=model, temperature = temperature,
api_key=api_key, secret_key=Wenxin_secret_key)
        elif model in ["Spark-1.5", "Spark-2.0"]:
            if api_key == None or appid == None and Spark_api_secret == None:
                api_key, appid, Spark_api_secret = parse_llm_api_key("spark")
            llm = Spark_LLM(model=model, temperature = temperature, appid=appid,
api_secret=Spark_api_secret, api_key=api_key)
        elif model in ["chatglm_pro", "chatglm_std", "chatglm_lite"]:
            if api_key == None:
                api_key = parse_llm_api_key("zhipuai")
            llm = ZhipuAILLM(model=model, zhipuai_api_key=api_key, temperature =
temperature)
        else:
            raise ValueError(f"model{model} not support!!!")
        return llm
```

3.3 prompt和构建问答链

接下去来到了最后一步，设计完基于知识库问答的 prompt，就可以结合上述检索和大模型调用进行答案的生成。构建 prompt 的格式如下，具体可以根据自己业务需要进行修改：

```
from langchain.prompts import PromptTemplate

# template = """基于以下已知信息，简洁和专业的来回答用户的问题。
#             如果无法从中得到答案，请说 "根据已知信息无法回答该问题" 或 "没有提供足够的
相关信息"，不允许在答案中添加编造成分。
#             答案请使用中文。
#             总是在回答的最后说“谢谢你的提问！”。
# 已知信息：{context}
# 问题: {question}"""
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，不要试图编
造答
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}
有用的回答:"""

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

# 运行 chain
```

并且构建问答链：创建检索 QA 链的方法 RetrievalQA.from_chain_type() 有如下参数：

- llm：指定使用的 LLM
- 指定 chain type : RetrievalQA.from_chain_type(chain_type="map_reduce")，也可以利用load_qa_chain()方法指定chain type。
- 自定义 prompt ：通过在RetrievalQA.from_chain_type()方法中，指定chain_type_kwargs参数，而该参数：chain_type_kwargs = {"prompt": PROMPT}
- 返回源文档：通过RetrievalQA.from_chain_type()方法中指定：return_source_documents=True参数；也可以使用RetrievalQAWithSourceChain()方法，返回源文档的引用（坐标或者叫主键、索引）

```
# 自定义 QA 链
self.qa_chain = RetrievalQA.from_chain_type(llm=self.llm,
                                            retriever=self.retriever,
                                            return_source_documents=True,
                                            chain_type_kwargs=
{"prompt":self.QA_CHAIN_PROMPT})
```

问答链效果如下：基于召回结果和 query 结合起来构建的 prompt 效果

```
question_1 = "什么是南瓜书？"
question_2 = "王阳明是谁？"Copy to clipboardErrorCopied
```

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

```
大模型+知识库后回答 question_1 的结果：
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。谢谢你的提问！
```

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

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

```
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！
```

上述详细不带记忆的检索问答链代码都在该项目：**project/qa_chain/QA_chain_self.py** 中，此外该项目还实现了带记忆的检索问答链，两种自定义检索问答链内部实现细节类似，只是调用了不同的 LangChain 链。完整带记忆的检索问答链条代码 **project/qa_chain/Chat_QA_chain_self.py** 如下：

```
from langchain.prompts import PromptTemplate
from langchain.chains import RetrievalQA
from langchain.vectorstores import Chroma
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI

from qa_chain.model_to_llm import model_to_llm
from qa_chain.get_vectordb import get_vectordb


class Chat_QA_chain_self:
    """
    带历史记录的问答链
    - model：调用的模型名称
    - temperature：温度系数，控制生成的随机性
    - top_k：返回检索的前k个相似文档
    - chat_history：历史记录，输入一个列表，默认是一个空列表
    - history_len：控制保留的最近 history_len 次对话
    - file_path：建库文件所在路径
    - persist_path：向量数据库持久化路径
    - appid：星火
    - api_key：星火、百度文心、OpenAI、智谱都需要传递的参数
    - Spark_api_secret：星火秘钥
    - Wenxin_secret_key：文心秘钥
    - embeddings：使用的embedding模型
    - embedding_key：使用的embedding模型的秘钥（智谱或者OpenAI）
    """
    def __init__(self,model:str, temperature:float=0.0, top_k:int=4,
chat_history:list=[], file_path:str=None, persist_path:str=None, appid:str=None,
api_key:str=None, Spark_api_secret:str=None,Wenxin_secret_key:str=None, embedding
= "openai",embedding_key:str=None):
        self.model = model
        self.temperature = temperature
        self.top_k = top_k
        self.chat_history = chat_history
        #self.history_len = history_len
        self.file_path = file_path
        self.persist_path = persist_path
        self.appid = appid
```

```
        self.api_key = api_key
        self.Spark_api_secret = Spark_api_secret
        self.Wenxin_secret_key = Wenxin_secret_key
        self.embedding = embedding
        self.embedding_key = embedding_key


        self.vectordb = get_vectordb(self.file_path, self.persist_path,
self.embedding,self.embedding_key)


    def clear_history(self):
        "清空历史记录"
        return self.chat_history.clear()


    def change_history_length(self,history_len:int=1):
        """
        保存指定对话轮次的历史记录
        输入参数：
        - history_len ：控制保留的最近 history_len 次对话
        - chat_history：当前的历史对话记录
        输出：返回最近 history_len 次对话
        """
        n = len(self.chat_history)
        return self.chat_history[n-history_len:]


    def answer(self, question:str=None,temperature = None, top_k = 4):
        """"
        核心方法，调用问答链
        arguments:
        - question：用户提问
        """

        if len(question) == 0:
            return "", self.chat_history

        if len(question) == 0:
            return ""

        if temperature == None:
            temperature = self.temperature

        llm = model_to_llm(self.model, temperature, self.appid, self.api_key,
self.Spark_api_secret,self.Wenxin_secret_key)

        #self.memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True)

        retriever = self.vectordb.as_retriever(search_type="similarity",
                                               search_kwargs={'k': top_k})  #默认
similarity, k=4
```

```
        qa = ConversationalRetrievalChain.from_llm(
            llm = llm,
            retriever = retriever
        )

        #print(self.llm)
        result = qa({"question": question,"chat_history": self.chat_history})
#result里有question、chat_history、answer
        answer =  result['answer']
        self.chat_history.append((question,answer)) #更新历史记录

        return self.chat_history  #返回本次回答和更新后的历史记录
```

3.总结与展望

3.1 个人知识库关键点总结

该实例是一个基于大型语言模型（LLM）的个人知识库助手项目，通过智能检索和问答系统，帮助用户快速定位和获取与DATa whale相关的知识。以下是该项目的关键点：

关键点一

1. 项目使用多种方法完成Datawhale中所有md文件的抽取与概括，生成对应的知识库。在完成md文件抽取与概括的同时，还是用相应的方法完成readme文本中网页链接和可能引起大模型风控词汇的过滤；
2. 项目利用Langchain中的文本切割器完成知识库向量化操作前的文本分割，向量数据库使用高效的索引和查询算法来加速向量数据的存储和检索过程，快速的完成个人知识库数据建立与使用。

关键点二

项目对不同的API进行了底层封装，用户可以避免复杂的封装细节，直接调用相应的大语言模型即可。

3.2 未来发展方向

1. 用户体验升级：支持用户自主上传并建立个人知识库，构建属于自己的专属个人知识库助手；
2. 模型架构升级：从 REG 的普遍架构升级到 Multi-Agent 的多智体框架；
3. 功能优化升级：对现有结构内的检索函数进行优化，提高个人知识库的检索准确性。

4.致谢

在此感谢散师傅的项目中爬虫及总结部分。