文本检索大作业

2000011714 王骥一 化学与分子工程学院

1.分词。读入数据集，对数据集中的文章进行去标点、分词、去除停用词及低频词，英文数据集考虑词性还原。

1. **数据处理（20'）。需完成**：读入数据集，对数据集中的文章进行去标点、分词、去除停用词及低频词，英文数据集可以考虑提取词根等。构建词典（下述检索词及相似词都只需在词典范围内完成），保存为*vocab.txt*。（该部分可直接调包）

```
English_f=pd.read_csv('all_news.csv')


stoplist = ['very', 'ourselves', 'am', 'doesn', 'through', 'me', 'against', 'up', 'just', 'her', 'ours',
            'couldn', 'because', 'is', 'isn', 'it', 'only', 'in', 'such', 'too', 'mustn', 'under', 'their',
            'if', 'to', 'my', 'himself', 'after', 'why', 'while', 'can', 'each', 'itself', 'his', 'all', 'once',
            'herself', 'more', 'our', 'they', 'hasn', 'on', 'ma', 'them', 'its', 'where', 'did', 'll', 'you',
            'didn', 'nor', 'as', 'now', 'before', 'those', 'yours', 'from', 'who', 'was', 'm', 'been', 'will',
            'into', 'same', 'how', 'some', 'of', 'out', 'with', 's', 'being', 't', 'mightn', 'she', 'again', 'be',
            'by', 'shan', 'have', 'yourselves', 'needn', 'and', 'are', 'o', 'these', 'further', 'most', 'yourself',
            'having', 'aren', 'here', 'he', 'were', 'but', 'this', 'myself', 'own', 'we', 'so', 'i', 'does', 'both',
            'when', 'between', 'd', 'had', 'the', 'y', 'has', 'down', 'off', 'than', 'haven', 'whom', 'wouldn',
            'should', 've', 'over', 'themselves', 'few', 'then', 'hadn', 'what', 'until', 'won', 'no', 'about',
            'any', 'that', 'for', 'shouldn', 'don', 'do', 'there', 'doing', 'an', 'or', 'ain', 'hers', 'wasn',
            'weren', 'above', 'a', 'at', 'your', 'theirs', 'below', 'other', 'not', 're', 'him', 'during', 'which']

```

```
#分词
lemmatizer=nltk.stem.WordNetLemmatizer()
def wordcount(news):
    freq=[]
    news=re.sub(r'\W+|\d+',' ',news)
    for word in news.split():
        if word not in stoplist:
            freq.append(lemmatizer.lemmatize(word))
    return ' '.join(freq)
English_f['word']=English_f['body'].apply(wordcount)
words=English_f['word'].str.split(' ',expand=True).stack().rename('words').reset_index()
words['level_0']=words['level_0']+1
words.columns=['id','level_1','words']
print(words)
```

这一部分首先利用正则表达式去除了除字母之外的所有字符（包括数字和标点符号）然后利用网上的停词表（后来了解到可以直接调用 nltk 包）去除停用词，使用 lemmatizer 进行词性的还原，分词后整理、重命名列得到每篇文章的分词结果

2.构建 TF-IDF 矩阵

```
#计算TF
def counttf(x):
    l=len(x)
    p=x['words'].value_counts()
    d={'words':p.index,
       'TF':p.values/l}
    p_2d=pd.DataFrame(data=d)

    r=pd.merge(x,p_2d,on='words')
    return r

TF_data=words.groupby('id').apply(counttf)
#统计完频率之后 去除重复词
TF_data.drop_duplicates(subset=['id','words'],inplace=True)

#计算IDF
doc_num=len(English_f)
def countidf(x):
    df_t=len(x['id'].value_counts())
    return np.log(doc_num/df_t)

IDF_data=words.groupby('words').apply(countidf).reset_index()
IDF_data.columns=['words','IDF']
tf_idf=pd.merge(TF_data,IDF_data)
tf_idf['TF-IDF']=tf_idf['TF']*tf_idf['IDF']
tf_idf.to_csv('tfidf.csv')
```

```
#转化为词向量
print(tf_idf['words'])
a=list(set(tf_idf['words']))
k=len(a)
word_list=dict(zip(a,list(range(k))))
wordlst=pd.Series(word_list)

def wordtovec(x):
    vec=np.zeros(k,)
    for i in range(x.shape[0]):
        idx=word_list[x.iloc[i]['words']]
        vec[idx]=x.iloc[i]['TF-IDF']
    return vec
newdata=tf_idf.groupby('id').apply(wordtovec)
print(newdata)

#降维
new_data=np.vstack(newdata.values)

model=PCA(n_components=300)
dev=model.fit_transform(new_data)
print(dev)
```

对每篇文章计算其 TF-IDF，并生成一个词典（这里生成了一个 key-value 对，以便之后进行查询），之后利用这个生成了 29060 维的词向量，并降到 300 维。

3.文章相似度计算以扩展搜索结果，词语相似度计算以扩展搜索词

```
#计算各个文章之间的相似度
for i in range(dev.shape[0]):
    dev[i]/=np.linalg.norm(dev[i])
cos_sim=dev@dev.T

print(cos_sim.shape)
cos_sim=cos_sim-np.eye(2225)
print(cos_sim)
```

```
#利用tf-idf矩阵，计算词之间的相似度
new_data=np.load('tf_idf.npy')
matrix=new_data.T@new_data

x=matrix.shape[0]
idxar=np.argsort(matrix,axis=1)[:,-4:-1]#取相似度排序前三个为模糊词
synonym=word_list
for word in word_list:
    idxlst=idxar[word_list[word]]
    lst=[]
    for k in idxlst:
        lst.append(a[k])#a是前面提到的词典，它和word_list互为补充，使查询时间缩短到O(1)
    synonym[word]=lst
```

4.降维效果评价

```
import numpy as np
from sklearn.cluster import KMeans
import pandas as pd

English_f=pd.read_csv('all_news.csv')
dev=np.load('dev.npy')
dict1=dict(zip(set(English_f['topic']),list(range(5))))
def replace(x):
    return dict1[x]
y=English_f['topic'].apply(replace)

```

```
oper=KMeans(n_clusters=5)
oper.fit(dev)
newX=oper.predict(dev)
df=pd.DataFrame({'origin_class':y,'new_class':newX})
def maxcount(x):
    return x.value_counts().iloc[0]
m1=df.groupby('new_class').apply(maxcount)
purity=sum(m1)/len(y)
print(purity)
```

```
0.8179775280898877
```

这里重新开了一个 notebook 进行评价，purity 为 0.82，还是比较高的，说明降到 300 维没有损失太多信息

5.服务器搭建

```
class LocalServer(object):
    def __init__(self, host, port):
        self.address = (host, port)
```

```
    def run(self):
        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        server.bind(self.address)
        i=0
        server.listen(5)
        while True:
            conn,add=server.accept()
            print(f'Connected to {add}')

            words=json.loads(conn.recv(1024).decode('utf-8'))#接收传参
            if 'break_000' in words:                          #此处认为break_000为结束符
                break

            print(words)
            p=Thread(target=self.lookupfor,args=(words,conn))#多线程实现并发处理
            p.start()
        p.join()
        server.close()
```

使用 socket 套接字搭建服务器端，接收到客户端的 words 列表后，将其和连接名称 conn 一同传入 lookupfor 函数进行搜索，lookupfor 函数负责向客户端返回搜索结果。这里使用 thread 进行多线程的并发处理（使用 Process 多进程处理失败了，不知道为什么，可能是拷贝连接 conn 的时候发生了占线问题）

在提交的作业中，多个 gui 是为了测试并发处理（但似乎并不能很好地测试）

当输入的 words 中包含 break000 时，断开连接，终止服务器运行

6.服务器端检索文本并返回结果

```
def lookupfor(self,words,conn):
    lemmatizer=nltk.stem.WordNetLemmatizer()
    for i in range(len(words)):
        words[i]=lemmatizer.lemmatize(words[i])#使用lemmatizer进行词性还原
        words+=synonym[words[i]]#将模糊词添加到列表中
    global topics
    global titles
    k=len(words)
    l=len(topics)
    returnlist=[]
    #首先检索主题和题目，这里认为二者重要性相同，如果又出现了一次，认为其重要性更高，排在前面
    for j in range(k):
        for i in range(l):
            if topics[i]==words[j]:
                if i in returnlist:
                    returnlist[returnlist.index[i]],returnlist[0]=returnlist[0],returnlist[returnlist.index[i]]
                else:
                    returnlist.append(i)
            if titles[i]==words[j]:
                if i in returnlist:
                    returnlist[returnlist.index[i]],returnlist[0]=returnlist[0],returnlist[returnlist.index[i]]
                else:
                    returnlist.append(i)
```

lookupfor 函数首先对传入的词进行词形还原和模糊词扩充，接着开始检索，将检索到的文章 id 放入 returnlist

首先检索主题和题目，将含有关键词的主题和题目放入 returnlist，如果 returnlist 中已经出现该文章，则将该文章放到最前面（一个粗糙的排序方法）

```
#其次检索与已有文章相似度最高的一篇文章，依次放入returnlist
for num in returnlist:
    idx=np.argmax(cos_sim[num])
    if idx in returnlist:
        returnlist[returnlist.index[idx]],returnlist[0]=returnlist[0],returnlist[returnlist.index[idx]]
    else:
        returnlist.append(idx)
#最后按关键字进行查询，按TF-IDF由高到低依次输出
for j in range(k):
    if words[j] in word_list:
        df=tf_idf[tf_idf['words']==words[j]]
        for i in df['id']:
            if i not in returnlist:
                returnlist.append(i-1)
articles=[]
for k in returnlist:
    articles.append(tuple(English_f.iloc[k,0:2]))

conn.send(json.dumps(articles).encode('utf-8'))#将结果转化为json字符串后返回客户端
return articles
```

接着将和已有文章相似度最高的文章放入 returnlist，与前相同，如果已经出现，则将该文章放到最前面

最后按关键字进行查询，按 TF-IDF 由高到低依次输出，不过这里不再改变原有文章的次序（最后一个重要性比较低）

接着将文章 id 转换为 article，打包发送给客户端