Simple_RAG 实现

1. 配置虚拟环境，安装依赖

1.1 构建环境

```
conda create -n rag_test python=3.10
source activate rag_test
```

1.2 安装依赖

requirements.txt 文件

```
faiss-cpu
faiss-gpu==1.7.2
argparse==1.1
transformers==4.37.2
torch==2.0.0
tqdm==4.66.2
```

运行以下脚本

```
pip install -r requirements.txt -i https://pypi.tuna.tsinghua.edu.cn/simple --
ignore-installed
```

Embedding 模型

- embedding模型使用下载好的m3e-large模型

```
tokenizer = AutoTokenizer.from_pretrained('m3e-large')
model = AutoModel.from_pretrained('m3e-large').to('cuda:3')
```

2. 代码

2.1 导包

```python
import os
import argparse
import faiss
from transformers import AutoTokenizer, AutoModel, AutoModelForCausalLM
import torch
from tqdm import tqdm
```

2.1.1 加载transformers出现报错

```
ImportError: cannot import name 'COMMON_SAFE_ASCII_CHARACTERS' from
'charset_normalizer.constant'
```

- 解决方案

```
pip install --upgrade charset-normalizer
```

2.2 文件处理函数

```python
def process_file(file_path):
    with open(file_path, encoding = 'utf-8') as f:
        text = f.read()
        sentences = text.split('\n')
        return text, sentences

process_file(os.path.join(os.getcwd(),'requirements.txt'))
```

Output:

```
('faiss-cpu\nfaiss-gpu==1.7.2\nargparse==1.1\ntransformers==4.37.2\ntorch==2.0.0\ntqdm==4.66.2',
 ['faiss-cpu',
  'faiss-gpu==1.7.2',
  'argparse==1.1',
  'transformers==4.37.2',
  'torch==2.0.0',
  'tqdm==4.66.2'])
```

2.3 构建prompt函数

```python
def generate_rag_prompt(data_point):
    return f'''### Instruction:
    {data_point['instruction']}
    ### Input:
    {data_point['input']}
    ### Response:
    '''
```

2.4 生成embedding

```python
class DocumentEmbedder:
    def __init__(self, max_length = 128, max_number_of_sentences = 3):
        self.model = model
        self.tokenizer = tokenizer
        self.max_length = max_length
        self.max_number_of_sentences = max_number_of_sentences

    def get_document_embeddings(self,sentences):
        sentences = sentences[:self.max_number_of_sentences]
        encoded_input = self.tokenizer(sentences,
                                       padding = True,
                                       truncation = True,
                                       max_length = 128,
                                       return_tensors = 'pt')
        encoded_input.to('cuda:3')
        with torch.no_grad():
            model_output = self.model(**encoded_input)
        return torch.mean(model_output.pooler_output,dim = 0,keepdim = True)
```

2.5 提前准备好知识库

```python
from langchain_text_splitters import RecursiveCharacterTextSplitter
from tqdm import tqdm
text_splitter = RecursiveCharacterTextSplitter(
    # Set a really small chunk size, just to show.
    chunk_size=500,
    chunk_overlap=20,
    length_function=len,
    is_separator_regex=False,
)
texts = text_splitter.split_text(text)
for i in range(len(texts)):
    with open(f'RAG-file/chunks/chunks-{i}','w') as f:
        f.write(texts[i])
```

- 将一篇长文本txt文件进行500个词分词写入chunks文件夹内，本次使用的文本是self-RAG那篇文章的文本

2.6 大模型类

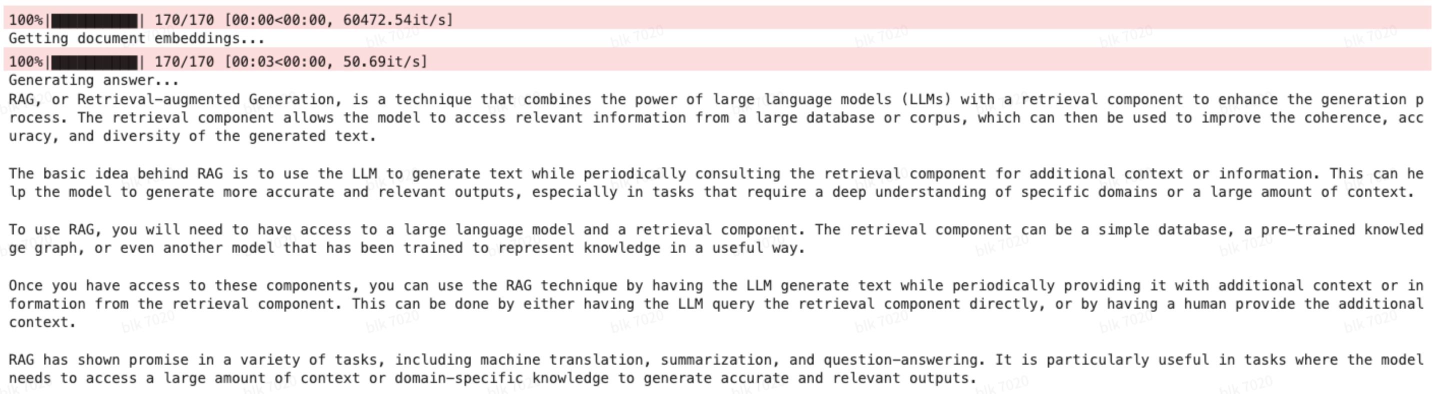
- 这里调用了zhipuAI的API

主函数定义

```python
if __name__ == '__main__':
    documents = {}
    for idx, file in enumerate(tqdm(os.listdir('RAG-file/chunks'))):
        current_filepath = os.path.join('RAG-file/chunks',file)
        text, sentences = process_file(current_filepath)
        documents[idx] = {'file_path':file,
                          'sentences':sentences,
                          'document_text':text}

    print('Getting document embeddings...')
    document_embedder = DocumentEmbedder()
    embeddings = []
    for idx in tqdm(documents):
        
embeddings.append(document_embedder.get_document_embeddings(documents[idx]
['sentences']))
    embeddings = torch.concat(embeddings,dim = 0).data.cpu().numpy()
    embedding_dimensions = embeddings.shape[1]
    faiss_index = faiss.IndexFlatIP(int(embedding_dimensions))
    faiss_index.add(embeddings)

    question = 'Introduce to me what is RAG and how to use it'

    query_embedding = document_embedder.get_document_embeddings([question])
    distances, indices = faiss_index.search(query_embedding.data.cpu().numpy(),k=int(5))

    context = ''
    for idx in indices[0]:
        context += documents[idx]['document_text']

    rag_prompt = generate_rag_prompt({'instruction':question,
                                      'input':context})
    print('Generating answer...')
    answer = GenerativeModel(rag_prompt)
    print(answer)
```

输出结果

- 输入问题：
- Introduce to me what is RAG and how to use it
- 输出：



```
100%|██████████| 170/170 [00:00<00:00, 60472.54it/s]
Getting document embeddings...
100%|██████████| 170/170 [00:03<00:00, 50.69it/s]
Generating answer...
RAG, or Retrieval-augmented Generation, is a technique that combines the power of large language models (LLMs) with a retrieval component to enhance the generation p
rocess. The retrieval component allows the model to access relevant information from a large database or corpus, which can then be used to improve the coherence, acc
uracy, and diversity of the generated text.

The basic idea behind RAG is to use the LLM to generate text while periodically consulting the retrieval component for additional context or information. This can he
lp the model to generate more accurate and relevant outputs, especially in tasks that require a deep understanding of specific domains or a large amount of context.

To use RAG, you will need to have access to a large language model and a retrieval component. The retrieval component can be a simple database, a pre-trained knowled
ge graph, or even another model that has been trained to represent knowledge in a useful way.

Once you have access to these components, you can use the RAG technique by having the LLM generate text while periodically providing it with additional context or in
formation from the retrieval component. This can be done by either having the LLM query the retrieval component directly, or by having a human provide the additional
context.

RAG has shown promise in a variety of tasks, including machine translation, summarization, and question-answering. It is particularly useful in tasks where the model
needs to access a large amount of context or domain-specific knowledge to generate accurate and relevant outputs.
```