

# HW3 向量空间模型(Vector Space Model)

姓名：管昀玫

学号：2013750

专业：计算机科学与技术

## 作业要求

给定查询文档集合（诗词txt文件），完成向量空间模型并对文档集合实现查询功能。

## 算法思想

### 相似性

相似性，可以用距离来衡量。而在数学上，可使用余弦来计算两个向量的距离。

$$\cos(\vec{a}, \vec{b}) = \frac{\vec{a} * \vec{b}}{||\vec{a}|| * ||\vec{b}||}$$

因此，用向量来表示文档，然后就可以用余弦来计算两篇文章之间的相似度了。

### 词袋模型

一篇文档里面有很多很多句子，每个句子又是由一个个的词组成。词袋模型，通俗地讲，就是：把一篇文档看成**词袋**，里面装着一个一个的词。从而将一篇文档转化成了一个一个的词(或者称之为 term)，显然地，文档转化成一个个的词之后，词与词之间的顺序关系丢失了。

### TF-IDF

- 有一个文档集合C，文档集合C里面一共有N篇文档
- 有一个词典D，或者叫词库（Vocabulary），词库里面一共有M个词

### 文档到向量的转化

向量是有长度的，向量中的每个元素是数值。

首先将文档通过词袋模型转化成一个个的词，一般地，由于文档中的词都会存在于词典D中，定义一个M维向量（M等于词典大小），若文档中的某个词在词典中出现了，就给这个词赋一个实数值。若未出现，则在相应位置处赋值为0。

### 使用TF-IDF来衡量每个词的权重

#### TF值

tf(term frequency)，是指 term 在某篇文档中出现的频率。tf 是针对单篇文档而言的，即：**某个词在这篇文档中出现了多少次**。词频是计算文档得分的一个因子，因此为了计算某篇文档的得分，使用的词频指的就是term在这篇文档中出现的次数，而不是term在所有文档中出现的次数。

$$TF = \log_{10}(N + 1)$$

## IDF值

idf值 由 词(term) 出现在各个文档中数目来决定。idf 值是针对所有文档(文档集合)而言的，即：**数一数这个词都出现在哪些文档中。**

$$idf_t = \log \frac{N}{df_t}$$

## TF-IDF值

$$TF-IDF = tf * idf$$

前面提到，将文档向量化，需要给文档中的每个词赋一个实数值，这个实数值，就是通过tf\*idf得到。










由此，文档和查询都能形成向量，计算他们之间的余弦相似度就可得知他们的相似性。相似度越高则说明越相关，返回的结果也更靠前。


## 算法实现

下面讲按照代码的思路顺序进行说明

### 数据集

在 dataset 文件夹下，一共有9个txt文档，内容为作者+诗句

 A Drinking Song.txt	2022/10/25 18:56	文本文档	1 KB
 Aedh wishes for the Cloths of Heave...	2022/10/25 18:58	文本文档	1 KB
 Down by the Salley Gardens.txt	2022/10/25 18:57	文本文档	1 KB
 Freedom.txt	2022/10/25 19:05	文本文档	1 KB
 Leave This.txt	2022/10/25 19:04	文本文档	1 KB
 Tonight I Can Write.txt	2022/10/25 18:46	文本文档	1 KB
 Walk with Me in Moonlight.txt	2022/10/25 18:48	文本文档	1 KB
 When you are old.txt	2022/10/25 18:51	文本文档	1 KB
 Where the Mind is Without Fear.txt	2022/10/25 19:03	文本文档	1 KB

 A Drinking Song.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

Author: William Butler Yeats  
Wine comes in at the mouth  
And love comes in at the eye;  
That's all we shall know for truth  
Before we grow old and die.  
I lift the glass to my mouth,  
I look at you, and I sigh.

### 1. 输入关键词并选择要查询的域

```
def select():  
    selected_region = []  
    for i in range(len(region)):  
        if input(f'是否选择{region[i]}? (Y/n): ') in ['y', 'Y', '']:  
            selected_region.append(region[i])  
    return selected_region
```

其中有三个域，为 ("author", "name", "content")，即（作者、诗名、诗句）。可以自定义组合域的查询，如只选作者和诗句，或只选择诗句。

除此之外，还需要输入关键词，并转换为小写。

```
input_word = input('请输入关键词: ')
# 将输入的关键词全部字母转化为小写
input_word = input_word.lower()
```

## 2. 文档集预处理

首先读取所有文件，获取每个文档的作者、诗名和诗句。去除所有的标点，并用" "代替。

建立一个dataframe，索引为第i个文档，有三个属性： "name", "author", "content"

```
# 预处理文件
## 新建一个包含name author content 三列的dataframe
df = pd.DataFrame(columns=["name", "author", "content"])
## 遍历文件
path = './dataset'
for file in os.listdir(path):
    with open(path + '/' + file, 'r') as f:
        lines = f.readlines()
        content = ''
        for line in lines:
            if 'Author: ' in line:
                author = line.split('Author: ')[1].replace('\n', '')
            elif line != '\n':
                content += line.replace(',', ' ').replace('; ', ' ').replace('.', ' '),
                '\n').replace('?', ' ').replace('!', ' ').replace('\n', ' ').replace('\n', ' ') # 删除, ; . ? ! 等标点符号
            content += ' ' # 换行的时候加一个空格
        # 将name author content 加入dataframe
        df = pd.concat([df, pd.DataFrame([[file.replace('.txt', ''), author, content]], columns=["name", "author", "content"])], ignore_index=True)
```

将选中的域的内容加入一个dataframe中，并将每一个文档所选中域的内容拼接为一个字符串，用" "隔开。

```
# 遍历selected_region，将符合条件的列加入df_selected
df_selected = pd.DataFrame()
for region in selected_region:
    df_selected = pd.concat([df_selected, df[region]], axis=1)

# 将df_selected中的每一行拼接成一个字符串。索引为第i个文档，内容为选中的域中的内容
df_selected_dict = {}
for i in range(len(df_selected)):
    df_selected_dict[i] = ''
    for region in selected_region:
        df_selected_dict[i] += df_selected[region][i] + ' '
```

利用python中set()的无序不重复属性存储在所有文档中出现的所有词项。用一个dict来存储每一个文档的分词结果。

对词项进行排列，并统一成小写。

```

split_dict = {} # 存储每一行的分词结果，包含所有选定的词，索引为第i个文档，内容为一个list(文档内所有词)
word_set = set() # 新建用于存储的set(set无序不重复)
for k, v in df_selected_dict.items():
    split_result = v.split(' ')
    # 遍历split_result，全部转化为小写
    for i in range(len(split_result)):
        split_result[i] = split_result[i].lower()
    split_dict[k] = split_result
    word_set = word_set.union(split_result) # 求并集
# 将分词集合升序排序
word_set = sorted(word_set)

```

### 3. 计算TF

计算**每个词项**在**不同文档**中的词频tf(Term Frequency)。

这里为了减少文本长度带来的影响，使用log来减小词频的影响。 $TF = \log_{10}(N + 1)$ ，其中N表示词项在对应文档中出现的次数。

```

# 统计词项tj在文档Di中出现的次数，也就是词频。
def computeTF(word_set, split):
    tf = dict.fromkeys(word_set, 0)
    for word in split:
        tf[word] += 1
    for word, cnt in tf.items():
        tf[word] = math.log10(cnt + 1) # TF = log10(N + 1) 减少文本长度带来的影响
    return tf

tf_dict = {} # 存储tf(词频)。索引为第i个文档，内容又是一个dict，索引为词，内容为df
for k, v in split_dict.items():
    tf_dict[k] = computeTF(word_set, v)

```

### 4. 计算IDF

计算每个词项在整个文档集合的逆向文件频率idf(Inverse Document Frequency)

$idf_t = \log \frac{N}{df_t}$  其中N表示文档总数， $df_t$ 表示包含词项的文档数。

```

idfs = computeIDF(list(tf_dict.values()))
tfidf_dict = {} # 存储每一篇文档的向量(tf-idf)
for k, v in tf_dict.items():
    tfidf_dict[k] = computeTFIDF(v, idfs)
tfidf_list = list(tfidf_dict.values()) # 将结果转化为list，方便后续调用

# 计算tf-idf(term frequency-inverse document frequency)
def computeTFIDF(tf, idfs): # tf词频,idf逆文档频率
    tfidf = {}
    for word, tfval in tf.items():
        tfidf[word] = tfval * idfs[word]
    return tfidf

```

## 5. 筛选关键词

为了计算效率，选出tf-idf最大的前 `key_valid_number` 个词。根据此次文档集的长度，`key_valid_number` 设置为100。

```
key_tfidf_dict = {} # 存储关键词的tfidf。筛选出tf-idf最大的前100个词，降序排列
for k, v in tfidf_dict.items():
    key_tfidf_dict[k] = sorted(tfidf_dict[k].items(), key=lambda d: d[1],
reverse=True)[:key_valid_number] # d.items() 以列表的形式返回可遍历的元组数组
key_tfidf_list = list(key_tfidf_dict.values()) # 将结果转化为list，方便后续调用
```

同理，计算查询的df和tf-idf

```
tf_input = computeTF(word_set, split_input) # 查询的tf
tfidf_input = computeTFIDF(tf_input, idfs) # 查询的tf-idf
key_input = sorted(tfidf_input.items(), key=lambda d: d[1], reverse=True)
[:key_valid_number] # 查询的前100个关键词
len_key_input = length(key_input)
```

## 6. 输出向量空间

为了简洁，去掉无关单词

```
df_result = pd.DataFrame([*tfidf_list, tfidf_input]) # 将每个文档和查询的向量
合成一张表
print(df_result)
i = 0
while i < len(df_result.columns):
    if any(df_result.values.T[i]) == 0:
        df_result = df_result.drop(columns=df_result.columns[i], axis=1)
# 去掉向量全为0的列
    else:
        i = i + 1
```

## 7. 计算余弦相似度并排名

`length`函数是为了计算向量的长度，采用欧氏距离。

```
def length(key_list):
    num = 0
    for i in range(len(key_list)):
        num = num + key_list[i][1] ** 2
    return round(math.sqrt(num), 2)
```

```

# 计算余弦相似度并排序
result = []
for i in range(len(key_tfidf_list)): # 遍历每个文档
    num = 0
    for j in range(len(key_input)): # 遍历每个关键输入词
        for k in range(len(key_tfidf_list[i])): # 遍历每个文档内的每个关键词
            if key_input[j][0] == key_tfidf_list[i][k][0]: # 若为相同单词
                num = num + key_input[j][1] * key_tfidf_list[i][k][1]
    result.append((i, (round(num / (len_key_input *
length(key_tfidf_list[i])), 4)))) # 存储第i个文档的余弦相似度
result = sorted(result, key=lambda d: d[1], reverse=True)

```

## 测试

### 输入

输入要查询的单词，用空格隔开；并选择要查询的域

```

请输入关键词: william had a drinking
检索关键词为: william had a drinking
是否选择author? (Y/n): Y
是否选择name? (Y/n): Y
是否选择content? (Y/n): Y

```

### 输出

输出余弦相似度与文档排序

其中，余弦相似度的格式为：(i-th, 余弦相似度)

文档排序的格式为 <文档名> <余弦相似度>

```

a adventurous aedh ... you young your yourself
0 0.0 0.106023 0.000000 0.000000 ... 0.106023 0.000000 0.000000 0.000000
1 0.0 0.000000 0.000000 0.287243 ... 0.106023 0.000000 0.227635 0.000000
2 0.0 0.106023 0.000000 0.000000 ... 0.000000 0.455269 0.000000 0.000000
3 0.0 0.106023 0.287243 0.000000 ... 0.168042 0.000000 0.287243 0.287243
4 0.0 0.106023 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000
5 0.0 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000
6 0.0 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000
7 0.0 0.000000 0.000000 0.000000 ... 0.212046 0.000000 0.333479 0.000000
8 0.0 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000
9 0.0 0.106023 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000

[10 rows x 296 columns]
*****向量空间*****
a adventurous aedh again ... you young your yourself
0 0.106023 0.000000 0.000000 0.000000 ... 0.106023 0.000000 0.000000 0.000000
1 0.000000 0.000000 0.287243 0.000000 ... 0.106023 0.000000 0.227635 0.000000
2 0.106023 0.000000 0.000000 0.000000 ... 0.000000 0.455269 0.000000 0.000000
3 0.106023 0.287243 0.000000 0.000000 ... 0.168042 0.000000 0.287243 0.287243
4 0.106023 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000
5 0.000000 0.000000 0.000000 0.455269 ... 0.000000 0.000000 0.000000 0.000000
6 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000
7 0.000000 0.000000 0.000000 0.000000 ... 0.212046 0.000000 0.333479 0.000000
8 0.000000 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000
9 0.106023 0.000000 0.000000 0.000000 ... 0.000000 0.000000 0.000000 0.000000

[10 rows x 294 columns]
*****余弦相似度*****
[(0, 0.1854), (1, 0.0714), (7, 0.0714), (2, 0.0268), (4, 0.0173), (3, 0.0139), (5, 0.0), (6, 0.0),
(8, 0.0)]
*****文档排序*****
A Drinking Song 0.1854
Aedh wishes for the Cloths of Heaven 0.0714
When you are old 0.0714
Down by the Salley Gardens 0.0268
Leave This 0.0173
Freedom 0.0139
Tonight I Can Write 0.0
Walk with Me in Moonlight 0.0
Where the Mind is Without Fear 0.0
*****

```

## 实验感悟

这次实验主要的难度还是在于要理解向量空间模型各个步骤的意义，只要能够弄清每个步骤在做什么、公式是什么，写起代码就会容易很多。对我来说最难的地方在于是python的dict结构较为复杂，我又嵌套了好几层，有时候dict需要转换为list，后续才方便处理。

通过这次实验，我理解并掌握了向量空间模型各个步骤的含义，更深入地了解了信息检索，更加熟练了python编程，收获了一些python编程小技巧，收获颇多。