

题目 B4：基于关键词的文本排序检索系统

一、课题内容和要求

（一）课题内容

对于给定文本库（数据集：“题目 B4 的数据集.zip”解压后即可获得数据集，该数据集为纽约时报的报刊文章），用户提交检索关键词（例如：NBA, basket, ball），在文本库中查询与检索关键词最相关的 k 个文本（例如 k=5），并根据文本与检索关键词的相关度，对这 k 个文本进行排序，将排序后的结果返回给用户。

（二）课题要求

1. 基本要求

- （1）利用 TF-IDF 模型，为文本库中的文本创建索引（如倒排索引）；
- （2）用户输入的关键词可以是一个或多个；
- （3）对于返回的结果文本，需同时显示各检索关键词在结果文本中的出现频度信息；
- （4）系统内支持返回结果文本的查看；

2. 扩展要求

- （1）支持文本库的动态装载和处理；
- （2）支持停用词的管理和维护，停用词是指在没有检索价值的单词，如 is, am, are, a, an, the 等；
- （3）支持大规模文本库的高效排序检索（50 万级别及以上）。

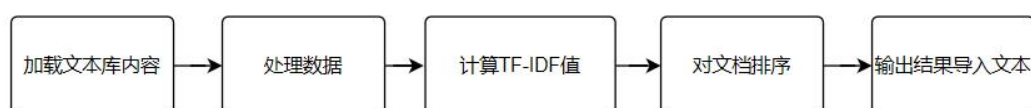
【其他要求】

- （1）变量、函数命名符合规范。
- （2）注释详细：每个变量都要求有注释说明用途；函数有注释说明功能，对参数、返回值也要以注释的形式说明用途；关键的语句段要求有注释解释。
- （3）程序的层次清晰，可读性强。
- （4）界面美观，交互方便。

二、课题需求分析

Tf-idf 是一个计算权值的算法，权值用于衡量关键词对于某篇文章的重要性（相关度），从而可以对指定关键词按照 tf-idf 值来对文本进行排序。起初为了方便，我准备使用 python 中的 nltk 库中的函数来完成分词和计算权值等操作，但是后来因为下载 nltk 库出现问题，于是我就自己完成了 tf-idf 的代码实现，因为还需要完成停用词的处理，于是调用了 nltk.corpus 中的 stopwords 库。

主思路：



涉及的函数：

函数名	功能
def loadDataSet(path)	读取文本库文本以字典形式输出
def dealDataSet(all_docu_dic)	处理文件库字典的数据
def computeTF(in_word, words_num_dic)	计算单词 in_word 在每篇文档的 TF
def computeIDF(in_word, words_num_dic)	计算 in_word 的 idf 值
def computeTFIDF(in_word, words_num_dic)	计算 in_word 在每篇文档的 tf-idf 值
def text_save(filename, data, word)	对检索词 word 的字典 data 输出到 filename 的文件中
def sortOut(dic)	对字典内容排序，并保留 value 值

三、课题相关数据结构及算法设计

Python 代码涉及的数据结构有基于线性表的 list，基于链表的字典等，涉及的算法有 TF-IDF 算法。

(1) 需要引入的 Python 模块

模块名称	导入目的
import math	用于计算数学中 log
import os	用于完成对文本库内容提取相关操作
import re	用于调用 split 完成自定义分词
from nltk.corpus import stopwords	调用 stopwords 库用于完成停用词的处理

(2) 主函数的实现

① 思路

首先将文本库的所有文本内容加载到程序中，并将加载的数据进行处理（包括分词、去除停用词等操作）得到一个记录各文档中各单词词数的字典，和一个文本库的总词库。文本库处理完毕后，用户输入一个或多个关键词，将用户的输入保存在一个 list 表中，然后对用户的输入进行分词处理，得到若干个关键词，如果其中存在关键词在文本词库中，那么就计算此关键词在各文本中的 tf-idf 值，并输出该关键词的按照 tf-idf 值降序排列的文本序列到 result1.txt 文件中；如果用户输入的所有关键词都不在词库中，就输出“无任何搜索结果”。一轮搜索结束后询问用户是否继续搜索，是则继续执行上述操作，并将输出保存在 result2.txt，...以此类推；否则退出程序。

② 代码

```
if __name__ == '__main__':
    # 载入文件
    print("\t默认文本库路径为: D:/study/B4/data")
    print("\t搜索结果文本路径为: D:/study/B4/result")
    path = "D:/study/B4/data" # 文本库路径
    all_docu_dic = loadDataSet(path) # 加载文本库数据到程序中
    words_set, words_num_dic = dealDataSet(all_docu_dic) # 处理数据返回值1. 文本词库(已去除停用词), 2. 各文本词数的词典
    n = 0 # 记录搜索次数
    a = -1 # 控制程序终止的变量
    while a != 0:
        in_words = input("搜索: ")
        input_list = re.split("[!? '.,)(+-. :]", in_words)
        k = 0 # 用于记录单次输入的有效关键词的个数
        n += 1
        for i in range(len(input_list)):
            if input_list[i] in words_set:
                k += 1
                tfidf_dic = computeTFIDF(input_list[i], words_num_dic) # 单词的tfidf未排序字典
                # 控制台输出
                print("关键词:" + input_list[i])
                print(sortOut(tfidf_dic)[0:5]) # 输出前五个相关文本
                # 文本输出
                text_save("result" + str(n) + ".txt", sortOut(tfidf_dic)[0:5], input_list[i]) # 将排序后的tfidf字典保存到文件中
        if k == 0:
            print("无任何搜索结果")
            a = input("任意键继续搜索, '0'退出:")
        print("-----")
```

(3) TF-IDF 模型的实现

1. 思路

刚开始实现 tf-idf 的时候是按照网上的实现代码加以调整, 但是实现成功后发现, 这个算法是计算文本库中所有单词的所有 tf-idf, 小规模文本库没什么影响, 但是实际情况是文本库规模很大, 用户没有输入的单词的数据多余, 会大大增加时间复杂度和空间复杂度, 所以经过思考我准备调整思路。最终的实现思路是: 用户输入的检索单词, 然后调用函数分别计算出此单词在各个文本的 tf 值, 此单词在文本库中 idf 值, 并计算出 tf-idf 值, 从而实现了高效检索。

2. 代码实现

涉及三个函数, 分别计算 tf 值, idf 值, 还有一个函数调用前两个函数计算 tf-idf 值。这三个函数都是相同的形式参数, in_word 是需要检索的单词, words_num_dic 是一个文本库中所有文档对应的单词词数字典, 格式如下: {txt1:{word1:num1,word2:num2},txt2:{word1:num3,word3:num4},...}。

① 计算 TF 的代码

计算 TF 值的步骤就是先计算各个文本的总次数, 然后计算检索单词在各个文档中的出现的词数, 再取两者的商, 就是该单词的 TF 值, 返回值是该检索词在各个文档的 TF 值的词典。

```

"""
计算单词in_word在每篇文档的TF

:param in_word: 单词
:param words_num_dic: 文本词数字典 {txt1:{word1:num1,word2:num2},...}
:return: tfDict: 单词in_word在所有文本中的tf值字典 {文件名1: tf1,文件名2: tf2,...}
"""

def computeTF(in_word, words_num_dic):
    allcount_dic = {} # 各文档的总词数
    tfDict = {} # in_word 的 tf 字典
    # 计算每篇文档总词数
    for filename, num in words_num_dic.items():
        count = 0
        for value in num.values():
            count += value
        allcount_dic[filename] = count
    # 计算 tf
    for filename, num in words_num_dic.items():
        if in_word in num.keys():
            tfDict[filename] = num[in_word] / allcount_dic[filename]
    return tfDict

```

② 计算 IDF 的代码

先计算出总文档数，再计算包含检索词的文档个数，对两者求商再取对数，对分母加 1 处理（目的是防止分母等于 0）并返回该结果。一个单词的 IDF 值只与整个文本库有关，换言之，一个单词在固定文本库中的 IDF 值固定，所以返回结果是个数。

```

"""
计算in_word的idf值

:param in_word: 单词
:param words_num_dic: 文本词数字典 {txt1:{word1:num1,word2:num2},...}
:return: 单词in_word在整个文本库中的idf值
"""

def computeIDF(in_word, words_num_dic):
    docu_count = len(words_num_dic) # 总文档数
    count = 0
    for num in words_num_dic.values():
        if in_word in num.keys():
            count += 1
    return math.log10((docu_count) / (count + 1))

```

③ 计算 TF-IDF 的代码

调用前两个函数，计算此单词在各个文档的 tf-idf 值，返回一个字典（同 tf）

```
"""
    计算in_word在每篇文档的tf-idf值

:param in_word: 单词
:param words_num_dic: 文本词数字典 {txt1:{word1:num1,word2:num2},...}
:return: tfidf_dic: 单词in_word在所有文本中的tf-idf值字典 {文件名1: tfidf1, 文件名2: tfidf2,...}
"""

def computeTFIDF(in_word, words_num_dic):
    tfidf_dic = {}
    idf = computeIDF(in_word, words_num_dic)
    tf_dic = computeTF(in_word, words_num_dic)
    for filename, tf in tf_dic.items():
        tfidf_dic[filename] = tf * idf
    return tfidf_dic
```

三、源程序代码

```
import math

import os

import re

from nltk.corpus import stopwords

def loadDataSet(path):
    """
    读取文本库中的文本内容以字典形式输出

    :param path: 文本库地址
    :return: 文本库字典 {文本名 1: 文本内容 1, 文本名 2: 文本内容 2...}
    """

    # 将文件夹内的文本全部导入程序

    files = os.listdir(path) # 得到文件夹下的所有文件名称

    all_docu_dic = {} # 接收文档名和文档内容的词典

    for file in files: # 遍历文件夹

        if not os.path.isdir(file): # 判断是否是文件夹，不是文件夹才打开
```

```

        f = open(path + "/" + file, encoding='UTF-8-sig') # 打开文件
        iter_f = iter(f) # 创建迭代器
        strr = ""
        for line in iter_f: # 遍历文件，一行行遍历，读取文本
            strr = strr + line

        all_docu_dic[file] = strr.strip('.') # 去除末尾的符号.

    print("文件库： ")
    print(all_docu_dic)
    return all_docu_dic

def dealDataSet(all_docu_dic):
    """
    处理文件库字典的数据

    :param all_docu_dic:文本库字典 {文本名 1: 文本内容 1, 文本名 2: 文本内容 2...}
    :return: 1.all_words_set 文本库的词库 {word1,word2,...}
            2.words_num_dic 文本词数字典 {txt1:{word1:num1,word2:num2},...}
    """
    all_words = []
    all_docu_cut = {} # 分完词后的 dic(dic 嵌套 list)

    stop_words = stopwords.words('english') # 原始停用词库
    ##停用词的扩展
    # print(len(stop_words))
    # extra_words = [' ' ]#新增的停用词
    # stop_words.extend(extra_words)#最后停用词
    # print(len(stop_words))

    # 计算所有文档总词库和分隔后的词库

```

```

for filename, content in all_docu_dic.items():

    cut = re.split("[!?' '),(+-=。 :]", content) # 分词

    new_cut = [w for w in cut if w not in stop_words if w] # 去除停用词，并且去除
split 后产生的空字符

    all_docu_cut[filename] = new_cut # 键为文本名，值为分词完成的 list

    all_words.extend(new_cut)

all_words_set = set(all_words) # 转化为集合形式


# 计算各文本中的词数

words_num_dic = {}

for filename, cut in all_docu_cut.items():

    words_num_dic[filename] = dict.fromkeys(all_docu_cut[filename], 0)

    for word in cut:

        words_num_dic[filename][word] += 1

# print("词库： ")

# print(all_words_set)

print("文件分词库： ")

print(all_docu_cut)

return all_words_set, words_num_dic # 返回词库和文档词数字典


def computeTF(in_word, words_num_dic):
    """
    计算单词 in_word 在每篇文档的 TF

    :param in_word: 单词
    :param words_num_dic: 文本词数字典 {txt1:{word1:num1,word2:num2},...}
    :return: tfDict: 单词 in_word 在所有文本中的 tf 值字典 {文件名 1: tf1,文件名 2: tf2,...}
    """

    allcount_dic = {} # 各文档的总词数

```



```

tfDict = {}      # in_word 的 tf 字典
# 计算每篇文档总词数
for filename, num in words_num_dic.items():
    count = 0
    for value in num.values():
        count += value
    allcount_dic[filename] = count
# 计算 tf
for filename, num in words_num_dic.items():
    if in_word in num.keys():
        tfDict[filename] = num[in_word] / allcount_dic[filename]
return tfDict

```

```

def computeIDF(in_word, words_num_dic):
    """
    计算 in_word 的 idf 值

    :param in_word: 单词
    :param words_num_dic: 文本词数字典 {txt1:{word1:num1,word2:num2},...}
    :return: 单词 in_word 在整个文本库中的 idf 值
    """
    docu_count = len(words_num_dic)    # 总文档数
    count = 0
    for num in words_num_dic.values():
        if in_word in num.keys():
            count += 1
    return math.log10((docu_count) / (count + 1))

```

```

def computeTFIDF(in_word, words_num_dic):

```

```
"""
```

计算 in_word 在每篇文档的 tf-idf 值

```
:param in_word: 单词
```

```
:param words_num_dic: 文本词数字典 {txt1:{word1:num1,word2:num2},...}
```

```
:return: tfidf_dic:单词 in_word 在所有文本中的 tf-idf 值字典 {文件名 1: tfidf1,文件名  
2: tfidf2,...}
```

```
"""
```

```
tfidf_dic = {}
```

```
idf = computeIDF(in_word, words_num_dic)
```

```
tf_dic = computeTF(in_word, words_num_dic)
```

```
for filename, tf in tf_dic.items():
```

```
    tfidf_dic[filename] = tf * idf
```

```
return tfidf_dic
```

```
def text_save(filename, data, word):
```

```
"""
```

对检索词 word 的字典输出到 filename 的文件中

```
:param filename:输出文本的文件名
```

```
:param data: 字典类型
```

```
:param word: 关键词
```

```
"""
```

```
fp = open("D:/study/B4/" + filename, 'a')
```

```
fp.write("关键词:" + str(word) + '\n')
```

```
for line in data:
```

```
    for a in line:
```

```
        s = str(a)
```

```
        fp.write('\t' + s)
```

```
        fp.write('\t')
```

```

        fp.write('\n')

    fp.close()

def sortOut(dic):
    """
    对字典内容按照 value 值排序，并保留 value 值

    :param dic: 字典
    :return: 嵌套元组的 list
    """
    return sorted(dic.items(), key=lambda item: item[1], reverse=True)

if __name__ == '__main__':
    # 载入文件

    print("\t 默认文本库路径为: D:/study/B4/data")
    print("\t 搜索结果文本路径为: D:/study/B4/result")

    path = "D:/study/B4/data"    # 文本库路径

    all_docu_dic = loadDataSet(path)  # 加载文本库数据到程序中

    words_set, words_num_dic = dealDataSet(all_docu_dic)    # 处理数据返回值 1.文本词
    库（已去除停用词），2.各文本词数的词典

    n = 0    # 记录搜索次数

    a = -1    # 控制程序终止的变量

    while (a != 0):
        in_words = input("搜索: ")

        input_list = re.split("[!?' .),(+-=。:]", in_words)

        k = 0    # 用于记录单次输入的有效关键词的个数

        n += 1

        for i in range(len(input_list)):
            if (input_list[i] in words_set):

```

```

        k += 1

        tfidf_dic = computeTFIDF(input_list[i], words_num_dic) # 单词的 tfidf
未排序字典

        # 控制台输出

        print("关键词:" + input_list[i])

        print(sortOut(tfidf_dic)[0:5]) # 输出前五个相关文本

        # 文本输出

        text_save("result" + str(n) + ".txt", sortOut(tfidf_dic)[0:5], input_list[i]) #
将排序后的 tfidf 字典保存到文件中

    if (k == 0):

        print("无任何搜索结果")

    a = input("任意键继续搜索, '0'退出:")

    print("-----")

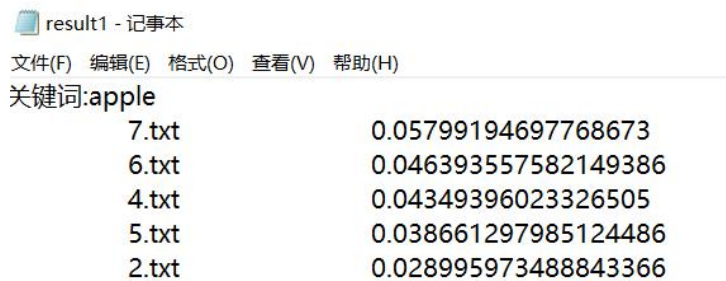
```

四、测试数据及其结果分析

文本数据：

'1.txt':	'water'
'2.txt':	'English apple'
'3.txt':	'maths apple'
'4.txt':	'apple apple apple maths'
'5.txt':	'apple apple maths'
'6.txt':	'apple apple apple apple water'
'7.txt':	'apple'
'8.txt':	'app'

输入：apple:



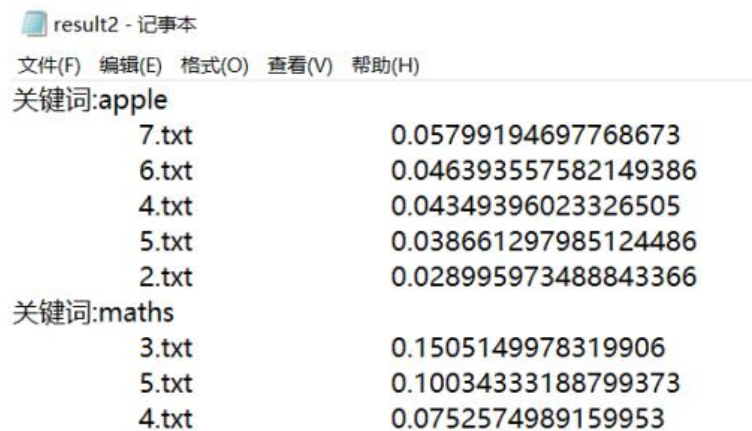
result1 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

关键词:apple

7.txt	0.05799194697768673
6.txt	0.046393557582149386
4.txt	0.04349396023326505
5.txt	0.038661297985124486
2.txt	0.028995973488843366

输入：apple maths



result2 - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

关键词:apple

7.txt	0.05799194697768673
6.txt	0.046393557582149386
4.txt	0.04349396023326505
5.txt	0.038661297985124486
2.txt	0.028995973488843366

关键词:maths

3.txt	0.1505149978319906
5.txt	0.10034333188799373
4.txt	0.0752574989159953

分析：

程序导入文本库内容后，用户输入一个或多个关键词，程序都能够检索出相关性排名前五的文本，并且能够同时显示关键词的频度信息，基本可以满足此题的要求。

六、课题完成过程中遇到的问题及解决方法

(1) 分词后出现空字符

在对文本数据进行分词的时候遇到了空字符的情况，于是为了方便我就去博客上面寻找解决方案，最后通过 `if w` 语句去除了空字符

(2) 分词后文本末尾的标点依然保留


在分词完成后发现，句子末尾的句号也被作为一个字母划分到一个单词里面了，于是就在加载文本库的时候调用 `strip` 函数去除了句子末尾的标点

(3) 调用 `str()` 函数时，出现对象无法调用的问题

在将文本的排序结果输出到 `txt` 文本中时，必须要将内容转化为字符形式，也会用到系统自带的 `str()` 函数，但是调用 `str` 函数的时候控制台报出：`TypeError: 'str' object is not callable` 的异常，为了了解 `str` 不能被调用的原因，结果了解得知：函数 `str()` 是系统自带的，你不能在用它的时候自己同时定义一个别的叫做 `str` 的变量，这样会冲突。原来我之前定义过一个 `str` 的变量，所以我就修改了之前定义的 `str` 变量为 `strr`。

(4) 编码出现异常：单词前多出 `\ufeff` 前缀

经过查阅后得到了解决方案，在打开文件操作中增加参数 `encoding='UTF-8-sig'`



```
'\ufeffEnglish'
```

七、总结

这个课题让我知道 `tf-idf` 算法的作用以及掌握了代码实现，让我了解了倒排索引的概念及作用，加强了 `python` 的代码能力，知道了 `python` 里面有许多封装好的库可供调用，锻炼了自己的逻辑思维能力和解决问题动手实践的能力。总之，经过这次算法与数据结构这门实践课程，我学到了很多，将书本上面的知识运用到实际项目中是理解掌握知识的好方法。