03_作业_实验报告

姓名: 孟笑朵
学号: 2010349

实验概述

实验要求:

实现对给定文本的自定义域中文本查询。

实验步骤:

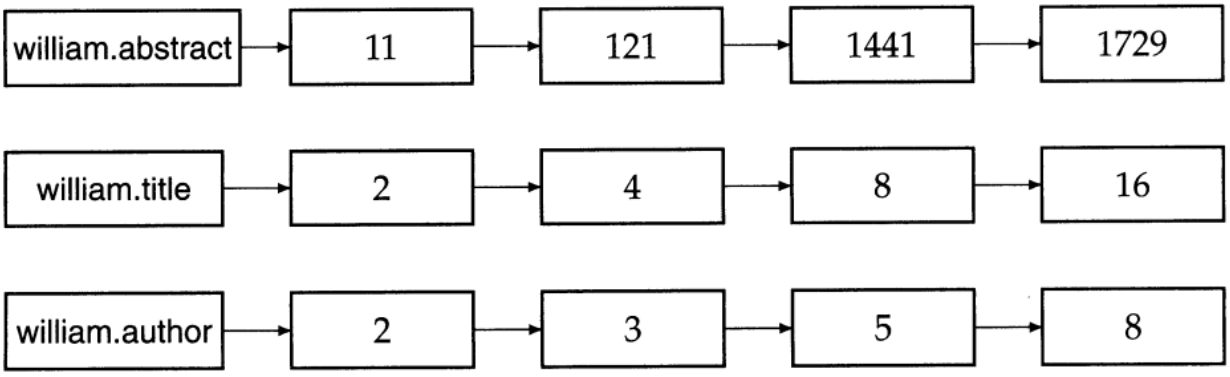
1. 首先对词项进行排序, 需要对词项进行一些处理: 包括单词变小写 变原型的处理;
2. 计算每个词项在不同文章中的词频, 计算每个词项在整个文档集合中的idf;
3. 将查询语句的向量分别与每个文档的向量计算余弦相似度, 返回对应的结果的排序;

实验过程

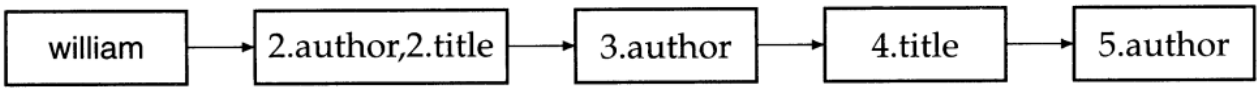
观察给定文本，实现自定义域中文本的查询包括以下三个文本域:

1. 标题 2. 作者 3. 文本

给定文本域中索引的构建方式有两种:
方法一：对于每一个文本域都构建一个索引结构，如下图所示:



方法二：将域的信息记录到倒排索引表中而非词典中，如下图所示:



在这里，我们采用方法二的模式进行构建包含域的倒排索引，同时方便后边的域加权评分。

文本预处理

首先本程序对给定文本进行了预处理,预处理主要有以下三个方面:

1. 大小写处理预处理
2. 标点空格预处理
3. 词形还原预处理

注:由于诗歌的特殊性,我们这里不做对应的去除停用词的预处理操作

在进行词形还原预处理时,本程序首先进行了单词的词性获取,然后根据单词词性进行预处理,如下所示:

PYTHON

```python
from nltk import word_tokenize, pos_tag
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer

# 获取单词的词性
def get_wordnet_pos(tag):
    if tag.startswith('J'):
        return wordnet.ADJ
    elif tag.startswith('V'):
        return wordnet.VERB
    elif tag.startswith('N'):
        return wordnet.NOUN
    elif tag.startswith('R'):
        return wordnet.ADV
    else:
        return None

for doc in os.listdir(input_path):
    # 大小写处理
    name = doc[:-4].lower()
    # 存放位置处理
    new_doc_path = os.path.join(output_path, name)
    # 存储文本
    if os.path.exists(output_path):
        pass
    else:
```

```
            os.mkdir(output_path)
        shutil.copyfile(os.path.join(input_path,doc), new_doc_path)
        author_flag = True
        lines = []
        for line in open(new_doc_path).readlines():
            # 删除文本中的标点
            line = line.translate(str.maketrans("","", string.punctuation))
            # 删除文本前后空格
            line = line.strip()
            # 对作者域进行处理
            if author_flag:
                line = line.lower()[7:]
                author_flag = False
            else:
                # 大小写处理
                line = line.lower()
                # 对词形进行还原
                words = line.split(' ')
                # print(words)
                word_list = []
                tagged_sent = pos_tag(words)     # 获取单词词性
                wnl = WordNetLemmatizer()
                for tag in tagged_sent:
                    wordnet_pos = get_wordnet_pos(tag[1]) or wordnet.NOUN
                    word_list.append(wnl.lemmatize(tag[0], pos=wordnet_pos))
# 词形还原
                line = ' '.join(word_list)
                # print(line)
            lines.append(line)
```

在这本程序将处理后的文件放在了`src/newdata`文件夹下,方便查看对应的预处理文本.

同时,我们对相应的查询文本也进行同样的预处理工作,如下所示:

```
# 先进行一个词干还原
query = query.lower().strip().translate(str.maketrans("","",
string.punctuation))
words = query.split(' ')
# print(words)
word_list = []
tagged_sent = pos_tag(words)     # 获取单词词性
```

```
wnl = WordNetLemmatizer()
for tag in tagged_sent:
    wordnet_pos = get_wordnet_pos(tag[1]) or wordnet.NOUN
    word_list.append(wnl.lemmatize(tag[0], pos=wordnet_pos)) # 词形还原
query = ' '.join(word_list)
```

包含域的倒排索引索引的构建

首先,本程序仿照之前Lab1中的词典形式构建了词典，如下所示:

PYTHON

```
class IdMap:
    def __init__(self):
        self.str_to_id = {}
        self.id_to_str = []

    def __len__(self):
        """Return number of terms stored in the IdMap"""
        return len(self.id_to_str)

    def _get_str(self, i):
        """Returns the string corresponding to a given id (`i`)."""
        if i >= self.__len__():
            raise IndexError
        else:
            return self.id_to_str[i]

    def _get_id(self, s):
        """Returns the id corresponding to a string (`s`).
        If `s` is not in the IdMap yet, then assigns a new id and returns
the new id.
        """
        if s in self.str_to_id:
            return self.str_to_id[s]
        else:
            self.id_to_str.append(s)
            self.str_to_id[s]=len(self.id_to_str)-1
            return self.str_to_id[s]

    def __getitem__(self, key):
        """If `key` is a integer, use _get_str;
           If `key` is a string, use _get_id;"""
```

```
        if type(key) is int:
            return self._get_str(key)
        elif type(key) is str:
            return self._get_id(key)
        else:
            raise TypeError
```

倒排索引构建的过程中，本程序将对应的文本域和相应词项出现的次数添加到了对应词项-文本对的文本id中，如下所示：

[
(0, ['0_head_1', '2_body_3', '3_body_2', '4_body_1', '6_body_1']),
(1, ['0_head_1']),
(2, ['0_head_1']),
(3, ['0_author_1', '1_author_1', '2_author_1', '7_author_1']),
...
]

> 注：这里的`(0, ['0_head_1', '2_body_3', '3_body_2', '4_body_1', '6_body_1'])`代表**id为0的term在id为0的文档的标题域出现了1次，在id为2的文档的内容域出现了3次……以此类推，对应的`0_author_1`代表在id为0的文档的作者域出现了1次.**

如下是构建包含域的倒排索引的核心代码：

PYTHON

```
class RegionIndex:
    head = 1
    author = 2
    body = 4
    def __init__(self, data_dir):
        self.term_id_map = IdMap()
        self.doc_id_map = IdMap()
        self.data_dir = data_dir
        self.index = []
        doc_region_length = []
        # self.output_dir = output_dir

    def parse_pairs(self):
        '''
        将对应的文本域中的单词文本转化为对应pair对
        '''
```

```
        td_pairs = []
        for doc in os.listdir(self.data_dir):
            new_path = os.path.join(self.data_dir,doc)
            doc_id = self.doc_id_map[new_path]
            for term in doc.split(' '):

td_pairs.append((self.term_id_map[term],str(doc_id)+"_head"))
            first_flag = True
            for line in open(new_path).readlines():
                if first_flag:
                    for term in line.strip().split(' '):
                        td_pairs.append((self.term_id_map[term],
str(doc_id)+"_author"))
                    first_flag = False
                else:
                    for term in line.strip().split(' '):
                        td_pairs.append((self.term_id_map[term],
str(doc_id)+"_body"))
        return td_pairs

    def parse_index(self,td_pairs):
        '''
        将pair对中的一些单词和对应的文本提取出来变为倒排索引
        对应的倒排索引格式:
        [(0, ['0_head_1', '2_body_3', '3_body_2', '4_body_1',
'6_body_1']),
         (1, ['0_head_1']),
         (2, ['0_head_1']),
         (3, ['0_author_1', '1_author_1', '2_author_1', '7_author_1']),
         ...
        ]
        '''
        self.index = []
        term_id = -1
        doc = ""
        num = 0
        posting_list = []
        for pair in sorted(td_pairs):
            # 按对应的term进行建立索引
            # print(pair)
```

```
            if pair[0] != term_id:
                if term_id != -1:
                    posting_list.append(doc+"_"+str(num))
                    self.index.append((term_id, posting_list))
                    doc = ""
                    num = 0
                term_id = pair[0]
                posting_list = []
            if pair[1] != doc:
                if doc != "":
                    posting_list.append(doc+"_"+str(num))
                    num = 0
                doc = pair[1]
            num += 1
        posting_list.append(doc+"_"+str(num))
        self.index.append((term_id, posting_list))
        return self.index
```

基于三维向量空间模型的查询的实现

本程序支持`诗歌名，诗人，诗歌内容`三个文本域自由组合查询, 如下所示:

```
1.诗歌名 2.诗人 3.诗歌内容 4.诗歌名和诗人 5.诗歌名和诗歌内容 6.诗人和诗歌内容
7.诗歌名和诗人和诗歌内容 8.全文本检索 9.退出
```

包含域的查询和对应的全文本查询实现方式一致, 我们先来看对应的全文本查询的实现.

全文本查询

如下为三维向量空间模型, 文本相似度的计算公式:

$$sim(\vec{d}, \vec{q}) = \sum_{t} \frac{\text{tf-idf}_{td} \times \text{tf-idf}_{tq}}{\|\vec{q}\| \times \|\vec{d}\|}$$

它有不同的组合变体, 在这里我们采用的是`lnc.ltn`的文档向量查询模式, 如下所示:

Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1+\log(\mathrm{tf}_{t,d})$	t (idf)	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented)	$0.5+\frac{0.5\times \mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases}1 & \text{if } \mathrm{tf}_{t,d}>0\\ 0 & \text{otherwise}\end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha<1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

在进行三维向量空间的查询之前，我们需要将对应的词项的`IDF`提前计算得到，**在这里我们还额外将文本的向量长度计算保存，方便之后实现快速查询**

IDF计算的代码实现：

PYTHON

```python
def cal_idf(self):
    '''
    计算idf的接口 计算idf的方式：出现该词项的所有文档的数目df idf = log(N/df)
    '''
    doc_num = len(os.listdir(self.data_dir))
    term_doc_num = []
    for index in self.index:
        doc_list = []
        for doc in index[1]:
            doc_list.append(doc.split("_")[0])
        term_doc_num.append(len(set(doc_list)))
        # print(term_doc_num)
    for i in range(len(term_doc_num)):
        term_doc_num[i] = round(math.log10(doc_num/term_doc_num[i]),4)
    return term_doc_num
```

文本向量长度计算的代码实现：

PYTHON

```python
def cal_doc_length(self):
    '''
    计算向量的长度
    '''
    doc_num = len(os.listdir(self.data_dir))
    term_doc_num = self.cal_idf()
    # 初始化列表
    self.doc_length = [0]*doc_num
    # !不能这样子初始化 浅拷贝
```

```
    #all_doc_vec = [[0] * len(self.index)] * doc_num
    all_doc_vec = [[0 for i in range(len(self.index))] for j in
range(doc_num)]
    # print(all_doc_vec)
    for index in self.index:
        term_id = index[0]
        postings = index[1]
        # print(postings)
        for post in postings:
            post_id = post.split('_')[0]
            post_num = post.split('_')[2]
            all_doc_vec[int(post_id)][int(term_id)] += int(post_num)
    for doc_id in range(len(all_doc_vec)):
        for term_id in range(len(doc)):
            all_doc_vec[doc_id][term_id] *= term_doc_num[term_id]
        for i in range(len(all_doc_vec[doc_id])):
            self.doc_length[doc_id] += math.pow(all_doc_vec[doc_id][i],
2)
        self.doc_length[doc_id] =
round(math.sqrt(self.doc_length[doc_id]),4)
    return self.doc_length
```

对应的查询代码的实现如下所示:

PYTHON

```
def all_retrieve(self, query):
    '''
    计算tf的方式：词项在文档中出现的次数，tf在原始的值的基础上进行计算 减少文档
长度的影响tf = log10(N+1)
    计算余弦相似度，将对应文本向量中的每一项用tf * idf来进行表示，
    具体的计算方法：用对应的在query中出现的term的query的tf*idf * 对应的doc的
term的tf*idf 再除以对应doc的向量的长度
    '''
    # 先进行一个词干还原
    query = query.lower().strip().translate(str.maketrans("","",
string.punctuation))
    words = query.split(' ')
    # print(words)
    word_list = []
    tagged_sent = pos_tag(words)     # 获取单词词性
    wnl = WordNetLemmatizer()
```

```
    for tag in tagged_sent:
        wordnet_pos = get_wordnet_pos(tag[1]) or wordnet.NOUN
        word_list.append(wnl.lemmatize(tag[0], pos=wordnet_pos)) # 词形还
原
    query = ' '.join(word_list)
    # 对应的idf
    term_doc_num = self.cal_idf()
    self.cal_doc_length()
    # 进行计算获得query的向量
    query_word = query.split(" ")
    query_vec = []
    for i in range(len(set(query_word))):
        query_vec.append(0)
        for j in range(len(query_word)):
            if query_word[j] == list(set(query_word))[i]:
                query_vec[i] += 1
    for i in range(len(query_vec)):
        query_vec[i] = round(math.log10(query_vec[i]+1),4)
    # print(query_vec)
    # 下面进行文档的向量构建
    all_doc_vec = []
    doc_num = len(os.listdir(self.data_dir))
    for i in range(doc_num):
        doc_vec = []
        for j in range(len(set(query_word))):
            # 找到对应的词项id
            word_id = self.term_id_map[list(set(query_word))[j]]
            # print(list(set(query_word))[j])
            doc_vec.append(0)
            if word_id < len(self.index):
                posting_list = self.index[word_id]
                # print(posting_list)
                for post in posting_list[1]:
                    if post.split('_')[0] == str(i):
                        doc_vec[j] += int(post.split('_')[2])
                doc_vec[j] =
round(math.log10(doc_vec[j]+1),4)*term_doc_num[word_id]
        all_doc_vec.append(doc_vec)
    # print(all_doc_vec)
    all_doc_vec.append(query_vec)
```

```
    ans = pd.DataFrame(all_doc_vec)
    query_doc_sim = []
    for i in range(len(all_doc_vec)-1):

query_doc_sim.append((ans.loc[i,:]*ans.loc[9,:]).sum()/self.doc_length[i]
)
        # print("Query和文档D"+str(i+1)+"的相似度SC(Q, D"+str(i+1)+") :",
(ans.loc[i,:]*ans.loc[9,:]).sum())
    sorted_id = sorted(range(len(query_doc_sim)), key=lambda k:
query_doc_sim[k], reverse=True)
    for i in range(len(sorted_id)):
        print("排名第"+str(i+1)+"位的是文档D"+str(sorted_id[i]+1)+", 与
Query余弦相似度SC(Q, D"+str(sorted_id[i]+1)+")
:",query_doc_sim[sorted_id[i]])
```

针对域进行基于三维向量空间模型的查询

对于不同的域的查询, 本程序采用的策略是: **对于文档的不同域进行Query和文本域的向量相似度计算, 然后将对应的余弦相似度按照一定的比例进行相加得到最终文本与查询的相似度排序**

> 注: 这里的比例采用的是一般的求平均的方式

要解决多文本域的查询, 需要解决单一文本域的查询, 然后将单一文本域查询结果按照对应的比例相加即得到对应的结果, 如下所示:

PYTHON

```
def region_retrieve(self, query_list, region):
    '''
    支持域查询，根据出现查询中单词的文本所在域上的三维空间模型的计算结果来进行返
回结果排序
    不同域的查询思路是：在不同域中进行计算相似度，然后将相关性按0.5 0.5比例进行
累加最后排序
    '''
    region_list = []
    query_doc_sim = [0]*len(os.listdir(data_path))
    if region == 1:
        region_list = ["head"]
        query = query_list[0]
        query_doc_sim = self.single_region_retrieve(query, region)
    elif region == 2:
        region_list = ["author"]
```

```
        query = query_list[1]
        query_doc_sim = self.single_region_retrieve(query, region)
    elif region == 3:
        region_list = ["head", "author"]
        query1 = query_list[0]
        query2 = query_list[1]
        query_doc_sim1 = self.single_region_retrieve(query1, 1)
        query_doc_sim2 = self.single_region_retrieve(query2, 2)
        for i in range(len(query_doc_sim1)):
            query_doc_sim[i] = query_doc_sim1[i]+query_doc_sim2[i]
    elif region == 4:
        region_list = ["body"]
        query = query_list[2]
        query_doc_sim = self.single_region_retrieve(query, region)
    elif region == 5:
        region_list = ["head", "body"]
        query1 = query_list[0]
        query2 = query_list[2]
        query_doc_sim1 = self.single_region_retrieve(query1, 1)
        query_doc_sim2 = self.single_region_retrieve(query2, 4)
        for i in range(len(query_doc_sim1)):
            query_doc_sim[i] = query_doc_sim1[i]+query_doc_sim2[i]
    elif region == 6:
        region_list = ["author", "body"]
        query1 = query_list[1]
        query2 = query_list[2]
        query_doc_sim1 = self.single_region_retrieve(query1, 2)
        query_doc_sim2 = self.single_region_retrieve(query2, 4)
        for i in range(len(query_doc_sim1)):
            query_doc_sim[i] = query_doc_sim1[i]+query_doc_sim2[i]
    elif region == 7:
        region_list = ["head","author", "body"]
        query0 = query_list[0]
        query1 = query_list[1]
        query2 = query_list[2]
        query_doc_sim0 = self.single_region_retrieve(query0, 1)
        query_doc_sim1 = self.single_region_retrieve(query1, 2)
        query_doc_sim2 = self.single_region_retrieve(query2, 4)
        for i in range(len(query_doc_sim1)):
            query_doc_sim[i] =
```

```
query_doc_sim1[i]+query_doc_sim2[i]+query_doc_sim0[i]
    else:
        # 默认为三维空间检索
        # 这里给出一个接口
        self.all_retrieve(query_list)
        return

    sorted_id = sorted(range(len(query_doc_sim)), key=lambda k:
query_doc_sim[k], reverse=True)
    for i in range(len(sorted_id)):
        print("排名第"+str(i+1)+"位的是文档D"+str(sorted_id[i]
```

单一文本域的查询

单一文本域的查询与对应的全文本的查询方式类似,只不过只是在对应文档的单个文本域进行向量相似度计算,注意对应的文本的向量长度也应该是对应文本域的向量长度

如下所示为单个文本的查询接口:

PYTHON

```
def single_region_retrieve(self, query, region):
    '''
    单个域进行查询的接口
    '''
    if region == 1:
        region_list = "head"
    elif region == 2:
        region_list = "author"
    elif region == 4:
        region_list = "body"
    else:
        return
    # 先进行一个词干还原
    query = query.lower().strip().translate(str.maketrans("","",
string.punctuation))
    words = query.split(' ')
    # print(words)
    word_list = []
    tagged_sent = pos_tag(words)     # 获取单词词性
    wnl = WordNetLemmatizer()
    for tag in tagged_sent:
        wordnet_pos = get_wordnet_pos(tag[1]) or wordnet.NOUN
```

```
        word_list.append(wnl.lemmatize(tag[0], pos=wordnet_pos)) # 词形还
原
    query = ' '.join(word_list)
    # 对应的idf
    term_doc_num = self.cal_idf()
    doc_region_length = self.cal_doc_region_length(region)
    # 进行计算获得query的向量
    query_word = query.split(" ")
    query_vec = []
    for i in range(len(set(query_word))):
        query_vec.append(0)
        for j in range(len(query_word)):
            if query_word[j] == list(set(query_word))[i]:
                query_vec[i] += 1
    for i in range(len(query_vec)):
        query_vec[i] = round(math.log10(query_vec[i]+1),4)
    # print(query_vec)
    # 下面进行文档的向量构建
    all_doc_vec = []
    doc_num = len(os.listdir(self.data_dir))
    for i in range(doc_num):
        doc_vec = []
        for j in range(len(set(query_word))):
            # 找到对应的词项id
            word_id = self.term_id_map[list(set(query_word))[j]]
            # print(list(set(query_word))[j])
            doc_vec.append(0)
            if word_id < len(self.index):
                posting_list = self.index[word_id]
                # print(posting_list)
                for post in posting_list[1]:
                    if post.split('_')[0] == str(i) and post.split('_')
[1] == region_list:
                        doc_vec[j] += int(post.split('_')[2])
                doc_vec[j] =
round(math.log10(doc_vec[j]+1),4)*term_doc_num[word_id]
        all_doc_vec.append(doc_vec)
    # print(all_doc_vec)
    all_doc_vec.append(query_vec)
    ans = pd.DataFrame(all_doc_vec)
```

```
    query_doc_sim = []
    for i in range(len(all_doc_vec)-1):

query_doc_sim.append((ans.loc[i,:]*ans.loc[9,:]).sum()/doc_region_length[
i])
    return query_doc_sim
```

对应的单个向量的长度的计算接口代码如下:

PYTHON

```
def cal_doc_region_length(self, region):
    '''
    计算向量的长度
    '''
    if region == 1:
        region_list = "head"
    elif region == 2:
        region_list = "author"
    elif region == 4:
        region_list = "body"
    else:
        return
    doc_num = len(os.listdir(self.data_dir))
    term_doc_num = self.cal_idf()
    # 初始化列表
    doc_region_length = [0]*doc_num
    all_doc_vec = [[0 for i in range(len(self.index))] for j in
range(doc_num)]
    # print(all_doc_vec)
    for index in self.index:
        term_id = index[0]
        postings = index[1]
        # print(postings)
        for post in postings:
            post_id = post.split('_')[0]
            post_num = post.split('_')[2]
            if post.split('_')[1] == region_list:
                all_doc_vec[int(post_id)][int(term_id)] += int(post_num)
    for doc_id in range(len(all_doc_vec)):
        for term_id in range(len(doc)):
            all_doc_vec[doc_id][term_id] *= term_doc_num[term_id]
```

```
        for i in range(len(all_doc_vec[doc_id])):
            doc_region_length[doc_id] += math.pow(all_doc_vec[doc_id][i],
2)
        doc_region_length[doc_id] =
round(math.sqrt(doc_region_length[doc_id]),4)
    return doc_region_length
```

程序运行结果

```
 _____          _              _____     _       _                 _
|  __ \        | |            |  __ \   | |     (_)               | |
| |__) |__   __| |_ _ __ _   _ | |__) |__| |_ _ __ _  _____   __ _| |
|  ___/ _ \ / _ \ __| '__| | | ||  _  // _ \ __| '__| |/ _ \ \ / / _` | |
| |  | (_) |  __/ |_| |  | |_| | | \ \  __/ |_| |  | |  __/\ V / (_| | |
|_|   \___/ \___|\__|_|   \__, | |_|  \_\___|\__|_|  |_|\___| \_/ \__,_|_|
                           __/ |
                          |___/

----------------------------正在进行倒排索引构建-----------------------------
----------------------------正在构建倒排索引的IDF-----------------------------
----------------------------正在计算倒排索引文档向量长度-----------------------------
----------------------------构建完成，下面进行按域检索-----------------------------
请选择你要检索的域:
1.诗歌名 2.诗人 3.诗歌内容 4.诗歌名和诗人 5.诗歌名和诗歌内容 6.诗人和诗歌内容 7.诗歌名和诗人和诗歌内容 8.全文本检索 9.退出
诗歌名域查询语句:  freedom
诗歌内容域查询语句:  my love
排名第1位的是文档D4，与Query余弦相似度SC(Q, D4) : 0.08802803903201581
排名第2位的是文档D6，与Query余弦相似度SC(Q, D6) : 0.009916792078226684
排名第3位的是文档D3，与Query余弦相似度SC(Q, D3) : 0.009539291856051171
排名第4位的是文档D1，与Query余弦相似度SC(Q, D1) : 0.009379496443338362
排名第5位的是文档D8，与Query余弦相似度SC(Q, D8) : 0.00935762505841519
排名第6位的是文档D2，与Query余弦相似度SC(Q, D2) : 0.004385310305453646
排名第7位的是文档D7，与Query余弦相似度SC(Q, D7) : 0.002642754273504273
排名第8位的是文档D5，与Query余弦相似度SC(Q, D5) : 0.0
排名第9位的是文档D9，与Query余弦相似度SC(Q, D9) : 0.0
----------------------------当前查询完毕，下面新一轮查询-----------------------------
请选择你要检索的域:
1.诗歌名 2.诗人 3.诗歌内容 4.诗歌名和诗人 5.诗歌名和诗歌内容 6.诗人和诗歌内容 7.诗歌名和诗人和诗歌内容 8.全文本检索 9.退出
诗人域查询语句:  Pablo
诗歌内容域查询语句:  bid
排名第1位的是文档D6，与Query余弦相似度SC(Q, D6) : 0.06113100989958988
排名第2位的是文档D3，与Query余弦相似度SC(Q, D3) : 0.01365261134613277
排名第3位的是文档D1，与Query余弦相似度SC(Q, D1) : 0.0
排名第4位的是文档D2，与Query余弦相似度SC(Q, D2) : 0.0
排名第5位的是文档D4，与Query余弦相似度SC(Q, D4) : 0.0
排名第6位的是文档D5，与Query余弦相似度SC(Q, D5) : 0.0
排名第7位的是文档D7，与Query余弦相似度SC(Q, D7) : 0.0
排名第8位的是文档D8，与Query余弦相似度SC(Q, D8) : 0.0
排名第9位的是文档D9，与Query余弦相似度SC(Q, D9) : 0.0
----------------------------当前查询完毕，下面新一轮查询-----------------------------
```