大模型的局限性

2024年7月17日 18:01

局限性：

1、虽然模型掌握大量数据，但是他们的结构和参数数量使得对其进行修改、微调或重新训练变得异常困难，且相关成本相对可观

2、大型语言模型的应用依赖于构建Prompt来引导模型生成所需文本。

现存问题：

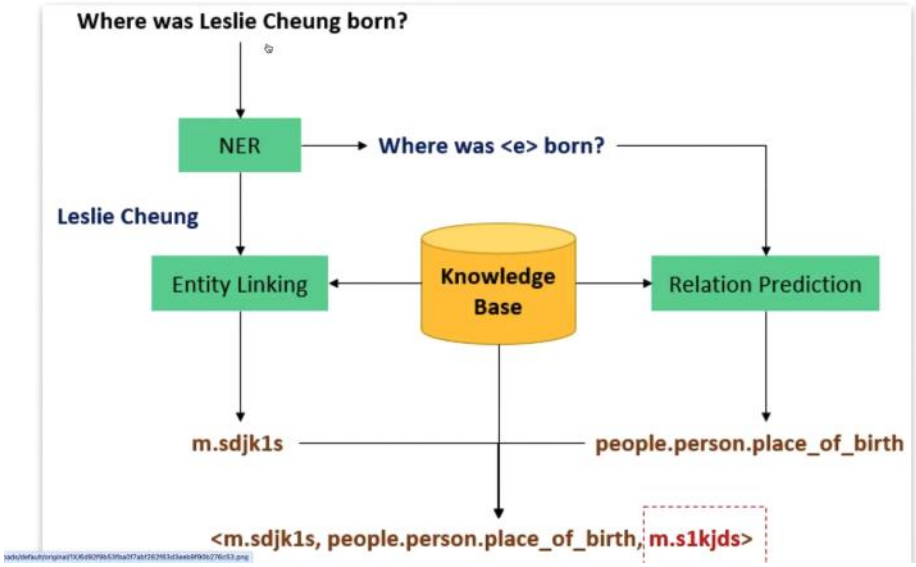
1、模型幻觉问题：生成内容可能不准确

2、时效性问题：生成的内容不具备时效性

3、数据安全性问题：可能存在敏感信息泄露风险

知识库问答（KBQA）

一种对话系统方法，旨在利用结构化的知识库进行自然语言问题的回答，这种方法基于一个存储在图数据库中的知识库，通常以三元组的形式表示为<主题、关系、对象>，其中每个三元组都附带相关的属性信息。

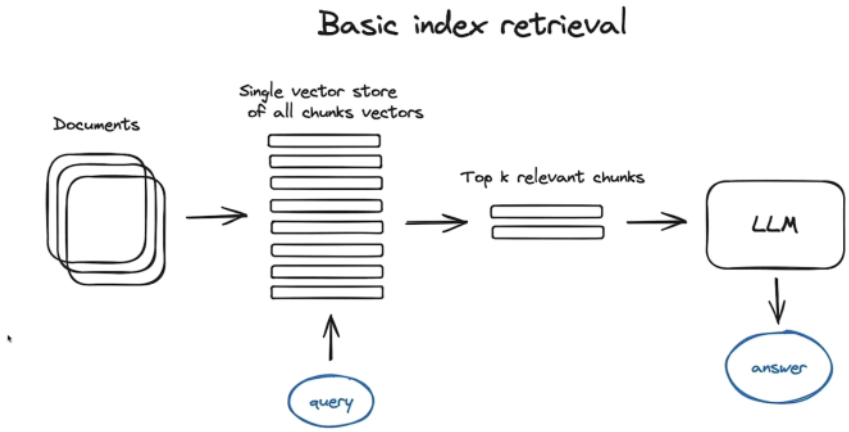


该方法依赖于信息抽取。

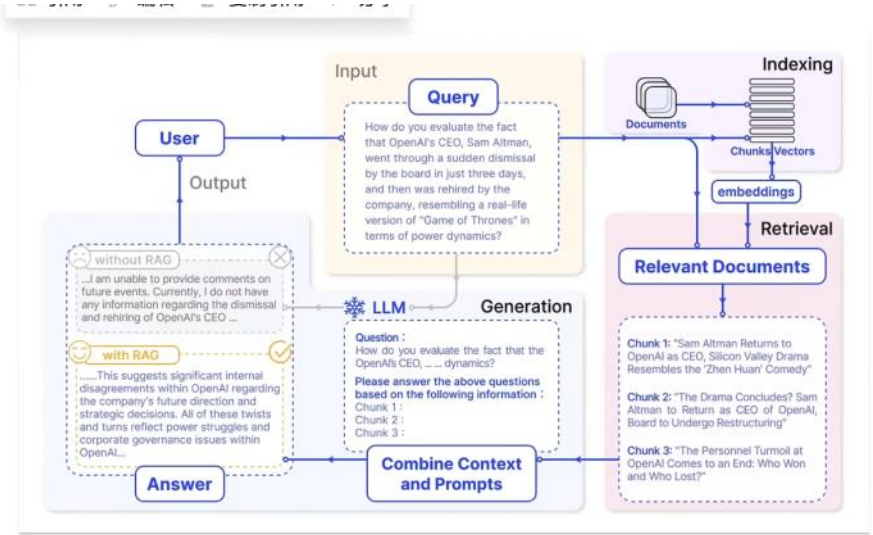
RAG介绍

2024年7月17日 18:03

RAG为检索增强生成，弥补大型语言模型（LLM）的局限性方面取得了显著进展，尤其是解决幻觉问题和时效性问题方面。
RAG技术通过引入外部知识库的检索机制，成功提升了生成内容的准确性、相关性和时效性



RAG被构建为一个应用于大型语言模型的框架，其目标是通过结合大模型的生成能力和外部知识库的检索机制，提升自然语言处理任务的效果。



RAG框架的最终输出被设计为一种协同工作模式，将检索到的知识融合到大型语言模型的生成过程中，在应对任务特定问题时，RAG会生成一段标准化的句子，引导大模型进行回答，如：

你是一个{task}方面的专家，请结合给定的资料，并回答最终问题。请如实回答，如果问题再资料中找不到答案，则回答不知道

问题：{question}

资料：
-{information1}

-{information2}
-{information3}

其中，{task}代表任务的领域或主题，{question}是最终要回答的问题，而{information1}等则是提供给模型的外部知识库中的具体信息。

RAG实现步骤

2024年7月18日 9:46

使用RAG，主要包括信息检索和大型语言模型调用两个关键过程：

1、信息检索通过连接外部知识库，获取与问题相关的信息。

2、大型语言模型调研用于将这些信息整合到自然语言生成的过程中，以生成最终回答。

RAG流程：

1、问题理解：准确把握用户意图

2、知识检索：从知识库中检索相关的知识

3、答案生成

ChatGLM API

2024年7月17日 18:02

ChatGLM

官网：https://open.bigmodel.cn/

API文档：https://open.bigmodel.cn/dev/api

工作台：https://open.bigmodel.cn/overview

对话API

2024年7月18日 10:37

请求参数说明：

参数	类型	是否必填	描述
messages	Array	是	包含对话的消息列表
model	String	是	要使用的模型的ID
max_tokens	Integer/null	否	可以在聊天中完成的最大令牌数
seed	Integer/null	否	如果指定，系统将尽力进行确定性采样，以使具有相同seed和参数的重复请求应返回相同的结果。
n	Integer/null	否	为每个输入消息生成的聊天完成选择的数量

返回结果字段说明：

参数	类型	描述
id	String	用于唯一标识聊天完成的标识符
choices	Array	聊天完成选择的列表。如果n大于1，则可以有多个选择
created	Integer	聊天完成创建的Unix时间戳
model	String	用于聊天的模型
System_fingerprint	String	该指纹表示模型运行时的后端配置，可与seed请求参数一起使用
Finish_reason	String	表示聊天完成的原因

具体代码：

方法一：

```
fromzhipuaiimportZhipuAI

client=ZhipuAI(api_key="c3122cd0e5c11b9312ef7c6559826bc2.f3eufPOzX6u7U4j0")#填写您自己的
APIKey
response=client.chat.completions.create(
model="glm-4",#填写需要调用的模型名称
messages=[
{"role":"user","content":"作为一名营销专家，请为智谱开放平台创作一个吸引人的slogan"},
{"role":"assistant","content":"当然，为了创作一个吸引人的slogan，请告诉我一些关于您产品的信
息"},
{"role":"user","content":"智谱AI开放平台"},
{"role":"assistant","content":"智启未来，谱绘无限一智谱AI，让创新触手可及!"},
{"role":"user","content":"创造一个更精准、吸引人的slogan"}
],
)
print(response.choices[0].message)
```

方法二：（暂不行）

```
importrequests
importtime
importjwt

defgenerate_token(apikey:str,exp_seconds:int):
try:
id,secret=apikey.split(".")
exceptExceptionase:
raiseException("invalidapikey",e)

payload={
"api_key":id,
"exp":int(round(time.time()*1000))+exp_seconds*1000,
"timestamp":int(round(time.time()*1000)),
```

```
}

returnjwt.encode(
payload,
secret,
algorithm="HS256",
headers={"alg":"HS256","sigh_type":"SIGN"},
)


url="https://open.bigmodel.cn/api/paas/v4/embeddings"
headers={
"Content-Type":"application/json",
"Authorization":generate_token("5df05402dfe79a036798a084202ded4b.ZfNFzCP5bApmNimG",1000)
}

data={
"model":"embedding-2",
"messages":[{"role":"users","content":"你好"}]
}

response=requests.post(url,headers=headers,json=data)

print("Status:",response.status_code)
print("JSON:",response.json())
```

读取问答数据集

2024年7月17日 18:03

逻辑：先读取问题，再根据问题找到具体页码，再分析该页内容回答问题

代码：

```
importjson
importpdfplumber
importtorch
importtransformers
importjieba

questions=json.load(open("questions.json",encoding='utf-8'))
forquestioninrange(len(questions)):
print(questions[question])



pdf=pdfplumber.open("初赛训练数据集.pdf")
len(pdf.pages)#页数
pdf.pages[0].extract_text()#读取第一页的内容(可复制的pdf)
print(pdf.pages[0].extract_text())

pdf_content=[]
forpage_idxinrange(len(pdf.pages)):#从0开始
pdf_content.append({
'page':'page_'+str(page_idx+1),#第一页为1
'content':pdf.pages[page_idx].extract_text()
})
```

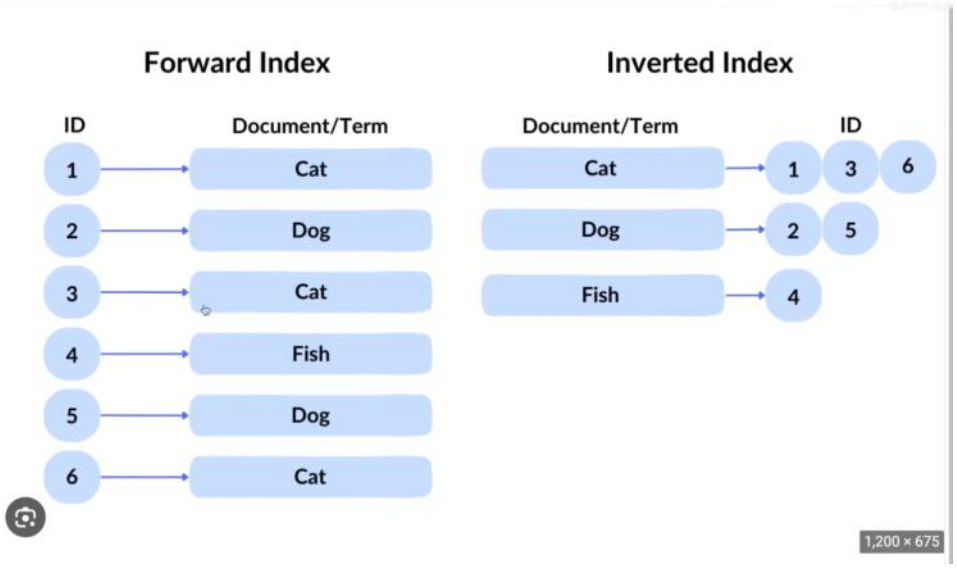
文本检索流程

2024年7月17日 18:04

搜索引擎的本质是文本检索的过程

文本检索是一个多步骤的过程，其核心是构建倒排索引以实现高效的文本检索：
步骤一（文本预处理）：对原始文本进行清理和规范化，包括去除停用词，标点符号等噪声，并将文本统一转为小写。接着，采用词干化或词形还原等技术，将单词转换为基本形式，以减少词汇的多样性，为后续建立索引做准备。
步骤二（文本索引）：构建倒排索引是文本检索的关键步骤，通过对文档集合进行分词，得到每个文档的词项列表，并为每个词项构建倒排列表，记录高喊该词项的文档及其位置信息。这种结构使得在查询时能够快速找到包含查询词的文档，为后续的文本检索奠定了基础。
步骤三（文本检索）：查询处理阶段，用户查询经过预处理后，与建立的倒排索引进行匹配。计算查询中每个词项的权重，并利用检索算法（TFIDF or BM25）对文档进行排序，将相关性较高的文档排在前面。（检索算法可理解为打分算法）

倒排列表示例：



TFIDF

2024年7月19日 10:28

TFIDF是一种用于信息检索和文本挖掘的常用权重计算方法，旨在衡量一个词项对于一个文档集合中某个文档的重要性。该方法主要结合了两个方面的信息：词项在文档中的频率（TF）和在整个文档集合中逆文档频率（IDF）

1、词项在文档中的频率（TF）：针对单个文档

$$TF(t,d) = \frac{\text{词项t在文档d中出现的次数}}{\text{文档d中所有词项的总数}}$$

TF表示了一个词项在文档中的相对频率，即在文档中出现的次数相对于文档总词项的比例。

2、逆文档频率（IDF）：针对整个文档集合

$$IDF(t) = \log\left(\frac{\text{文档集合中的文档总数}}{\text{包含词项t的文档数}+1}\right)$$

IDF表示了一个词项在整个文档集合中的稀有程度。若词项在许多文档中都出现，则IDF值较低。

3、TFIDF的计算

$$TFIDF(t,d,D) = TF(t,d) \times IDF(t)$$

D表示文档集合。TFIDF的最终值是将词项在文档中的频率和在整个文档集合中的逆文档频率相乘，这样可以得到一个更全面的评估，既考虑了在文档中的重要性，也考虑了在整个文档中的稀有性。

具体实现：

```
#对提问和PDF内容进行分词
#创建x['question']获取JSON文件中question对应的内容，赋予question_words
#jieba.lcut()函数用于将这段文本进行分词，即将一个句子分割成多个词语，并返回一个包含所有词语的列表。
#join()函数用于将列表中的元素连接成一个字符串，元素之间用指定的分隔符分隔。
question_words=[''.join(jieba.lcut(x['question']))forxinquestions]
pdf_content_words=[''.join(jieba.lcut(x['content']))forxinpdf_content]

#fit方法会分析传入的文本数据，构建一个词汇表(包含所有单词），并为词汇表中的每个词计算IDF值。
tfidf=TfidfVectorizer()
tfidf.fit(question_words+pdf_content_words)


#提取TFIDF
#transform方法将question_words和pdf_content_words中的文本转换为TF-IDF特征矩阵。
#transform方法会利用fit方法学习到的词汇表和IDF值，将新的文本数据转换为特征矩
阵。question_feat和pdf_feat分别包含了问题和PDF内容的TF-IDF特征矩阵。
#question_feat中，包含301行，对应301个问题，有好多列，每一列对应一个单词，每个值代表每个单词
```

```
的TFIDF值，也就是该词的重要性
#pdf_content_feat中，包含354行，对应354页，有好多列，每一列对应一个单词，每个值代表每个单词
的TFIDF值，也就是该词的重要性
question_feat=tfidf.transform(question_words)
pdf_content_feat=tfidf.transform(pdf_content_words)
```

得到question每一个问题的每个词与pdf_content每一页的每个词分别的TFIDF值

BM25

2024年7月22日 17:13

BM25Okapi是BM25算法的一种变体，它在信息检索中用于评估文档与查询之间的相关性。

原理与打分方式概述：

$$\text{score} = \sum_{q \in \text{query}} \left(\text{IDF}(q) \cdot \frac{q\_freq \cdot (k1 + 1)}{q\_freq + k1 \cdot (1 - b + b \cdot \frac{\text{doc\_len}}{\text{avgdl}})} \right)$$

K1：控制词项频率对分数的影响，通常设置为1.5

b：控制文档长度对分数的影响，通常设置为0.75

BM25Okapi的打分过程基于以下三个因素：词项在文档中的频率（TF）、文档的长度（doc_len）以及逆文档频率（IDF），BM25Okapi的打分公式综合考虑以上三个要素，通过对每个词项的打分求和得到最终的文档与查询的相关性系数。

具体实现：

```
#提取BM25
#对文本进行切分-采用单词切分，通过BM25Okapi导入bm25中
pdf_content_words=[jieba.lcut(x['content'])forxinpdf_content]
bm25=BM25Okapi(pdf_content_words)

#遍历每一个问题
forquery_idxinrange(len(questions)):
#bm25.get_scores返回第query_idx个问题与每一页的bm25值
doc_scores=bm25.get_scores(jieba.lcut(questions[query_idx]['question']))
max_score_page_idx=doc_scores.argsort()[-1]+1
questions[query_idx]['reference']='page_'+str(max_score_page_idx)
```

答案检索

2024年7月22日 16:58

找到每个问题最相关的那一页

通过每个问题的特征矩阵与每一页的特征矩阵点积得到相关度

具体实现：

```
#进行归一化
#将所有数据放在统一的尺度上
question_feat=normalize(question_feat)
pdf_content_feat=normalize(pdf_content_feat)

#检索进行排序
forquery_idx,featinenumerate(question_feat):
#for循环中，query_idx代表问题序号，feat代表该问题中每一个词对应的TFIDF值
#使用feat@pdf_content_feat.T计算第query_idx个问题的特征向量与PDF内容中每个页面的特征向量之间的点积。点积是衡量两个向量相似度的一种方法
#score表示第query_idx个问题与每个页面内容各自的相关性
score=feat@pdf_content_feat.T

#score.toarray()[0]将score从稀疏矩阵格式转换为NumPy数组，
score=score.toarray()[0]

#通过score.argsort()获取按得分排序的页面索引。argsort()函数返回的是数组值从小到大的索引值。
#[-1]取最后一位，得到最相关的页数
max_score_page_idx=score.argsort()[-1]+1
#赋予reference最相关的页数
questions[query_idx]['reference']='page_'+str(max_score_page_idx)

#生成提交结果
#https://competition.coggle.club/
withopen('submit.json','w',encoding='utf8')asup:
json.dump(questions,up,ensure_ascii=False,indent=4)
#open('submit.json','w',encoding='utf8')打开一个名为submit.json的文件用于写入（'w'模式）。如果文件不存在，它会被创建；如果文件已存在，它会被覆盖。encoding='utf8'指定文件使用UTF-8编码。
#json.dump()函数用于将Python对象（这里是questions）转换为JSON格式，并写入到指定的文件对象
print("建立submit.json成功")
```

向量介绍

2024年7月17日 18:04

嵌入模型：将单词以及句子转换到一个空间内，在空间内可以计算单词间的距离。

语义检索流程

2024年7月22日 15:14

语义检索是通过词嵌入和句子嵌入的技术（不需要单词完全相同），将文本表示为语义丰富的向量。通过相似度计算和结果排序找到最相关的文档。
用户查询经过自然语言处理，最终系统返回经过排序的相关文档，提供用户友好的信息展示。
流程：

```
graph TD
    A[加载模型] -->|Sentence Transformer| B((编码文本))
    B -->|问题句子| C[问题Embeddings]
    B -->|PDF内容句子| D[PDF内容Embeddings]
    C -->|标准化| C
    D -->|标准化| D
    C -->|相似度计算| E[相似度矩阵]
    D -->|相似度计算| E
    E -->|排序| F[排序后的相似度]
    F -->|选取最大值| G[最相似的页码]
    G -->|写入结果| H[生成提交结果]
```

其核心为文本编码模型。

文本编码模型

2024年7月22日 15:22

大多数语义检索系统采用预训练模型进行文本编码，实现文本嵌入，其中最为常见的是基于BERT的模型，或使用GPT等。
这些预训练模型通过大规模语料上进行训练，能够捕捉词语和句子之间的复杂语义关系。
实现文本嵌入后，可基于向量做向量检索

文本编码示例：
采用M3E模型

```
fromsentence_transformersimportSentenceTransformer
#对文本进行编码
#normalize_embeddings=True表示默认归一化
model=SentenceTransformer('moka-ai/m3e-base')
question_embeddings=model.encode(question_sentences,normalize_embeddings=True)
pdf_embeddings=model.encode(pdf_content_sentences,normalize_embeddings=True)
```

文本切分方法

2024年7月22日 16:15

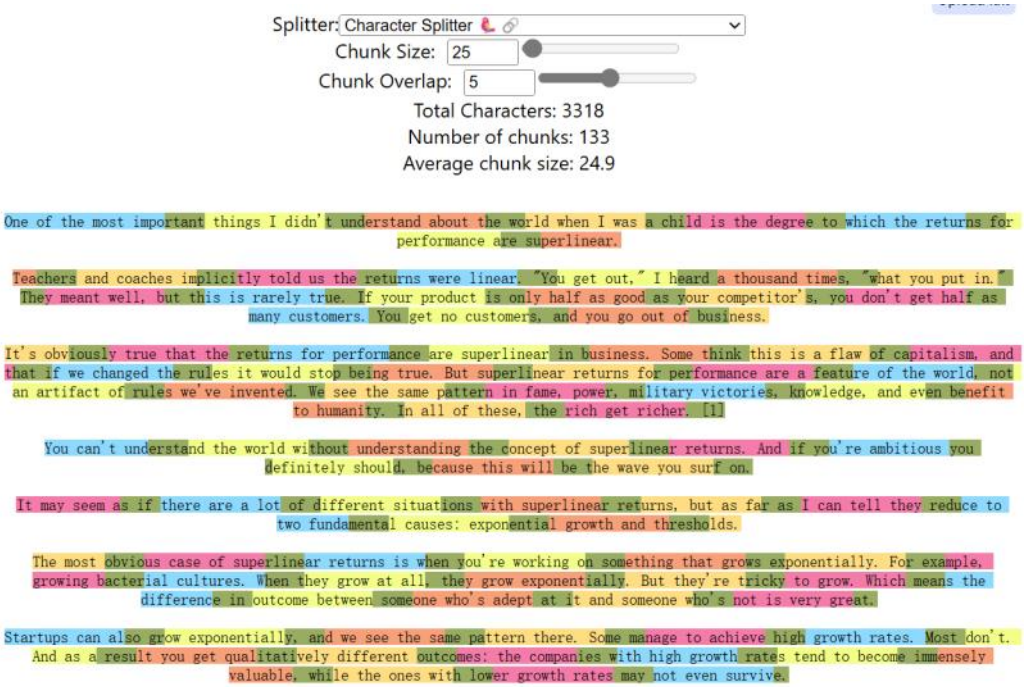
文本的长度影响了文本编码的结果，短文本和长文本再编码成向量时可能表达不同的语义信息。

名称	分割依据	描述
递归式分割器	一组用户定义的字符	递归地分割文本。递归分割文本的目的是尽量保持相关的文本段落相邻。这是开始文本分割的推荐方式。
HTML分割器	HTML特定字符	基于HTML特定字符进行文本分割。特别地，它会添加有关每个文本块来源的相关信息（基于HTML结构）。
Markdown分割器	Markdown特定字符	基于Markdown特定字符进行文本分割。特别地，它会添加有关每个文本块来源的相关信息（基于Markdown结构）。
代码分割器	代码（Python、JS）特定字符	基于特定于编码语言的字符进行文本分割。支持从15种不同的编程语言中选择。
Token分割器	Tokens	基于Token进行文本分割。存在一些不同的Token计量方法。
字符分割器	用户定义的字符	基于用户定义的字符进行文本分割。这是较为简单的分割方法之一。
语义分块器	句子	首先基于句子进行分割。然后，如果它们在语义上足够相似，就将相邻的句子组合在一起。

对于自然语言，可以推荐Token分割器，结合Chunk Size和Overlap Size可以得到不同的切分：

Chunk Size（块大小）：表示将文本划分为较小块的大小，这是分割后每个独立文本块的长度或容量。块大小的选择取决于应用的需求和对文本结构的理解。

Overlap Size（重叠大小）：指相邻两个文本块之间的重叠部分的大小。在切割文本时，通常希望保留一些上下文信息，重叠大小就是控制这种上下文保留的参数。



文本切分示例：

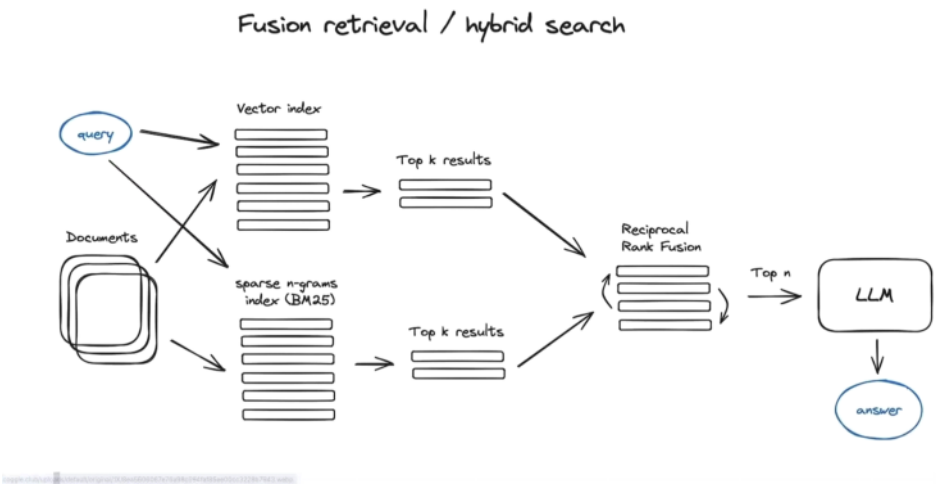
```
defsplit_text_fixed_size(text,chunk_size):
return[text[i:i+chunk_size]foriinrange(0,len(text),chunk_size)]


#从0遍历到最后一页
forpage_idxinrange(len(pdf.pages)):
#读取第page_idx页的text
text=pdf.pages[page_idx].extract_text()
#对第page_idx页的text进行切分，以40为chunk_size
forchunk_textinsplit_text_fixed_size(text,40):
#得到每一句切分后的小块以及对应的页码，如有需要还可以得到每个小块对应的序号
pdf_content.append({
'page':'page_'+str(page_idx+1),
'content':chunk_text
})
```

多路召回逻辑

2024年7月17日 18:04

多路召回逻辑是文本检索中常用的一种策略，其目的是通过多个召回路径（或方法）综合获取候选文档，以提高检索的全面性和准确性。
单一的召回方法可能由于模型特性或数据特点而存在局限性，多路召回逻辑引入了多个召回路径，每个路径采用不同的召回方法。

实现方法1：将BM25的检索结果和语义检索结果按照排名进行加权
实现方法2：按照段落、句子、页不同的角度进行语义编码进行检索，综合得到检索结果。



重排序逻辑

2024年7月22日 17:09

重排序逻辑（BM25 + BGE Rerank）是文本检索邻域中一种重要的策略，主要用于优化原有文本检索方法返回的候选文档排序，以提高最终的检索结果。在传统的文本检索文本中，往往采用打分的逻辑，如采用BERT嵌入向量之间的相似度。而重排序逻辑引入了更为复杂的文本交叉方法，通过特征交叉得到更进一步的打分，从而提高排序的准确性。

重排序逻辑常常使用更为强大的模型，如交叉编码器（cross-encoder）模型。这类模型能够更好地理解文本之间的交叉关系，捕捉更复杂的语义信息。
首先通过传统的嵌入模型获取初始的Top-k文档，然后使用重排序逻辑对这些文档进行重新排序。这样可以在保留初步筛选文档的基础上，更精确地排列它们的顺序。

具体实现：

```
#-------------------------------------------------------------------------------------------
--------------------------
#加载重排序模型
print("加载重排序模型中.....")
tokenizer=AutoTokenizer.from_pretrained('BAAI/bge-reranker-large')
rerank_model=AutoModelForSequenceClassification.from_pretrained('BAAI/bge-reranker-large')
rerank_model.cuda()
print("加载重排序模型完成")

#先进行BM25检索
print("BM25检索中.....")
pdf_content_words=[jieba.lcut(x['content'])forxinpdf_content]
bm25=BM25Okapi(pdf_content_words)

forquery_idxinrange(len(questions)):
doc_scores=bm25.get_scores(jieba.lcut(questions[query_idx]["question"]))
#取TOP3的页码
max_score_page_idxs=doc_scores.argsort()[-3:]

#对TOP3进行重排序
#构建文本对，将用户提问与检索得到的文本拼接为文本对,并进行编码
#pairs中，每个问题对应三页在BM25检索下最相关的页面
pairs=[]
foridxinmax_score_page_idxs:
pairs.append([questions[query_idx]["question"],pdf_content[idx]['content']])
print(pairs)

inputs=tokenizer(pairs,padding=True,truncation=True,return_tensors='pt',max_length=512)

#将文本对进行正向传播，得到匹配度的打分
withtorch.no_grad():
inputs={key:inputs[key].cuda()forkeyininputs.keys()}
scores=rerank_model(**inputs,return_dict=True).logits.view(-1,).float()

#得到重排序的结果，在TOP3中再选取匹配度最高的文本
max_score_page_idx=max_score_page_idxs[scores.cpu().numpy().argmax()]
```

```
questions[query_idx]['reference']='page_'+str(max_score_page_idx+1)

#-------------------------------------------------------------------------------------------------
-------------------------
#生成结果
print("生成结果文件中.....")
withopen('submit.json','w',encoding='utf8')asup:
json.dump(questions,up,ensure_ascii=False,indent=4)
print("生成结果文件完成")
```

代码中，先对问题与PDF进行BM25检索（简单），并在最后选取相关度TOP3的页面
然后在pairs中将每个问题与其三个页面建立文本对
通过重排序模型对pairs中每一项的页面通过更复杂的方法对三个文本进行重排序
最后保留TOP3中的TOP1

文本问答Promopt优化

2024年7月17日 18:04

具体实现：

```
importjson
importpdfplumber
importtorch
importtransformers
importjieba
importsklearn
importnumpy
fromsentence_transformersimportSentenceTransformer
fromtransformersimportAutoModelForSequenceClassification,AutoTokenizer
fromsklearn.feature_extraction.textimportTfidfVectorizer
fromsklearn.preprocessingimportnormalize
fromrank_bm25importBM25Okapi
fromzhipuaiimportZhipuAI

print(torch.cuda.is_available())


#-----------------------------------------------------------------------------------------------
---------------------------
#定义访问ChatGLM的函数
defask_alm(content):
client=ZhipuAI(api_key="c3122cd0e5c11b9312ef7c6559826bc2.f3eufPOzX6u7U4j0")#填写您自己的
APIKey
response=client.chat.completions.create(
model="glm-4",#填写需要调用的模型名称
messages=[
{"role":"user","content":content},
],
)
returnresponse.choices[0].message.content


#-----------------------------------------------------------------------------------------------
---------------------------
#读取问题的JSON文件
#共有301个问题
print("读取文件中.....")
questions=json.load(open("questions.json",encoding='utf-8'))
print(len(questions))

#读取数据集的PDF文件
#共有354页
pdf=pdfplumber.open("初赛训练数据集.pdf")
print(len(pdf.pages))
pdf_content=[]
forpage_idxinrange(len(pdf.pages)):
#从0开始
pdf_content.append({
'page':'page_'+str(page_idx+1),
#第一页为1，也就是在pdf的content中，第0项对应第1页
'content':pdf.pages[page_idx].extract_text()
})
```

```
#提取问题及PDF文档中的句子
question_sentences=[x['question']forxinquestions]
pdf_content_sentences=[x['content']forxinpdf_content]
print("读取文件完成")

#-------------------------------------------------------------------------------------------
--------------------------
#加载重排序模型
print("加载重排序模型中.....")
tokenizer=AutoTokenizer.from_pretrained('BAAI/bge-reranker-large')
rerank_model=AutoModelForSequenceClassification.from_pretrained('BAAI/bge-reranker-large')
rerank_model.cuda()
print("加载重排序模型完成")

#先进行BM25检索
print("BM25检索中.....")
pdf_content_words=[jieba.lcut(x['content'])forxinpdf_content]
bm25=BM25Okapi(pdf_content_words)

forquery_idxinrange(len(questions)):
doc_scores=bm25.get_scores(jieba.lcut(questions[query_idx]["question"]))
#取TOP3的页码
max_score_page_idxs=doc_scores.argsort()[-3:]

#对TOP3进行重排序
#构建文本对，将用户提问与检索得到的文本拼接为文本对,并进行编码
#pairs中，每个问题对应三页在BM25检索下最相关的页面
pairs=[]
foridxinmax_score_page_idxs:
pairs.append([questions[query_idx]["question"],pdf_content[idx]['content']])

inputs=tokenizer(pairs,padding=True,truncation=True,return_tensors='pt',max_length=512)

#将文本对进行正向传播，得到匹配度的打分
withtorch.no_grad():
inputs={key:inputs[key].cuda()forkeyininputs.keys()}
scores=rerank_model(**inputs,return_dict=True).logits.view(-1,).float()

#得到重排序的结果，在TOP3中再选取匹配度最高的文本
max_score_page_idx=max_score_page_idxs[scores.cpu().numpy().argmax()]
questions[query_idx]['reference']='page_'+str(max_score_page_idx+1)

#-------------------------------------------------------------------------------------------
--------------------------
#生成prompt
prompt='''你是一个汽车专家，帮我结合给定的资料，回答一个问题。如果问题无法从资料中获得，请输
出结合给定的资料，无法回答问题。
资料：{0}

问题：{1}
'''.format(
pdf_content[max_score_page_idx]['content'],
questions[query_idx]["question"]
)
answer=ask_alm(prompt)
print(answer)
questions[query_idx]['answer']=answer
print("导入答案完成")
```

```
#-------------------------------------------------------------------------------------------------
-------------------------
#生成结果
foriinrange(len(questions)):
print(questions[i])
```