# 知识工程-作业9 基于知识图谱的问答系统

2024214500 叶璨铭

## 代码与文档格式说明

本文档使用Jupyter Notebook编写，遵循Diátaxis 系统 Notebook实践 <https://nbdev.fast.ai/tutorials/best_practices.html，所以同时包括了实验文档和实验代码>。

本文档理论上支持多个格式，包括ipynb, docx, pdf 等。您在阅读本文档时，可以选择您喜欢的格式来进行阅读，建议您使用 Visual Studio Code (或者其他支持jupyter notebook的IDE, 但是VSCode阅读体验最佳) 打开 ipynb格式的文档来进行阅读。

为了记录我们自己修改了哪些地方，使用git进行版本控制，这样可以清晰地看出我们基于助教的代码在哪些位置进行了修改，有些修改是实现了要求的作业功能，而有些代码是对原本代码进行了重构和优化。我将我在知识工程课程的代码，在作业截止DDL之后，开源到 <https://github.com/2catycm/THU-Coursework-Knowledge-Engineering.git> ，方便各位同学一起学习讨论。

## 代码规范说明

在我们实现函数过程中，函数的docstring应当遵循fastai规范而不是numpy规范，这样简洁清晰，不会Repeat yourself。相应的哲学和具体区别可以看 <https://nbdev.fast.ai/tutorials/best_practices.html#keep-docstrings-short-elaborate-in-separate-cells>

为了让代码清晰规范，在作业开始前，使用 ruff format格式化助教老师给的代码;

alt text

很好，这次代码格式化没有报错。

Pylance 似乎也没有明显问题。

## 实验环境准备

采用上次的作业专属环境，为了跑通最新方法，使用3.12 和 torch 2.6

conda create -n assignments python=3.12  
conda activate assignments  
pip install -r ../requirements.txt  
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu126  
pip install -U git+https://github.com/TorchRWKV/flash-linear-attention

本次作业似乎没有新的依赖，只是用到了 transformers

import json  
import numpy as np  
from transformers import BertTokenizer, BertModel

## 原理回顾和课件复习

课上详细介绍了 Knowledge-based Question Answering

首先区分了一下属性和关系，属性是 实体, 属性类型, 字符串； 关系是 实体，关系类型，实体。

## 1.1 基于知识图谱的问答系统

根据助教老师的要求，我们有四步要做

1. 使用 Python 解析 zhishime.json 文件，创建知识图谱
2. 实现头实体检索模块（使用正向最大匹配或命名实体识别）
3. 使用预训练模型计算问题与关系的相似度
4. 提取答案并评估准确性

### 1. 使用 python 解析 zhishime.json 文件，并将解析出的 dict 保存为文件

注意到 zhishime 实际上是一个 jsonl文件，一行是一个json，每一行是

{  
 "\_id": {"$oid": "5a4a0579b63209a91d0c41c7"},  
 "head": "1987大悬案",  
 "relation": "监制",  
 "tail": "狄诺迪洛伦提斯\u003cbr/\u003eRichardRoth\u003cbr/\u003e柏尼·威廉斯",  
}

这样的格式, 也就是说有 \_id, head, relation 和 tail

这是知识图谱的典型的三元组。

老师已经给了我们 preprocess.py 的初步实现。

kg = {  
 "head2id": head2id,  
 "tail2id": tail2id,  
 "relation2id": relation2id,  
 "relation\_triplets": relation\_triplets,  
 }

这个函数的目标是为了得到 head tail 的id，其实和没处理差不多。

import json  
import os  
from tqdm import tqdm  
  
def preprocess():  
 """预处理知识图谱数据，构建实体映射和三元组索引"""  
 # 初始化数据结构  
 kg = {  
 "head2id": {}, # 头实体到ID的映射  
 "tail2id": {}, # 尾实体到ID的映射   
 "relation2id": {}, # 关系到ID的映射  
 "relation\_triplets": [] # 存储(hid, rid, tid)形式的三元组  
 }  
  
 try:  
 # 确保输出目录存在  
 os.makedirs("./processed", exist\_ok=True)  
   
 # 读取原始JSONL文件（注意：使用..表示上级目录）  
 with open("../zhishime.json", "r", encoding="utf-8") as f:  
 # 逐行解析JSON对象（适用于JSON Lines格式）  
 raw\_relation\_data = [json.loads(line) for line in f]  
   
 # 等价写法（更易理解）：  
 # raw\_relation\_data = []  
 # for line in f:  
 # data = json.loads(line)  
 # raw\_relation\_data.append(data)  
  
 # 遍历每个三元组进行索引构建  
 bar = tqdm(raw\_relation\_data)  
 for item in bar:  
 head = item["head"]  
 relation = item["relation"]  
   
 # 处理包含换行符的尾实体（示例数据中的<br/>分隔符）  
 tail = item["tail"].replace("\u003cbr/\u003e", ", ") # 将HTML换行符转换为逗号分隔  
   
 # 构建实体ID映射（自动递增分配ID）  
 # head2id.setdefault等效写法，但更推荐当前写法  
 if head not in kg["head2id"]:  
 kg["head2id"][head] = len(kg["head2id"])  
   
 if tail not in kg["tail2id"]:  
 kg["tail2id"][tail] = len(kg["tail2id"])  
   
 if relation not in kg["relation2id"]:  
 kg["relation2id"][relation] = len(kg["relation2id"])  
   
 # 构建三元组索引  
 hid = kg["head2id"][head]  
 rid = kg["relation2id"][relation]  
 tid = kg["tail2id"][tail]  
 kg["relation\_triplets"].append((hid, rid, tid))  
  
 # 打印统计信息  
 print(f"[统计] 头实体数: {len(kg['head2id'])} | 尾实体数: {len(kg['tail2id'])} | 关系类型数: {len(kg['relation2id'])}")  
 print(f"[统计] 总三元组数: {len(kg['relation\_triplets'])}")  
  
 # 保存处理结果  
 with open("./processed/kg.json", "w", encoding="utf-8") as json\_file:  
 json.dump(kg, json\_file,   
 ensure\_ascii=False, # 保留非ASCII字符原文  
 indent=4) # 美化格式便于查看  
   
 except FileNotFoundError:  
 print("错误：未找到原始数据文件，请检查路径是否正确")  
 except json.JSONDecodeError as e:  
 print(f"JSON解析错误：第{e.lineno}行数据格式异常，错误详情：{e.msg}")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 preprocess()

我们稍微重构了下，让整个功能更加稳定。



### 2. 实现头实体检索模块（使用正向最大匹配或命名实体识别）

在实现这个函数之前，我们需要明确一个问题，

* what：到底什么是头实体？
* 在一个question里面，头实体应该是一个还是多个？

事实上，这个和对问题的模板预设有关，这次实验我们假设问题类似于 “{HEAD} 的 {RELATION}?” ， 回答是 “{TAIL}”， 所以认为问题中就只有一个实体。例如："周杰伦的出生日期是什么？" → 头实体是"周杰伦"。

实际上问题中可能包含多个实体，但通常只有一个是头实体

例如："周杰伦和林俊杰谁的粉丝多？" → 这种情况复杂一些，可能需要查询多个头实体。

刚才我们定义了 "head2id": {}, # 头实体到ID的映射

所以，我们只需要做一个“多字符串(模式)匹配", 找到 question 字符串中的那一个实体就好。

所谓 正向最大匹配法，就是在问题字符串中，尝试匹配知识图谱中最长的实体名称，优先匹配最长的实体名称，因为这样更可能是完整的实体。 那么相应的逻辑就很简单了

def search\_head\_entity(kg: dict, question: str) -> str:  
 """基于正向最大匹配的头实体识别  
 Args:  
 kg: 知识图谱字典，包含head2id等字段  
 question: 待查询的问题文本  
 Returns:  
 匹配成功的头实体字符串，未找到返回None  
 """  
 # 获取所有可能的头实体  
 all\_heads = list(kg['head2id'].keys())  
   
 # 对头实体按长度排序，优先匹配长的实体  
 all\_heads.sort(key=len, reverse=True)  
   
 # 遍历所有头实体，检查是否在问题中出现  
 for head in all\_heads:  
 if head in question:  
 return head  
   
 # 如果没有找到匹配的头实体  
 return ""

为了提高处理速度，我们使用缓存

sorted\_heads = None  
  
def search\_head\_entity(kg: dict, question: str) -> str:  
 """基于正向最大匹配的头实体识别  
 Args:  
 kg: 知识图谱字典，包含head2id等字段  
 question: 待查询的问题文本  
 Returns:  
 匹配成功的头实体字符串，未找到返回None  
 """  
 # 获取所有可能的头实体  
 global sorted\_heads  
 if sorted\_heads is None:  
 all\_heads = list(kg['head2id'].keys())  
   
 # 对头实体按长度排序，优先匹配长的实体  
 sorted\_heads = sorted(all\_heads, key=len, reverse=True)  
   
 # 遍历所有头实体，检查是否在问题中出现  
 for head in sorted\_heads:  
 if head in question:  
 return head  
   
 # 如果没有找到匹配的头实体  
 return ""

注意到后面的代码是这样的

head = search\_head\_entity(kg, question) if head is None:

为了不让类型有问题（不想用optional），我觉得应该改成 if head=="":

### 3. 使用预训练模型计算问题与关系的相似度

检查 main.py 代码，发现原本代码这里写错了

relations = list(kg[head].keys())

我们在kg中根本没有存储从 head 映射到 关系本身的信息！

所以我们需要重新修改 preprocess.py 把这个信息假如进去才对。

刚才处理json的时候搞了半天id其实是无用功，根本不需要id，核心问题是 一个head有多少个相关的relation。

同理，这段代码也不对

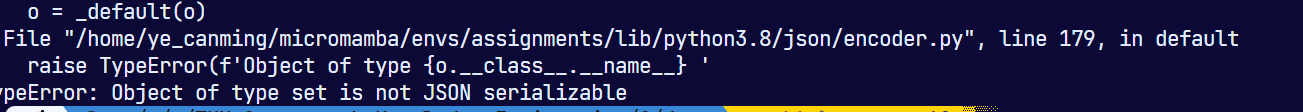
answer = kg[head][max\_relation]

这里要解决的核心问题是，给定一个head和一个relation，找到对应的tail的集合。

我们首先加上代码

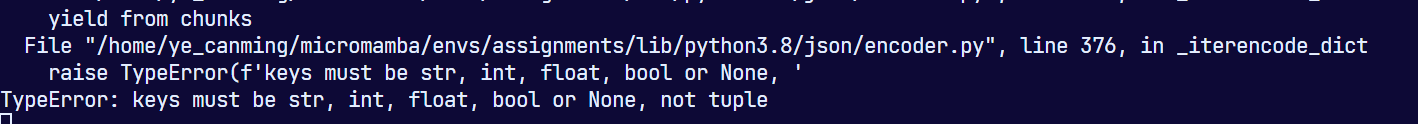
# 构建头实体到关系的映射  
if head not in kg["head2relations"]:  
 kg["head2relations"][head] = set()  
kg["head2relations"][head].add(relation)  
  
# 构建头实体到关系和答案的映射  
if (head, relation) not in kg["head\_relations2answers"]:  
 kg["head\_relations2answers"][(head, relation)] = ""  
kg["head\_relations2answers"][(head, relation)] += f"{tail}, "

重新运行 python preprocess.py



那么在json中只能用 list

if head not in kg["head2relations"]:  
 kg["head2relations"][head] = list()  
kg["head2relations"][head].append(relation)



json也没有 tuple 的概念



于是需要修改

head\_relation = f"({head}, {relation})"  
if head\_relation not in kg["head\_relations2answers"]:  
 kg["head\_relations2answers"][head\_relation] = ""  
kg["head\_relations2answers"][head\_relation] += f"{tail}, "

现在可以看到

import json  
with open("data/processed/kg.json", "r", encoding="utf-8") as f:  
 kg = json.load(f)  
kg.keys()

dict\_keys(['head2id', 'tail2id', 'relation2id', 'relation\_triplets'])

在main中使用

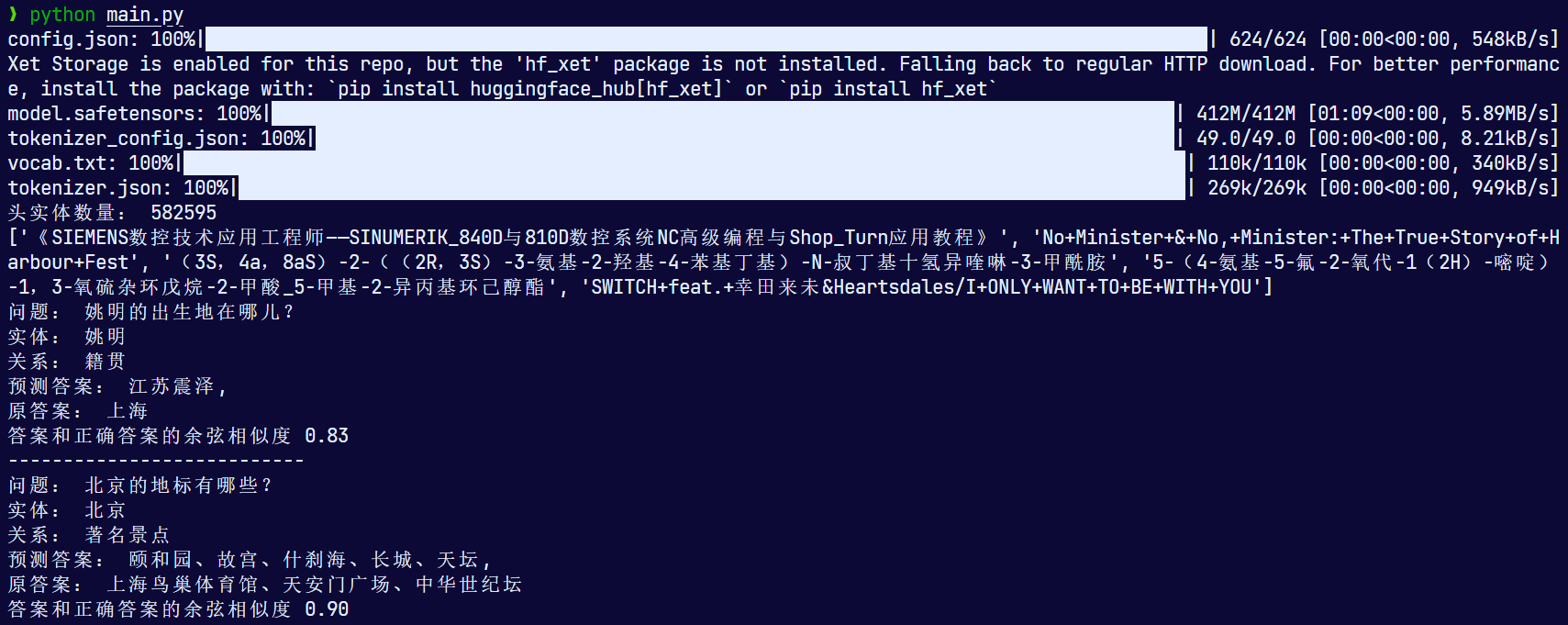
relations = list(kg["head2relations"][head])  
 ...  
 answer = kg["head\_relations2answers"][(head, max\_relation)]

注意json没有元组，所以我们要这样写

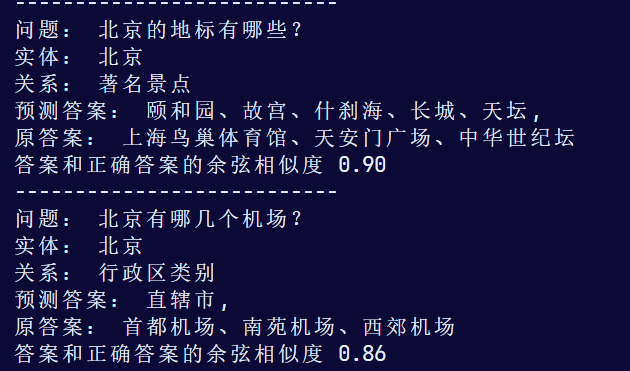
answer = kg["head\_relations2answers"][f"({head}, {max\_relation})"]

### 4. 提取答案并评估准确性

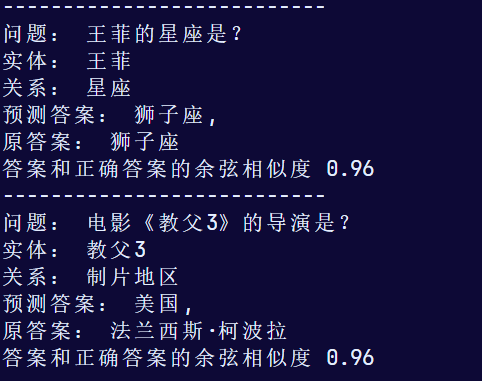
现在我们终于可以正确运行 main.py 了



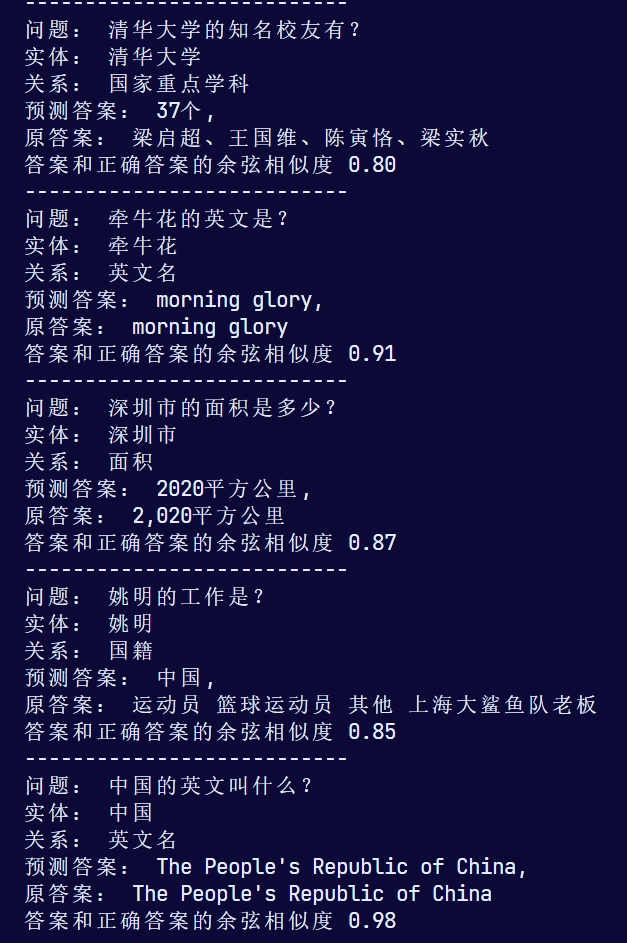
第一个问题就失败了，可能是因为 bert 以为 出生地 是 籍贯。



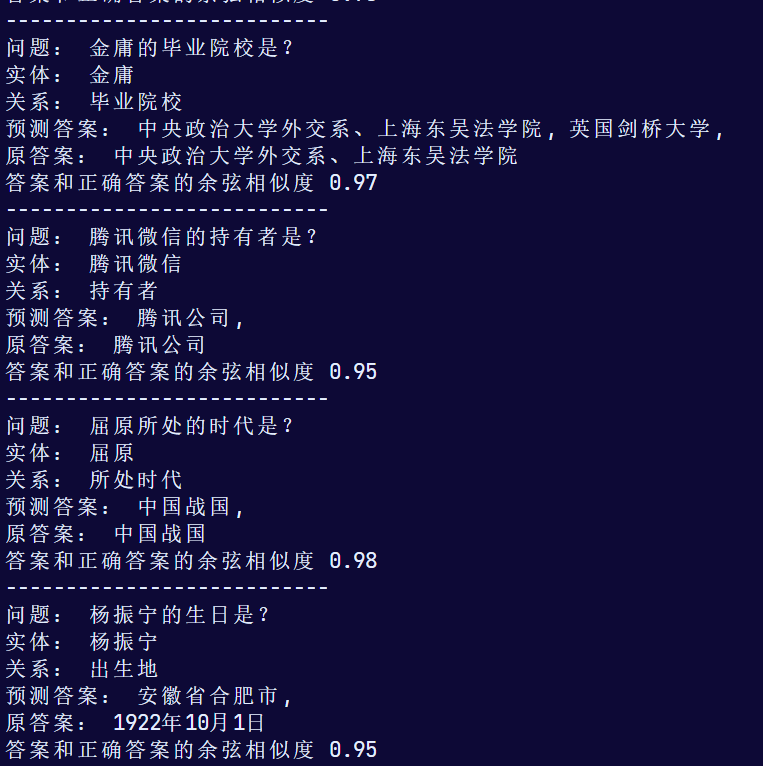
bert再次判断错误，以为 机场 是 行政区类别

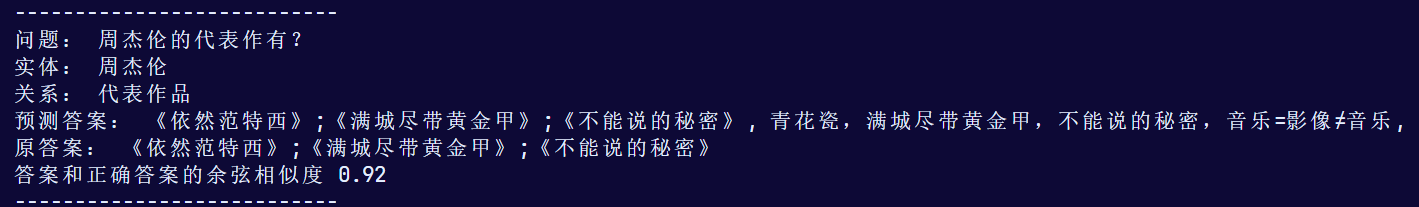


王菲的星座是正确的，但是导演关系被误以为是制片地区，而且判断答案是否正确的时候也以为是0.96很高的评分。



似乎对于英文的相似度，bert比较在行，其他的总会关系理解错误。





最后这个问题我刚才使用了拼接多个关系的方法，所以有逗号，比原答案要多更多东西，但是知识库里面信息有点问题，有个奇怪的“音乐=影像≠音乐”

## 学术问题

### 问题1

**问题一**：本次作业中所有测试问题都在知识库中有准确答案，然而在实际场景下，这是几乎不可能的。请你回答：有哪些方法可以解决无法被现有知识库很好覆盖的问题？

我想，对于人类而言，如果没有学过一个知识，那确实也是不好回答的，只能做两件事情

1. 基于现有知识进行推理，给出我对于未知事物性质的猜测，通过想象（幻觉）去给出一个看起来合理的答案。
2. 询问外界，比如问老师助教、查阅互联网信息

那么对于知识问答系统来说总体思路也是一样，对于知识库中没有的信息，调用联网搜索工具，先阅读网页上的有关内容，然后再回答，如果没有搜到相关的内容, 再自己随便推测一下。

具体来说，有这样一些技术

* 扩展知识库
  + 可以通过信息抽取（Information Extraction, IE）自动从互联网文本中挖掘新实体、新关系，动态扩充知识库。
  + 可以利用半监督学习、远程监督（Distant Supervision）等技术从未标注数据中提取知识。
* 基于已有知识进行推理，推导出未显式存储的答案。例如利用知识图谱补全（KG Completion）、链式推理（Chain-of-thought Reasoning）等方法。
* 结合知识图谱推理与生成模型，先尽可能用结构化数据回答，无法回答时交给生成式模块。利用生成式大模型（如T5, GPT）直接根据用户问题生成合理答案，即使知识图谱中没有明确对应知识。

### 问题2

**问题二**：使用预训练 BERT 进行问题检索有哪些优缺点？试从 KBQA 发展历程的角度进行回答。

**优点**：

* **语义理解更强**：BERT 是基于Transformer结构的深度预训练模型，能够捕捉复杂的上下文信息，比传统的词袋（Bag-of-Words）或浅层模型理解能力更强。
* **弱化表面匹配依赖**：传统 KBQA（Knowledge-Based QA）方法往往依赖表面词匹配，BERT 能通过向量空间建模实现更深层次的语义匹配。
* **适应性好**：在不同领域，只需少量微调或无监督检索（如Sentence-BERT等）就能迁移到新的任务上。

**缺点**：

* **检索效率较低**：原生BERT无法高效批量检索，需要引入额外的索引结构（如FAISS）或使用轻量版本（如DistilBERT）。
* **难以处理实体特定信息**：BERT擅长理解自然语言，但在结构化实体匹配、别名消歧等方面弱于专门针对实体设计的方法。
* **KBQA历史发展视角**：
  + **早期 KBQA**：基于规则模板（rule-based）或图遍历（graph traversal），不涉及复杂的语义建模。
  + **后期 KBQA**（2018年以后）：随着BERT类模型兴起，开始引入深度语义匹配，但同时也暴露出计算量大、对索引和检索优化需求高的问题。
  + 目前越来越多的方法倾向于 **dense retrieval + graph reasoning** 的组合，而不是单纯依赖BERT匹配。

### 问题3

**问题三**：我们都知道，大语言模型容易出现幻觉，你知道有哪些方法可以缓解大语言模型的幻觉现象？

大语言模型（LLMs）容易出现幻觉（Hallucination），即生成逻辑上正确但事实错误的内容。

大体来说，最有效的方法通常是**检索+生成**结合，同时在训练、推理两个阶段进行多层次控制。

具体缓解方法主要有：

1. **检索增强生成（Retrieval-Augmented Generation, RAG）**：
   * 在回答问题前检索真实世界文档，将检索到的内容作为条件输入给生成模型，提高回答的事实准确性。
2. **事实核查（Fact-Checking）机制**：
   * 在模型生成回答后，增加后验验证模块，检查生成内容是否符合事实数据库或知识图谱中的已知知识。
3. **训练时使用高质量数据**：
   * 使用严格筛选的、包含事实标注的数据集进行微调，减少模型在训练阶段学到错误信息。
4. **使用知识注入（Knowledge Injection）**：
   * 在训练过程中引入结构化知识（如知识图谱实体嵌入、关系嵌入），增强模型的知识记忆和推理能力。
5. **模型约束与提示优化（Prompt Engineering）**：
   * 通过在提示词中明确要求“基于事实”“引用来源”等，提高模型生成符合事实的可能性。
6. **模型架构优化**：
   * 设计专门针对可靠性优化的架构，比如以检索模块为主、生成模块为辅的混合模型。

## 2 进阶要求

刚才我们看到 bert 模型对于判断相似度还是太弱了，一开始我以为是它不懂中文，但是用的已经是“model = BertModel.from\_pretrained("bert-base-chinese") tokenizer = BertTokenizer.from\_pretrained("bert-base-chinese")”了

所以可能因为 bert 不是针对检索优化的嵌入模型，而是一个理解模型。

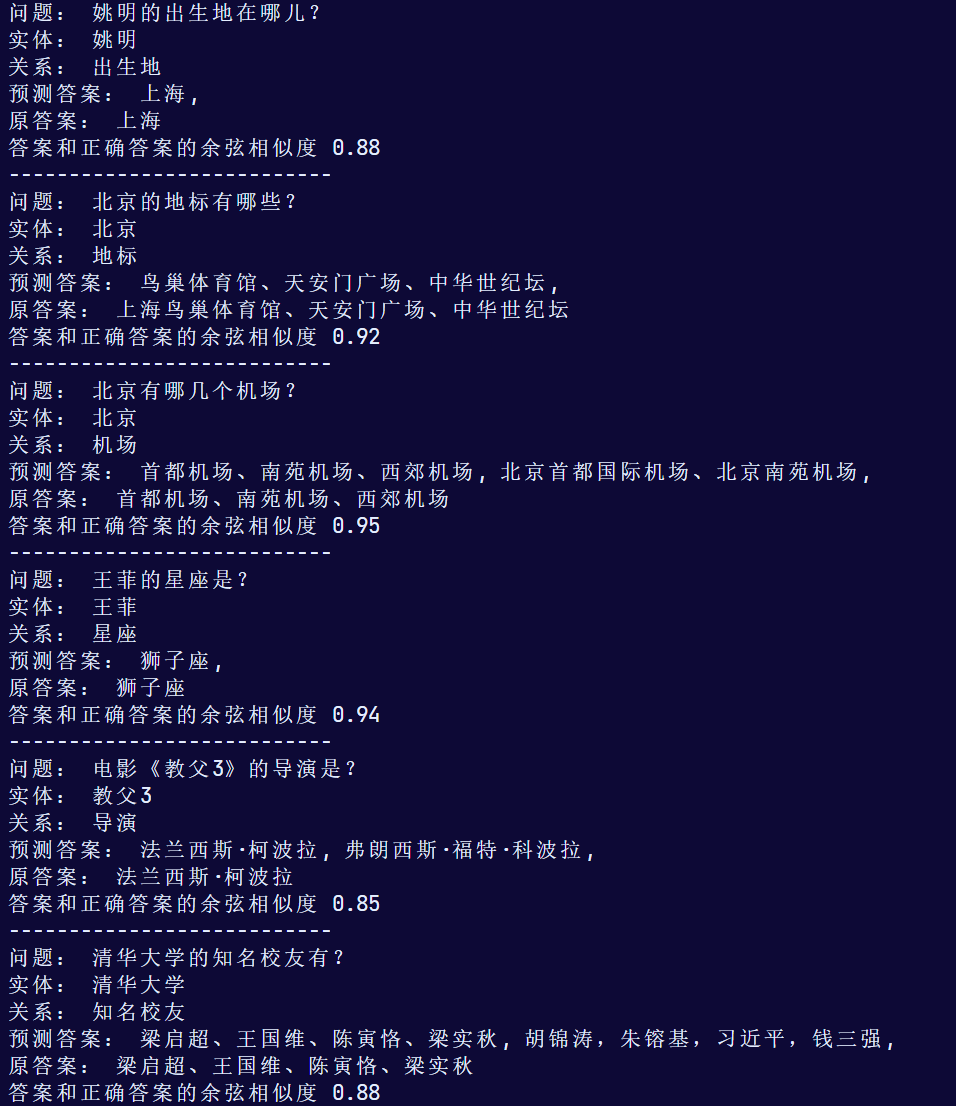
我决定找一找当今用来给LLM做RAG的开源嵌入模型。

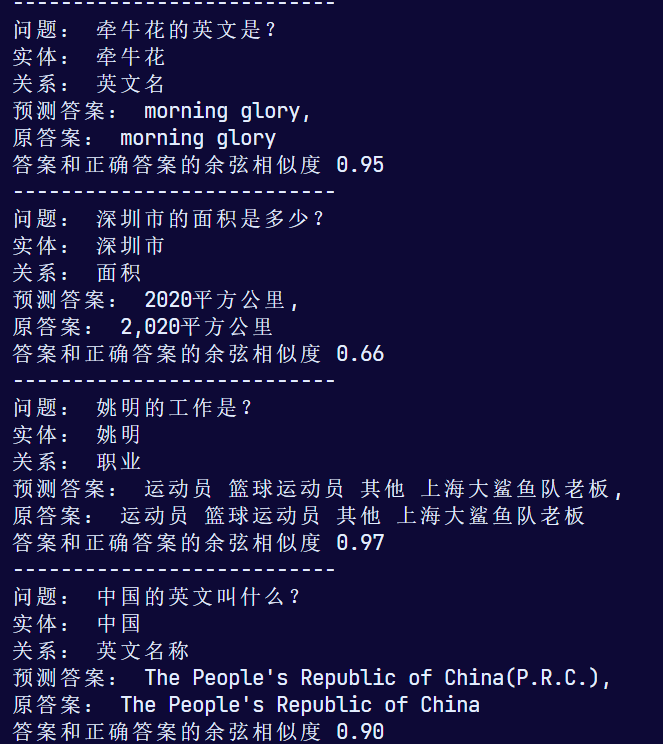
注意到 <https://huggingface.co/BAAI/bge-base-zh-v1.5> 被 Open WebUI 使用，很适合做RAG

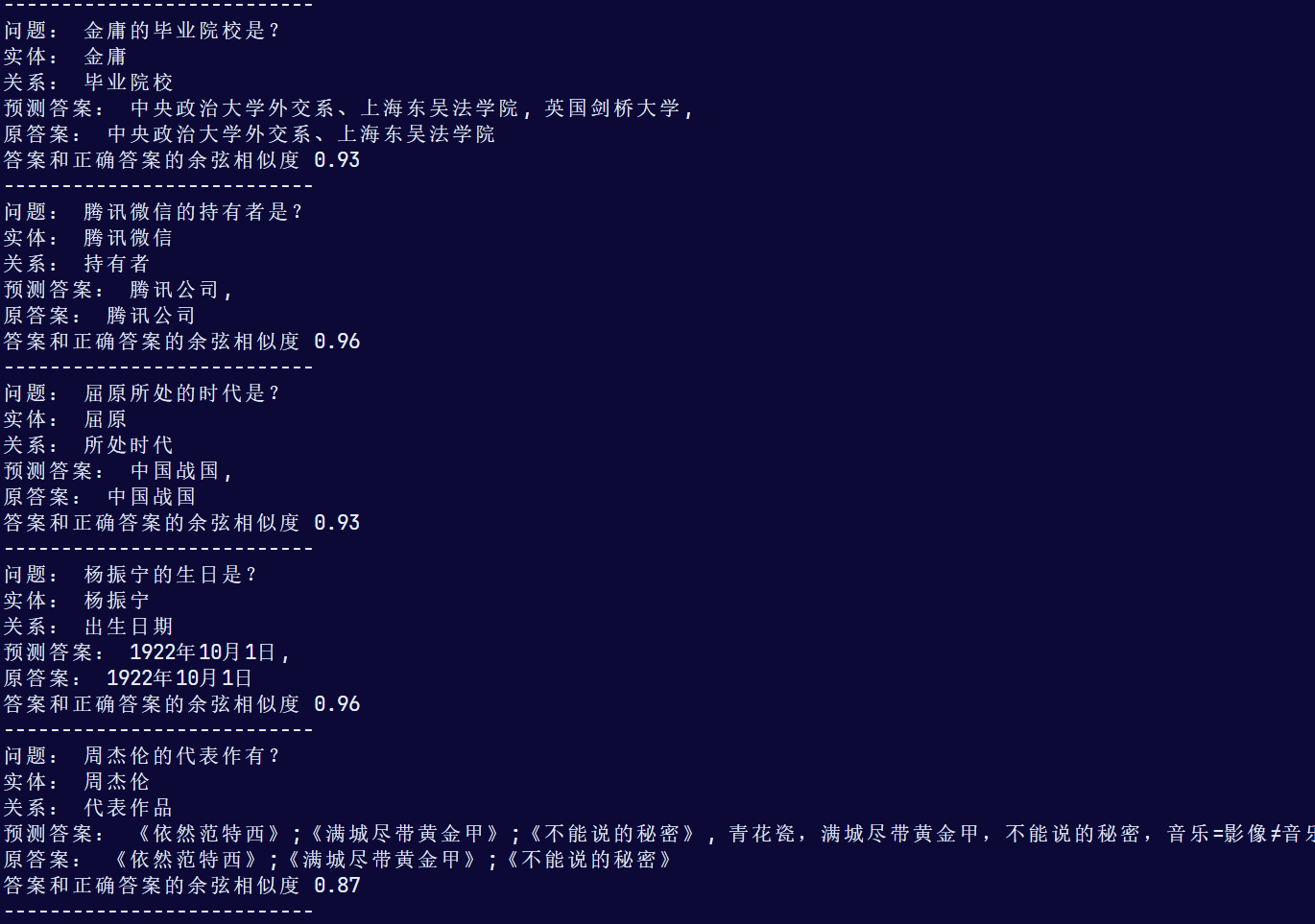
所以我直接替换为

model = AutoModel.from\_pretrained("BAAI/bge-base-zh-v1.5")  
tokenizer = AutoTokenizer.from\_pretrained("BAAI/bge-base-zh-v1.5")

这一次回答几乎完全正确！至少问题关系没有理解错，信息可能少一些多一些，但都是从知识库知识回答的







可以看到我们选择的模型显著优于BERT！！我们的改进非常有效。