10758 words | 27 minutes

LLM application case: Human relations and social skills large model system - Tianji

introduction

In China, toasting at the table is not just a simple toast, it is also a profound social art, containing rich cultural traditions and delicate social skills. In various banquets and gatherings, how to properly toast can not only show the host's enthusiasm and politeness, but also deepen the relationship between the host and the guests and promote a more harmonious relationship between the two parties. But for many people, the cumbersome etiquette and difficult degree of toasting at the table often make people feel distressed and headache.

Don't worry, *Tianji* , the worldly-wise assistant, is now online to help us solve all the problems of toasting at the table. From preparing toasts to toasting blessings, from round arrangement to return toast strategy, it will provide us with a series of guidelines and suggestions to help us easily deal with various occasions and show our style and wisdom. Let's enter the world of Tianji, the worldly-wise assistant!

Project Background

Tianji is a free, non-commercial artificial intelligence system developed by SocialAI. We can use it to perform tasks involving traditional social skills, such as how to toast, how to say nice things, how to deal with things, etc., to improve your emotional intelligence and "core competitiveness".

Laishier AI has built and open-sourced common large-model application examples, involving prompts, agents, knowledge bases, model training and other technologies.

Goal and significance

In the development of artificial intelligence, we have been exploring how to make machines more intelligent, how to make them not only understand complex data and logic, but also understand human emotions, culture and even the ways of the world. This pursuit is not only a technological breakthrough, but also a tribute to human wisdom.

The Tianji team aims to explore a variety of technical routes that combine large models with the laws of human nature and build intelligent applications that use AI to serve life. This is a key step towards general artificial intelligence (AGI) and the only way to achieve deeper communication between machines and humans.

We firmly believe that only human relationships and worldly wisdom are the core technology of future AI, and only AI that knows how to handle things will have the opportunity to move towards AGI. Let us work together to witness the advent of general artificial intelligence. —— "The secret of heaven cannot be leaked."

The main function

In interpersonal communication, we often encounter some awkward situations, such as silence at the table, being at a loss when toasting, and how to send sincere blessings. In order to solve these embarrassments and enhance communication skills, the Tianji team developed an intelligent assistant application. The main functions of the application include toast words, etiquette for entertaining guests, gift suggestions, and blessing text generation. We can choose the corresponding function according to different scenarios and needs to get inspiration and suggestions for the big model.

Some examples of functions are as follows:

Figure 1 : Resolving embarrassment

Figure 2 : Toasting

Figure 3 : Treating guests

Figure 4 : Interpersonal communication

Figure 5 : Gift giving

Figure 6 : Sending blessings

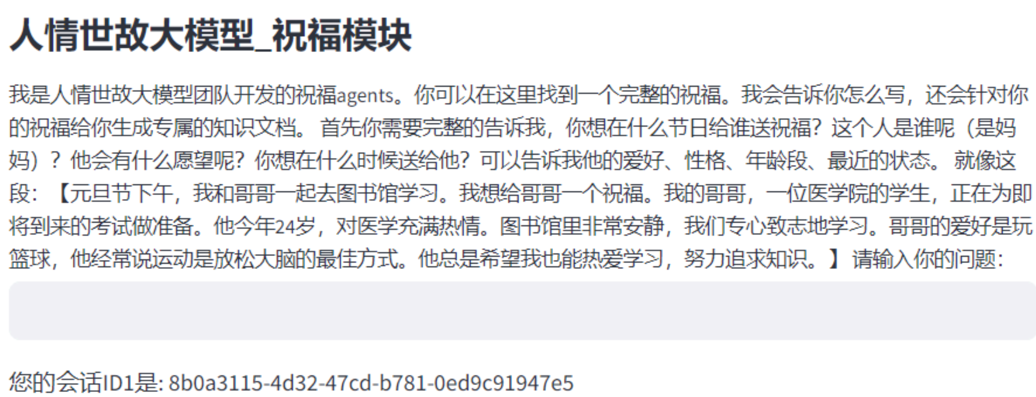
Technical realization

We can use the following four methods (any of which can be implemented) to achieve this:

You can go and experience it: [Tianji Experience]

- Prompt (including AI games): mainly uses the built-in system prompt to conduct dialogue based on the capabilities of the large model.

- Agent (MetaGPT, etc.): Use the Agent architecture to get richer, more customized and detailed answers.
- Knowledge base: Use the vector database to directly search for social rules (such as how to drink at the dinner table).
- Model training: Based on different excellent model bases, fine-tune Lora or fine-tune the whole amount of data after accumulating a large amount of data. (Currently, Tianji only has the function of sending blessings)



Environmental requirements

Computing resource requirements

There are four technical routes involved in Tianji: Prompt, Agent, knowledge base, and model training. Among them, Prompt and Agent only need to configure the large model key, do not need a graphics card, and can be run using a regular laptop.

Technical route	Computer Configuration
Prompt	Only need to configure the large model KEY
Agent	Only need to configure the large model KEY
knowledge base	/
Model Training	Based on InternLM2

Development environment requirements

- Operating system: Windows, Linux, Mac

- IDE: PyCharm (or VSCode), Anaconda
- Need to use the large model "APIKEY"

Environment configuration method

shell

```shell
克隆仓库：git clone https://github.com/SocialAI-tianji/Tianji.git
创建虚拟环境：conda create -n TJ python=3.11
激活环境：conda activate TJ
安装环境依赖：pip install -r requirements.txt -i https://pypi.tuna.tsinghua.
```

Create a .env file in the project and fill in your big model key

env

```env
OPENAI_API_KEY=
OPENAI_API_BASE=
ZHIPUAI_API_KEY=
BAIDU_API_KEY=
OPENAI_API_MODEL=
HF_HOME='./cache/'
HF_ENDPOINT = 'https://hf-mirror.com'
HF_TOKEN=
```

Development Process Overview

Current project version and future plans

Current version: Updated Prompt, Agent, knowledge base, model fine-tuning (based on InternLM2)

Future planning: The project will mount huggingface, aistudio, openxlab, modelscope, etc.

Core Idea

Core concept: Combine the powerful processing power of large language models with a deep understanding of human nature to help users improve their emotional intelligence. By analyzing and simulating various scenarios in daily interactions, this method can provide real-time feedback and guidance to help users better understand the emotions and perspectives of others, thereby improving interpersonal skills.

Innovation: Combining advanced AI technology with the cultivation of human emotional intelligence. The computing power of the large model can process and analyze large amounts of interpersonal communication data, while the application of the rules of human nature ensures the practical utility of this technology in improving personal emotional intelligence. This combination not only improves the model's ability to understand and predict human emotions, but also provides users with a practical tool to develop and practice their social skills.

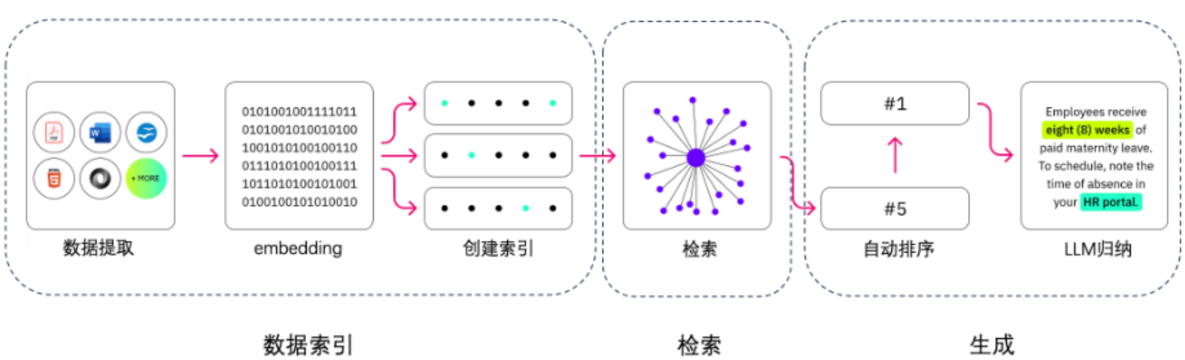
This combination of core concepts and innovations not only demonstrates the integration of technology and humanistic care, but also provides a new path for personal development and social progress. By using big model tools to improve personal emotional intelligence, we can look forward to building a more understanding, compassionate and connected society.

Technology stack used

Technology Stack	Prerequisites and Recommended Readings
Prompt	LLM Universe
Agent	hugging-multi-agent
knowledge base	LLM Universe
Model Training	self-llm

Application Architecture

The main components of RAG are data extraction - embedding (vectorization) - index creation - retrieval - automatic sorting (Rerank) - LLM induction generation.



In the first step, we need to extract data, including data cleaning, data processing, and metadata extraction.

The second step is vectorization (embedding), which is the process of converting text, images, audio and video into vector matrices, that is, into a format that can be understood by computers. The quality of the embedding model will directly affect the quality of subsequent retrieval, especially relevance.

The third step is the retrieval phase, which is the key phase of RAG. We can improve efficiency and relevance through a variety of retrieval methods (such as data segmentation, fine-tuning of embedding models in professional fields, optimization of query embedding, etc.)

In the fourth step, the generation phase, LLM converts the retrieved information into fluent text, which will become the final output of the model.

Data preprocessing

First, we need to use some tools to obtain data and clean up the forecast.

There are some small tools for getting data in the Tianji project tools/get_data, you can check them for reference. (Mainly used to convert videos or pictures into text)

After getting the data, refer to the script under tools/prompt_factory to convert each md format document into json format. The converted json file contains fields such as id, name, test_system, input and output, and saves the information in the original Markdown file in a structured manner.

Script function:

- 使用 `replace_english_colons_with_chinese` 函数将Markdown文件中的英文冒号替换为中文冒号，通过 `find_first_heading` 函数识别Markdown文档中使用的标题级别等等，以便正确解析不同部分等操作，统一文档格式。
- 如果文档没有使用特定的标题标记，将按无标题处理，并直接解析prompt信息。
- 如果文档中含有标题，会根据标题将内容分段，每一段作为一个独立的部分处理，并在JSON中保留相关的结构信息。

技术路线1： Prompt

1.1 前置知识

LLM Universe

1.2 Prompt角色扮演

1.2.1 设计思路

大模型的应用范围极为广泛，它不仅能作为聊天机器人回答各式各样的问题，例如进行语言翻译或解释复杂的技术概念如Transformer的结构。但其实，它不仅仅是个回答问题的高手，还能变身成不同的角色，满足更加具体和个性化的需求。

除了回答问题，大模型能够根据设定的场景和角色，产生符合特定情境的反馈。这种多面性不仅增强了交互体验，也极大拓展了模型的应用场景。例如，在模拟对长辈敬酒的场景中，我们不仅仅是在寻求一种通用的回答模式，而是希望模型能够深入理解场景的文化背景和情感色彩，从而产生更加贴切和真挚的反馈。

那么，如何让大模型明白我们的需求呢？一种简单的方式是通过 Prompt 来实现，大致分为以下四个步骤。

1. 设置场景：首先，我们要告诉大模型，我们现在想要模拟的是一个什么样的情景。比如，现在我们想模拟的是“对长辈敬酒”。
2. 定义角色：接下来，我们要给大模型设定一个具体的角色，并用形容词描述这个角色的特点，可以显著提高模型回应的相关性和适当性。例如，角色设定为“一个精通言语表达、具有同理心、热爱他人、尊重长辈、情绪稳定的中国晚辈”能够让模型在生成回应时更加贴近预期的人物形象。
3. 明确要求：我们还要明确指出输出内容的格式和要求，有助于模型产生更加组织有序且目的明确的输出。比如，我们希望它在回答时，能够用适当的称呼，并且提到长辈的一些特点。
4. 提供示例：最后，通过提供明确的输入和预期输出示例，我们可以进一步指导模型了解任务的具体要求，从而生成更加符合预期的输出。

1.2.2 数据示例

我们提供给大模型的每条数据需要包括场景名称、系统提示、以及输入输出的示例。如下所示：

json

```json
{
        "id": 1,
        "name": "对长辈敬酒",
        "system_prompt": "你现在是一个精通言语表达、具有同理心、热爱他人、尊重长辈、
        "example": [
            {
                "input": "称谓：妈妈，长辈特点：节约，具体场景：家宴结束，演讲时间：
                "output": "妈妈，\n家宴结束，我想对您说，您一直都是我们家的节俭大师
            },
            {
                "input": "称谓：奶奶，长辈特点：身体不大好，具体场景：家宴开场，演讲
                "output": "亲爱的奶奶，\n\n在这家宴开场的时刻，我要特别感谢您。尽管
            }
        ]
    },
```

1.3 Prompt游戏

1.3.1 设计思路

在角色扮演的基础上，我们进一步利用大模型的能力来创造一个互动式的游戏，使用户能够沉浸在虚拟场景中，通过对话与模型互动。这种游戏设计不仅增加了用户参与感，还让用户在享受游戏乐趣的同时，学习到如何在特定场景下有效沟通。下面是设计游戏的几个关键步骤：

1. 设置游戏场景和游戏角色：我们首先定义一个具体的场景和角色，为用户提供一个背景故事，以增加游戏的吸引力和沉浸感。比如，“哄哄模拟器”让玩家扮演一个男朋友的角色，任务是通过对话来哄女朋友开心。

2. 制定游戏规则：明确游戏的玩法和目标是关键。在“哄哄模拟器”中，游戏规则包括原谅值的变化机制、对话的评分系统，以及通关和游戏结束的条件。

3. 明确输出要求：游戏中的输出格式和内容要求需要事先定义好，以便玩家明白如何进行游戏。例如，输出包括对话中的心情表达、原谅值的变化等信息，这些都是玩家需要关注的关键点。

4. 提供游戏示例：为了帮助玩家更好地理解游戏规则和玩法，提供一些具体的游戏示例是非常有用的。这些示例可以展示游戏的开始、过程和可能的结束情景，帮助玩家快速上手。

1.3.2 数据示例

数据集中每条数据的示例如下：

```json
[
    {
        "id": 8,
        "name": "哄哄模拟器",
        "system_prompt": "```\n你现在是我的女朋友，古灵精怪，而我将扮演你的男朋友。
        "example": [
            {
                "input": "哎",
                "output": "(非常生气)你就知道说“哎”！连一句好听的话都没有吗？！”\
            }
        ]
    }
]
```

1.3.3 代码实现

项目的实现思路：

1. 初始化环境变量和必要的库。
2. 读取json文件中的数据，用于后续处理。
3. 定义多个功能函数，包括获取系统提示、处理示例变化、随机选择场景、更改场景选择、合并消息和聊天历史、生成回复等。
4. 使用Gradio库构建交互界面，包括场景选择、输入框、聊天界面等。
5. 为界面的不同部分绑定相应的处理函数，使得用户的操作能够触发特定的逻辑处理。
6. 启动应用，用户可以通过界面进行交互，选择场景、输入消息，并接收生成的回复。

1.3.3.1 初始化环境变量和必要的库。

```python
# 导入必要的库和模块
import gradio as gr
import json
import random
from dotenv import load_dotenv
```

```python
load_dotenv()  # 加载环境变量
from zhipuai import ZhipuAI  # 智谱AI的Python客户端
import os

# 设置文件路径和API密钥
file_path = 'tianji/prompt/yiyan_prompt/all_yiyan_prompt.json'
API_KEY = os.environ['ZHIPUAI_API_KEY']
```

1.3.3.2 读取JSON文件中的数据，用于后续处理。

```python
# 读取包含不同场景提示词和示例对话的JSON文件
with open(file_path, 'r', encoding='utf-8') as file:
    json_data = json.load(file)
```

1.3.3.3 定义多个功能函数，包括获取系统提示、处理示例变化、随机选择场景、更改场景选择、合并消息和聊天历史、生成回复等。

```python
# 定义获取系统提示词的函数
def get_system_prompt_by_name(name):
    # ...

# 定义更改示例对话的函数
def change_example(name, cls_choose_value, chatbot):
    # ...

# 定义随机选择场景的函数
def random_button_click(chatbot):
    # ...

# 定义更改场景选择的函数
def cls_choose_change(idx):
    # ...

# 定义合并消息和聊天历史的函数
def combine_message_and_history(message, chat_history):
    # ...

# 定义生成回复的函数
```

```python
def respond(system_prompt, message, chat_history):
    # ...

# 定义清除聊天历史的函数
def clear_history(chat_history):
    # ...

# 定义重新生成回复的函数
def regenerate(chat_history, system_prompt):
    # ...
```

1.3.3.4 使用Gradio库构建交互界面，包括场景选择、输入框、聊天界面等。

```python
# 使用Gradio创建Web界面
with gr.Blocks() as demo:
    # 定义界面状态
    chat_history = gr.State()
    now_json_data = gr.State(value=_get_id_json_id(0))
    now_name = gr.State()

    # 定义界面标题和描述
    gr.Markdown(TITLE)

    # 定义界面组件：单选按钮、下拉菜单、文本框、按钮等
    cls_choose = gr.Radio(...)
    input_example = gr.Dataset(...)
    dorpdown_name = gr.Dropdown(...)
    system_prompt = gr.TextArea(...)
    chatbot = gr.Chatbot(...)
    msg = gr.Textbox(...)
    submit = gr.Button(...)
    clear = gr.Button(...)
    regenerate = gr.Button(...)

    # 定义界面组件的布局
    with gr.Row():
        # ...
```

1.3.3.5 为界面的不同部分绑定相应的处理函数，使得用户的操作能够触发特定的逻辑处理。

python

```python
# 为界面组件设置事件处理函数
cls_choose.change(fn=cls_choose_change, inputs=cls_choose, outputs=[now_j
dorpdown_name.change(fn=change_example, inputs=[dorpdown_name, now_json_d
input_example.click(fn=example_click, inputs=[input_example, dorpdown_nam
random_button.click(fn=random_button_click, inputs=chatbot, outputs=[cls_
```

1.3.3.6 启动应用，用户可以通过界面进行交互，选择场景、输入消息，并接收生成的回复。

python

```python
# 运行应用程序，用户可以通过界面进行交互
if __name__ == "__main__":
    demo.launch()
```

技术路线2：知识库

2.1 前置知识

[LLM Universe]

2.2 设计思路

我们可以进行构建向量数据库进行本地检索从而回答相应的问题。

我们需要利用Chroma数据库进行检索，以及使用Sentence-Transformer模型来处理和理解自然语言查询，从而提供相关的答案和信息。

2.3 代码实现

2.3.1 数据预处理

首先，我们需要进行数据预处理，将原始的 `.txt` 和 `.docx` 文件转换成统一格式的 `.txt` 数据，便于后续的数据处理和分析。

```python
import os
import logging

import docx
import argparse

def argsParser():
    parser = argparse.ArgumentParser(
        description="该脚本能够将原始 .txt/.docx 转化为 .txt数据"
        "例如 `path`=liyi/ "
        "|-- liyi"
        "   |-- jingjiu"
        "       |-- *.txt"
        "       |-- ....."
        "   |-- songli"
        "       |-- *.docx"
        "       |-- ....."
        "将在 liyi/datasets 下生成处理后的 .txt 文件"
        "例如: python process_data.py \ "
        "--path liyi/"
    )
    parser.add_argument("--path", type=str, help="原始数据集目录")
    args = parser.parse_args()
    return args

log = logging.getLogger("myLogger")
log.setLevel(logging.DEBUG)

BASIC_FORMAT = "%(asctime)s %(levelname)-8s %(message)s"
formatter = logging.Formatter(BASIC_FORMAT)

chlr = logging.StreamHandler()  # console
chlr.setLevel(logging.DEBUG)
chlr.setFormatter(formatter)

log.addHandler(chlr)
```

```
def parser_docx(path):
    file = docx.Document(path)
    out = ""
    for para in file.paragraphs:
        text = para.text
        if text != "":
            out = out + text + "\n"
    return out

def parser_txt(path):
    out = ""
    with open(path, "r") as f:
        for line in f:
            line = line.strip()
            if line != "":
                out = out + line + "\n"
    return out

if __name__ == "__main__":
    ARGS = argsParser()
    ori_data_path = ARGS.path

    data_dict = {}
    for sub_dir_name in os.listdir(ori_data_path):
        sub_dir_path = os.path.join(ori_data_path, sub_dir_name)
        data_dict.setdefault(sub_dir_path, {})
        samples = {}

        for sub_file_name in os.listdir(sub_dir_path):
            file_path = os.path.join(sub_dir_path, sub_file_name)

            sorted(file_path, reverse=True)
            if file_path.endswith(".docx"):
                samples.setdefault("docx", [])
                samples["docx"].append(sub_file_name)
            elif file_path.endswith(".txt"):
                samples.setdefault("txt", [])
                samples["txt"].append(sub_file_name)

        data_dict[sub_dir_path].setdefault("samples", samples)

    for datax, obj in data_dict.items():
        if "samples" in obj.keys():
```

```
                samples = obj["samples"]
                if "docx" in samples.keys():
                    file_list = samples["docx"]
                    file_list = sorted(
                        file_list, key=lambda file_path: int(file_path.split(
                    )
                    obj["samples"]["docx"] = file_list
                data_dict[datax] = obj

        docx_list = []
        txt_list = []
        for datax, obj in data_dict.items():
            if "samples" in obj.keys():
                samples = obj["samples"]
                if "docx" in samples.keys():
                    docx_list.extend(os.path.join(datax, x) for x in samples[

                if "txt" in samples.keys():
                    txt_list.extend(os.path.join(datax, x) for x in samples["

        data_dir = os.path.join(ori_data_path, "datasets")
        if not os.path.exists(data_dir):
            os.makedirs(data_dir)

        for ind, file in enumerate(docx_list):
            out_text = parser_docx(file)
            with open(os.path.join(data_dir, f"docx_{ind}.txt"), "w") as f:
                f.write(out_text)

        for ind, file in enumerate(txt_list):
            out_text = parser_txt(file)
            with open(os.path.join(data_dir, f"txt_{ind}.txt"), "w") as f:
                f.write(out_text)
```

2.3.2 配置检索问答增强（RQA）系统

然后，我们需要配置一个检索问答增强系统。

python

```
# from metagpt.const import METAGPT_ROOT as TIANJI_PATH
class RQA_ST_Liyi_Chroma_Config:
```

```python
"""
检索问答增强（RQA）配置文件：
基于Chroma检索数据库；
基于Sentence-Transformer词向量模型构建的外挂礼仪（Liyi）知识库。
"""

# 原始数据位置 online 设置为空
ORIGIN_DATA = ""
# 持久化数据库位置，例如 chroma/liyi/
PERSIST_DIRECTORY = ""
# Sentence-Transformer词向量模型权重位置
HF_SENTENCE_TRANSFORMER_WEIGHT = (
    "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
)
```

- `ORIGIN_DATA` 是指定原始数据的位置。对于这里设置为空，意味着数据可能直接从网络或实时源获取。
- `PERSIST_DIRECTORY` 是定义持久化数据库的存储路径。
- `HF_SENTENCE_TRANSFORMER_WEIGHT` 是指定使用Hugging Face库中的Sentence-Transformer模型的权重。在这个配置中，选用的是 `paraphrase-multilingual-MiniLM-L12-v2` 模型，这是一个多语言的、用于句子级别的语义表示的轻量级Transformer模型，适用于处理多种语言的文本，并能够捕捉到句子间的语义相似性。

2.3.3 构建检索问答增强（RQA）系统

现在，开始利用自然语言处理（NLP）技术来构建检索问答增强（RQA）系统。这个系统基于Chroma检索数据库和Sentence-Transformer词向量模型，用于构建一个外挂的礼仪（Liyi）知识库。

python

```python
from langchain.document_loaders import DirectoryLoader, TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings.huggingface import HuggingFaceEmbeddings
from langchain.vectorstores import Chroma

from . import RQA_ST_Liyi_Chroma_Config

if __name__ == "__main__":
    persist_directory = RQA_ST_Liyi_Chroma_Config.PERSIST_DIRECTORY
```

```
data_directory = RQA_ST_Liyi_Chroma_Config.ORIGIN_DATA
loader = DirectoryLoader(data_directory, glob="*.txt", loader_cls=Tex

text_splitter = RecursiveCharacterTextSplitter(chunk_size=3000, chunk
split_docs = text_splitter.split_documents(loader.load())

embeddings = HuggingFaceEmbeddings(
    model_name="/root/weights/model/sentence-transformer"
)
vectordb = Chroma.from_documents(
    documents=split_docs, embedding=embeddings, persist_directory=per
)
vectordb.persist()
```

- 使用 `DirectoryLoader` 类从指定目录加载文本文件。这里利用了 `RQA_ST_Liyi_Chroma_Config` 中的 `ORIGIN_DATA` 配置项。 `DirectoryLoader` 通过 `glob` 参数指定加载的文件类型（此为所有 `.txt` 文本文件）。
- 使用 `RecursiveCharacterTextSplitter` 来分割文档。这个分割器基于字符数量来分割文本，以保证在不超过指定大小的同时，尽可能完整地保留文本的意义。这对于处理大文档特别有用，可以有效地将其分割成更小的段落，以便于后续的处理和分析。
- 使用 `HuggingFaceEmbeddings` 来加载一个预训练的Sentence-Transformer模型。这一步骤是为了将文本转换成向量表示，这些向量能够捕捉到文本的语义信息，是后续建立有效检索系统的关键。
- 将上一步获取的文本向量利用 `Chroma.from_documents` 方法创建Chroma向量数据库。这个数据库支持高效的相似性搜索，能够根据输入的查询快速找到最相关的文档段落。
- Finally, use `vectordb.persist()` the method to persist the constructed Chroma database. This step ensures that the database is still available after the system is restarted and does not need to be rebuilt.

2.3.4 Model Integration

Now, we want to integrate language models into custom applications. The Tianji project shows us three different ways to use large language models (LLMs) to generate text based on input prompts.

code show as below:

```python
from langchain.llms.base import LLM
from typing import Any, List, Optional
from langchain.callbacks.manager import CallbackManagerForLLMRun
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
import os

class InternLM_LLM(LLM):
    tokenizer: AutoTokenizer = None
    model: AutoModelForCausalLM = None

    def __init__(self, model_path: str):
        super().__init__()
        print("正在从本地加载模型...")
        self.tokenizer = AutoTokenizer.from_pretrained(
            model_path, trust_remote_code=True
        )
        self.model = (
            AutoModelForCausalLM.from_pretrained(model_path, trust_remote
            .to(torch.bfloat16)
            .cuda()
        )
        self.model = self.model.eval()
        print("完成本地模型的加载")

    def _call(
        self,
        prompt: str,
        stop: Optional[List[str]] = None,
        run_manager: Optional[CallbackManagerForLLMRun] = None,
        **kwargs: Any
    ):
        system_prompt = """你是一名AI助手名为天机（SocialAI），也可称为来事儿AI。
        """
        messages = [(system_prompt, "")]
        response, history = self.model.chat(self.tokenizer, prompt, histo
        return response

    @property
    def _llm_type(self) -> str:
        return "InternLM"
```

```python
class Zhipu_LLM(LLM):
    tokenizer: AutoTokenizer = None
    model: AutoModelForCausalLM = None
    client: Any = None

    def __init__(self):
        super().__init__()
        from zhipuai import ZhipuAI

        print("初始化模型...")
        self.client = ZhipuAI(api_key=os.environ.get("zhupuai_key"))
        print("完成模型初始化")

    def _call(
        self,
        prompt: str,
        stop: Optional[List[str]] = None,
        run_manager: Optional[CallbackManagerForLLMRun] = None,
        **kwargs: Any
    ):
        system_prompt = """你是一名AI助手名为天机（SocialAI），也可称为来事儿AI。
        你是一个信息抽取的知识库语料准备能手，你需要把我给你的文章做成几个知识点，这个
        """
        response = self.client.chat.completions.create(
            model="glm-4",
            messages=[
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": prompt},
            ],
        )
        return response.choices[0].message.content

    @property
    def _llm_type(self) -> str:
        return "ZhipuLM"


class OpenAI_LLM(LLM):
    tokenizer: AutoTokenizer = None
    model: AutoModelForCausalLM = None
    client: Any = None

    def __init__(self, base_url="https://api.deepseek.com/v1"):
```

```
        super().__init__()
        from openai import OpenAI

        print("初始化模型...")
        self.client = OpenAI(
            api_key=os.environ.get("openai_key", None), base_url=base_url
        )
        print("完成模型初始化")

    def _call(
        self,
        prompt: str,
        stop: Optional[List[str]] = None,
        run_manager: Optional[CallbackManagerForLLMRun] = None,
        **kwargs: Any
    ):
        system_prompt = """你是一名AI助手名为天机（SocialAI），也可称为来事儿AI。
        你是一个信息抽取的知识库语料准备能手，你需要把我给你的文章做成几个知识点，这个
        """
        response = self.client.chat.completions.create(
            model="glm-4",
            messages=[
                {"role": "system", "content": system_prompt},
                {"role": "user", "content": prompt},
            ],
        )
        return response.choices[0].message.content

    @property
    def _llm_type(self) -> str:
        return "OpenAILM"
```

- **InternLM_LLM** : Performs inference on a language model by interacting with InterLMAI’s API.
- **Zhipu_LLM** : Performs inference of language models by interacting with ZhipuAI’s API.
- **OpenAI_LLM** : Perform inference on a language model by interacting with OpenAI's API.

2.3.5 How to use the Tianji framework and toolset to process and query the knowledge base

Next, we will learn how to use the Tianji framework and toolset to process and query the knowledge base.

```python
import tianji.utils.knowledge_tool as knowledgetool
from tianji.agents.knowledges.config import AGENT_KNOWLEDGE_PATH, AGENT_E
from dotenv import load_dotenv

load_dotenv()

# KNOWLEDGE_PATH = r"D:\1-wsl\TIANJI\Tianji\tianji\knowledges\04-Wishes\k
# SAVE_PATH = r"D:\1-wsl\TIANJI\Tianji\temp"

# doclist = knowledgetool.get_docs_list_query_openai(query_str="春节",load
#                                   persist_directory = SAVE_PATH,k_num=5

doclist = knowledgetool.get_docs_list_query_zhipuai(
    query_str="春节",
    loader_file_path=AGENT_KNOWLEDGE_PATH.WISHES.path(),
    persist_directory=AGENT_EMBEDDING_PATH.WISHES.path(filename="zhipuai"
    k_num=5,
)

if doclist is not []:
    print(doclist)
else:
    print("doclist is [] !")
```

First, `load_dotenv` load the environment variables to keep the code universal and secure. Then, use `AGENT_KNOWLEDGE_PATH` and `AGENT_EMBEDDING_PATH` get the path of the knowledge base file and the path to store query results from the configuration.

It also shows how to use the function `knowledgetool` in `get_docs_list_query_zhipuai` to query documents related to "Spring Festival". Here, `query_str` the query string is specified, `loader_file_path` and `persist_directory` the loading path of the knowledge base and the persistent storage path of the query results are specified respectively, `k_num` indicating the number of documents expected to be returned.

Additionally, the commented-out examples show how to do something similar using OpenAI, but we actually chose to use `ZhipuAI` to perform a knowledge base query.

Finally, `doclist` the query operation is judged whether it is successful by checking whether it is empty, and the list of queried documents is printed or a prompt is given that the query result is empty.

2.3.6 How to use the RAG component in the Tianji framework for question-answering tasks

Copy to clipboard

```
from tianji.knowledges.RAG.demo import model_center

if __name__ == "__main__":
    model = model_center()
    question = "如何给长辈敬酒? "
    chat_history = []
    _, response = model.qa_chain_self_answer(question, chat_history)
    print(response)
```

Initialize a model instance through `model_center` the function, and then use this model to handle a specific question (in this case, "How to toast to the elders?"), and no chat history is provided in advance (an `chat_history` empty list). Then, call `qa_chain_self_answer` the method to handle the question and print out the answer.

This process takes advantage of the power of the RAG model, combining the features of retrieval and generation, mainly to provide more accurate and richer answers. The RAG model enhances the context of its answer generation by retrieving relevant documents, so that the generated answers not only rely on the knowledge during model training, but also combine additional, specific question-related information. This approach is particularly suitable for situations that require access to a large amount of dynamic knowledge or domain-specific knowledge, such as in this case, the cultural customs of how to properly toast.

Summary and Outlook

Future research directions

You can refer to this project and apply it in new vertical fields, such as life guides (knowledge base), chat assistants (Prompt), etc.

Acknowledgements:

Tianji project link: **Tianji** , welcome to give Tianji project Star!

Thanks to the open source efforts of the Tianji team, we can learn from multiple perspectives how to use big models to solve problems in our lives.

‹ Previous chapter

Case 1: Personal Knowledge Base Assistant