智能信息检索课程实验报告

简易布尔检索系统

姓名：程俊豪

学号：22920202204546

院系：信息学院

专业：人工智能

年级：2020 级

一、实验目的

编写一个基于tf-idf权重的简易布尔检索系统。根据用户输入and、or、not或者混合输入实现查询，并返回基于tf-idf权重排序后的检索结果（只有and和or查询可以排序）。

二、实验思路

create_inverted_index.py 构建倒排索引表

1. term_pattern

   编写匹配中文单词的正则表达式

   ```
   term_pattern = re.compile(r'\s*([\u4e00-\u9fa5]+)\s+')  # 匹配中文词项
   ```

2. build_stopwords_bloom（）函数

   因为要去除停用词，首先构建停用词构成的布隆过滤器，编写函数build_stopwords_bloom（）。将从Github上搜索到的中文停用词表cn_stopwords.txt构造成布隆过滤器，加快在构造过程中去除语料库中停用词的过程。

   ``

   ```
   def build_stopwords_bloom():
       """
       构建stop_words的bloom过滤器
       :return: stopwords_bloom
       """
       stop_file = open('cn_stopwords.txt', 'r', encoding='utf-8')  # 打开停用词表文件
       lines = stop_file.readlines()
       stopwords_bloom = ScalableBloomFilter(initial_capacity=MAX_SIZE)
       for line in lines:
           lline = line.rstrip()
           stopwords_bloom.add(lline)
       return stopwords_bloom
   ```

3. traverse_data () 函数

   遍历语料库文件夹data下的所有文件并返回未去除停用词的词项字典。对每个文件都做find_all操作，将这个文件中的所有单词和附属信息保存下来，方便后续操作。

   ``

   ```
   def travers_data(base):
       """
       遍历整个文件夹并返回未进行停用词去除的词项字典
       :param base:
       :return: terms_dic
       """
       terms_dic = dict()
       '''
           保存所有的词项的集合
           term_dic的key是词项 value是一个字典
   ```

```
        value的key是df value是docID的列表
    '''
    fullname_list = []
    count = 1
    for root, ds, fs in os.walk(base):
        for f in fs:
            '''
             对每个文件操作
            '''
            fullname = os.path.join(root, f)
            fullname_list.append(fullname)
            docID = count
            print("start doc" + str(docID))
            cur_f = open(fullname, 'r', encoding='gbk', errors='ignore')
            content = cur_f.read()
            all_words = re.findall(term_pattern, content)  # 找到该文件中所有的中文匹配项 存放在
all_words

            # 对于在当前文件中出现的所有单词
            for word in all_words:
                # 如果该单词之前已经出现过
                if terms_dic.__contains__(word):
                    if docID not in terms_dic[word].keys():
                        terms_dic[word][docID] = 1  # 在该文档中第一次出现tf赋初始值为1
                    else:
                        terms_dic[word][docID] += 1  # tf+1
                # 如果该单词在之前没有出现过
                else:
                    terms_dic[word] = {}
                    terms_dic[word][docID] = 1

            cur_f.close()
            print("finished doc" + str(docID))
            count += 1

    return terms_dic
```

4. del_stopwords_in_the_end(origin_dic: dict, stopwords_bloom) 函数

   在写入倒排索引表的csv文件之前删去当前倒排索引表中的所有停用词。

   ``

```
def del_stopwords_in_the_end(origin_dic: dict, stopwords_bloom):
    """
    在写入csv之前进行停用词的删除
    :return:
    """
    key_list = list(origin_dic.keys())
    for key in key_list:
        if key in stopwords_bloom:
            del origin_dic[key]
    return origin_dic
```

5. 对遍历文件夹data构成的倒排索引表根据字典序进行排序并计算每个词项的doc-frequency

   ``

```
# 先将字典根据词项排序
sorted_dic = dict((k, result_dic[k]) for k in sorted(result_dic.keys()))

# 统计doc_frequency
doc_frequency = []
for term in sorted_dic.keys():
    doc_frequency.append(len(sorted_dic[term]))
```

6. calculate_tf_idf(sorted_dic, doc_frequency)函数

   计算每个词项在每篇文档中的tf-idf权重。

   ``

```
def calculate_tf_idf(sorted_dic, doc_frequency):
    """
    计算所有词项在出现文档中的的tf_idf
    :return:
    """
    cnt = 0
    for i in sorted_dic:
        for j in sorted_dic[i]:
            tf_idf = (1 + math.log(sorted_dic[i][j], 10)) * math.log(N / doc_frequency[cnt],
10)
            sorted_dic[i][j] = tf_idf
        cnt += 1
    return sorted_dic
```

7. dic_to_csv(dic_data: dict, doc_frequency)

   将倒排索引表写入inverted_index_table_add_tf_idf.csv文件

   ``

```
"""
将字典写入csv文件
:param dic_data: 处理完的字典数据
:param doc_frequency: 统计后的df列表
:return:
"""
k = list(dic_data.keys())
df = doc_frequency
v = list(dic_data.values())
df = pd.DataFrame(list(zip(k, df, v)), columns=['词项', 'doc_frequency', 'docID-tf'])
df.to_csv('inverted_index_table_add_tf_idf.csv', encoding='utf-8-sig')
```

最终的倒排索引表是一张下面所展示的表格，由于数据过多，不作过多的展示

	词项	doc_frequency	docID-tf
0	一一对应	2	{39: 2.4248816366310666, 269: 2.4248816366310666}

	词项	doc_frequency	docID-tf
1	一丁点儿	12	{88: 1.6467303862474232, 89: 1.6467303862474232, 140: 1.6467303862474232, 154: 1.6467303862474232, 216: 2.6381608683110387, 318: 1.6467303862474232, 331: 1.6467303862474232, 399: 1.6467303862474232, 489: 1.6467303862474232, 521: 1.6467303862474232, 522: 1.6467303862474232, 526: 1.6467303862474232}
2	一下子	251	{1: 0.32623791081401, 4: 0.48189295215871175, 12: 0.32623791081401, 17: 0.32623791081401, 20: 0.32623791081401, 21: 0.32623791081401, 22: 0.42444530769177774, 24: 0.32623791081401, 26: 0.32623791081401, 27: 0.32623791081401, 33: 0.42444530769177774, 34: 0.32623791081401, 35: 0.48189295215871175, 37: 0.32623791081401, 39: 0.32623791081401, 40: 0.5801003490364794, 42: 0.48189295215871175, 44: 0.42444530769177774, 45: 0.32623791081401, 47: 0.32623791081401, 49: 0.32623791081401, 51: 0.48189295215871175, 53: 0.32623791081401, 54: 0.32623791081401, 56: 0.32623791081401, 57: 0.42444530769177774, 58: 0.32623791081401, 63: 0.48189295215871175, 67: 0.5542684247502523, 69: 0.42444530769177774, 72: 0.32623791081401, 73: 0.6019409298212757, 74: 0.32623791081401, 77: 0.32623791081401, 80: 0.42444530769177774, 81: 0.32623791081401, 82: 0.32623791081401, 84: 0.42444530769177774, 88: 0.32623791081401, 92: 0.32623791081401, 93: 0.32623791081401, 94: 0.32623791081401, 95: 0.32623791081401, 96: 0.48189295215871175, 97: 0.42444530769177774, 99: 0.32623791081401, 100: 0.48189295215871175, 102: 0.5226527045695454, 104: 0.32623791081401, 106: 0.32623791081401, 107: 0.32623791081401, 108: 0.32623791081401, 109: 0.32623791081401, 110: 0.5801003490364794, 111: 0.48189295215871175, 112: 0.42444530769177774, 115: 0.32623791081401, 118: 0.32623791081401, 120: 0.42444530769177774, 121: 0.32623791081401, 125: 0.32623791081401, 128: 0.32623791081401, 138: 0.42444530769177774, 142: 0.48189295215871175, 143: 0.32623791081401, 144: 0.32623791081401, 145: 0.32623791081401, 146: 0.42444530769177774, 149: 0.32623791081401, 150: 0.32623791081401, 151: 0.32623791081401, 153: 0.42444530769177774, 154: 0.48189295215871175, 158: 0.32623791081401, 159: 0.42444530769177774, 161: 0.42444530769177774, 163: 0.5226527045695454, 167: 0.32623791081401, 172: 0.48189295215871175, 174: 0.32623791081401, 182: 0.42444530769177774, 183: 0.42444530769177774, 184: 0.32623791081401, 185: 0.32623791081401, 187: 0.32623791081401, 188: 0.32623791081401, 189: 0.32623791081401, 193: 0.48189295215871175, 196: 0.32623791081401, 197: 0.32623791081401, 198: 0.32623791081401, 201: 0.32623791081401, 206: 0.32623791081401, 208: 0.32623791081401, 209: 0.32623791081401, 210: 0.32623791081401, 211: 0.42444530769177774, 213: 0.32623791081401, 216: 0.32623791081401, 217: 0.32623791081401, 219: 0.32623791081401, 220: 0.32623791081401, 222: 0.32623791081401, 223: 0.32623791081401, 227: 0.42444530769177774, 232: 0.32623791081401, 233: 0.48189295215871175, 235: 0.32623791081401, 236: 0.32623791081401, 238: 0.32623791081401, 240: 0.42444530769177774, 243: 0.32623791081401, 244: 0.32623791081401, 247: 0.32623791081401, 249: 0.32623791081401, 252: 0.42444530769177774, 253: 0.42444530769177774, 257: 0.32623791081401, 258: 0.6019409298212757, 263: 0.42444530769177774, 266: 0.32623791081401, 268: 0.32623791081401, 269: 0.32623791081401, 270: 0.42444530769177774, 271: 0.32623791081401, 273: 0.42444530769177774, 274: 0.42444530769177774, 275: 0.32623791081401, 279: 0.32623791081401, 281: 0.5226527045695454, 282: 0.32623791081401, 283: 0.32623791081401, 284: 0.32623791081401, 285: 0.42444530769177774, 286: 0.32623791081401, 287: 0.42444530769177774, 296: 0.42444530769177774, 299: 0.32623791081401, 300: 0.42444530769177774, 301: 0.32623791081401, 302: 0.32623791081401, 303: 0.42444530769177774, 304: 0.32623791081401, 305: 0.32623791081401, 307: 0.32623791081401, 308: 0.32623791081401, 309: 0.32623791081401, 310: 0.42444530769177774, 314: 0.32623791081401, 317: 0.32623791081401, 318: 0.32623791081401, 322: 0.5226527045695454, 323: 0.32623791081401, 324: 0.5226527045695454, 325:

	词项	doc_frequency	docID-tf
			0.32623791081401, 331: 0.42444530769177774, 334: 0.32623791081401, 335: 0.32623791081401, 338: 0.32623791081401, 342: 0.5542684247502523, 350: 0.32623791081401, 352: 0.42444530769177774, 355: 0.32623791081401, 359: 0.32623791081401, 360: 0.32623791081401, 362: 0.32623791081401, 363: 0.5226527045695454, 364: 0.32623791081401, 366: 0.42444530769177774, 367: 0.32623791081401, 368: 0.32623791081401, 369: 0.48189295215871175, 373: 0.42444530769177774, 374: 0.48189295215871175, 379: 0.42444530769177774, 381: 0.42444530769177774, 385: 0.32623791081401, 386: 0.32623791081401, 387: 0.42444530769177774, 389: 0.32623791081401, 394: 0.42444530769177774, 395: 0.32623791081401, 396: 0.32623791081401, 397: 0.32623791081401, 400: 0.42444530769177774, 401: 0.32623791081401, 402: 0.42444530769177774, 403: 0.32623791081401, 406: 0.32623791081401, 407: 0.32623791081401, 408: 0.32623791081401, 410: 0.42444530769177774, 412: 0.32623791081401, 413: 0.5542684247502523, 414: 0.42444530769177774, 415: 0.48189295215871175, 417: 0.32623791081401, 421: 0.42444530769177774, 423: 0.5226527045695454, 425: 0.32623791081401, 426: 0.32623791081401, 428: 0.32623791081401, 430: 0.32623791081401, 433: 0.42444530769177774, 436: 0.48189295215871175, 437: 0.32623791081401, 440: 0.32623791081401, 441: 0.42444530769177774, 444: 0.32623791081401, 445: 0.32623791081401, 446: 0.32623791081401, 448: 0.42444530769177774, 451: 0.32623791081401, 457: 0.32623791081401, 461: 0.42444530769177774, 463: 0.32623791081401, 465: 0.32623791081401, 466: 0.32623791081401, 467: 0.32623791081401, 472: 0.48189295215871175, 473: 0.32623791081401, 474: 0.32623791081401, 476: 0.42444530769177774, 479: 0.32623791081401, 482: 0.32623791081401, 487: 0.32623791081401, 488: 0.42444530769177774, 490: 0.42444530769177774, 491: 0.32623791081401, 492: 0.32623791081401, 493: 0.32623791081401, 496: 0.32623791081401, 499: 0.32623791081401, 500: 0.42444530769177774, 501: 0.5801003490364794, 504: 0.32623791081401, 505: 0.32623791081401, 506: 0.32623791081401, 509: 0.32623791081401, 511: 0.42444530769177774, 514: 0.32623791081401, 515: 0.32623791081401, 517: 0.32623791081401, 518: 0.42444530769177774, 519: 0.5542684247502523, 520: 0.32623791081401, 521: 0.32623791081401, 522: 0.42444530769177774, 525: 0.32623791081401, 526: 0.32623791081401, 532: 0.42444530769177774}
3	一不小心	213	{1: 0.39753202885631045, 2: 0.39753202885631045, 5: 0.39753202885631045, 6: 0.39753202885631045, 8: 0.39753202885631045, 11: 0.39753202885631045, 24: 0.39753202885631045, 25: 0.39753202885631045, 27: 0.39753202885631045, 33: 0.39753202885631045, 34: 0.5172010937792193, 37: 0.5172010937792193, 40: 0.39753202885631045, 41: 0.39753202885631045, 42: 0.6368701587021279, 43: 0.39753202885631045, 48: 0.39753202885631045, 51: 0.39753202885631045, 52: 0.39753202885631045, 54: 0.5872030092554863, 55: 0.39753202885631045, 56: 0.6368701587021279, 63: 0.5172010937792193, 66: 0.39753202885631045, 68: 0.39753202885631045, 69: 0.5172010937792193, 71: 0.39753202885631045, 72: 0.39753202885631045, 74: 0.39753202885631045, 75: 0.5172010937792193, 77: 0.39753202885631045, 79: 0.39753202885631045, 81: 0.39753202885631045, 88: 0.5172010937792193, 90: 0.5872030092554863, 92: 0.39753202885631045, 93: 0.39753202885631045, 96: 0.5872030092554863, 99: 0.39753202885631045, 100: 0.5172010937792193, 103: 0.39753202885631045, 104: 0.5872030092554863, 106: 0.39753202885631045, 109: 0.39753202885631045, 112: 0.5872030092554863, 117: 0.39753202885631045, 120: 0.39753202885631045, 122: 0.39753202885631045, 125: 0.39753202885631045, 126: 0.39753202885631045, 127: 0.5172010937792193, 128: 0.39753202885631045, 129: 0.39753202885631045, 130: 0.5172010937792193, 135: 0.39753202885631045, 136: 0.39753202885631045, 139: 0.39753202885631045, 140: 0.5172010937792193, 144: 0.5172010937792193, 145: 0.5172010937792193, 146: 0.39753202885631045, 149: 0.39753202885631045, 150: 0.39753202885631045, 153: 0.39753202885631045, 155: 0.39753202885631045, 156: 0.5172010937792193, 159: 0.39753202885631045, 161: 0.39753202885631045, 162: 0.39753202885631045, 163:

词项	doc_frequency	docID-tf
		0.39753202885631045, 164: 0.5172010937792193, 165: 0.5172010937792193, 168: 0.39753202885631045, 172: 0.39753202885631045, 176: 0.39753202885631045, 178: 0.39753202885631045, 179: 0.39753202885631045, 182: 0.39753202885631045, 186: 0.39753202885631045, 187: 0.39753202885631045, 188: 0.39753202885631045, 193: 0.5872030092554863, 194: 0.853154632208578, 200: 0.39753202885631045, 201: 0.39753202885631045, 203: 0.39753202885631045, 204: 0.5872030092554863, 206: 0.39753202885631045, 208: 0.39753202885631045, 209: 0.5872030092554863, 212: 0.39753202885631045, 213: 0.39753202885631045, 215: 0.39753202885631045, 220: 0.39753202885631045, 225: 0.5872030092554863, 228: 0.5172010937792193, 231: 0.39753202885631045, 237: 0.39753202885631045, 238: 0.6368701587021279, 239: 0.39753202885631045, 245: 0.39753202885631045, 247: 0.5172010937792193, 249: 0.39753202885631045, 250: 0.39753202885631045, 251: 0.39753202885631045, 253: 0.39753202885631045, 254: 0.39753202885631045, 258: 0.5172010937792193, 259: 0.5172010937792193, 260: 0.39753202885631045, 264: 0.39753202885631045, 270: 0.6368701587021279, 271: 0.39753202885631045, 272: 0.39753202885631045, 274: 0.5172010937792193, 275: 0.5172010937792193, 276: 0.39753202885631045, 278: 0.5172010937792193, 281: 0.5172010937792193, 287: 0.5172010937792193, 288: 0.5172010937792193, 289: 0.5172010937792193, 290: 0.39753202885631045, 299: 0.39753202885631045, 300: 0.39753202885631045, 302: 0.5172010937792193, 303: 0.39753202885631045, 304: 0.39753202885631045, 306: 0.5172010937792193, 308: 0.39753202885631045, 314: 0.5172010937792193, 315: 0.39753202885631045, 317: 0.39753202885631045, 319: 0.5172010937792193, 320: 0.5172010937792193, 322: 0.39753202885631045, 323: 0.39753202885631045, 324: 0.39753202885631045, 330: 0.39753202885631045, 337: 0.39753202885631045, 342: 0.39753202885631045, 344: 0.39753202885631045, 345: 0.39753202885631045, 347: 0.39753202885631045, 354: 0.5172010937792193, 357: 0.5172010937792193, 358: 0.706872074178395, 360: 0.39753202885631045, 361: 0.39753202885631045, 362: 0.39753202885631045, 367: 0.39753202885631045, 368: 0.5872030092554863, 370: 0.39753202885631045, 373: 0.39753202885631045, 374: 0.39753202885631045, 379: 0.39753202885631045, 392: 0.39753202885631045, 393: 0.5172010937792193, 394: 0.39753202885631045, 395: 0.5872030092554863, 396: 0.5172010937792193, 401: 0.39753202885631045, 403: 0.39753202885631045, 404: 0.39753202885631045, 406: 0.5172010937792193, 407: 0.5172010937792193, 410: 0.39753202885631045, 414: 0.39753202885631045, 415: 0.5872030092554863, 417: 0.39753202885631045, 418: 0.39753202885631045, 419: 0.6368701587021279, 422: 0.5172010937792193, 423: 0.39753202885631045, 427: 0.39753202885631045, 429: 0.5172010937792193, 432: 0.39753202885631045, 433: 0.5872030092554863, 435: 0.39753202885631045, 436: 0.5872030092554863, 439: 0.5872030092554863, 441: 0.39753202885631045, 442: 0.39753202885631045, 443: 0.39753202885631045, 445: 0.39753202885631045, 446: 0.39753202885631045, 448: 0.5172010937792193, 451: 0.39753202885631045, 457: 0.5172010937792193, 459: 0.39753202885631045, 463: 0.39753202885631045, 464: 0.39753202885631045, 467: 0.39753202885631045, 468: 0.5172010937792193, 473: 0.6368701587021279, 480: 0.5172010937792193, 483: 0.5172010937792193, 484: 0.39753202885631045, 487: 0.39753202885631045, 489: 0.39753202885631045, 496: 0.39753202885631045, 500: 0.39753202885631045, 503: 0.5172010937792193, 504: 0.39753202885631045, 508: 0.5172010937792193, 509: 0.39753202885631045, 515: 0.5872030092554863, 517: 0.39753202885631045, 518: 0.5172010937792193, 523: 0.39753202885631045, 526: 0.5172010937792193, 530: 0.5872030092554863, 532: 0.5172010937792193}

doc_length_normalization.py 对文档长度实现归一化操作

1. 读取倒排索引表，并读取需要的数据项存入列表。

``

```
data = pd.read_csv('inverted_index_table.csv', sep=',', header=0, usecols=[1, 2],
encoding='utf-8')
term = list(data['词项'])
doc_frequency = list(data['doc_frequency'])
load_dic = dict(zip(term, doc_frequency))
```

2. 同样需要构建停用词构成的布隆过滤器。

``

```
def build_stopwords_bloom():
    """
    构建stop_words的bloom过滤器
    :return: stopwords_bloom
    """
    stop_file = open('cn_stopwords.txt', 'r', encoding='utf-8')  # 打开停用词表文件
    lines = stop_file.readlines()
    stopwords_bloom = ScalableBloomFilter(initial_capacity=MAX_SIZE)
    for line in lines:
        lline = line.rstrip()
        stopwords_bloom.add(lline)
    return stopwords_bloom
```

3. length_normalization()函数

进行文档长度的归一化，计算公式如下：

假设文档doc1.txt中有3个单词x,y,z（实际上可能有很多个甚至上千个），将文档看作由每个词项作为一个维度，词频作为每一维度的分量，由此构成的向量空间。

$$x = (1 + log10(tf_x))/\sqrt{(1 + log10(tf_x))^2 + (1 + log10(tf_y))^2 + (1 + log10(ty_z))^2}$$

对y，z以及其他词项也进行相同的计算，以此对每个文档求出归一化后文档中每个词项的长度，并存入新文档，方便后续的计算。

``

```
def length_normalization():
    """
    进行文件的长度归一化操作
    :return: 长度归一化后的词项：权重文件
    """
    count = 1
    for root, ds, fs in os.walk(base):
        for f in fs:
            '''
             对每个文件操作
            '''
            # 创建当前文件的单词字典
            words_dic = dict()
            fullname = os.path.join(root, f)
            docID = count
            print("start normalize doc" + str(docID))
```

```
            cur_f = open(fullname, 'r', encoding='gbk', errors='ignore')
            content = cur_f.read()
            all_words = re.findall(term_pattern, content)
            # 找到该文件中所有的中文匹配项 存放在all_words

            for word in all_words:
                if word in stop_words_bloom:
                    all_words.remove(word)
                else:
                    # 如果该单词之前已经出现过
                    if words_dic.__contains__(word):
                        words_dic[word] += 1
                    # 如果该单词在之前没有出现过
                    else:
                        words_dic[word] = 1
            # 计算tf取对数+1
            for word in words_dic:
                words_dic[word] = 1 + math.log10(words_dic[word])

            L2 = 0
            # 计算文档长度的2范数
            for word in words_dic:
                L2 += math.pow(words_dic[word], 2)
            L2 = math.sqrt(L2)

            # 计算归一化长度
            for word in words_dic:
                words_dic[word] /= L2
            cur_f.close()

            # 写入文档
            t = list(words_dic.keys())
            l = list(words_dic.values())
            df = pd.DataFrame(zip(t, l), columns=['词项', '归一化长度'])
            path = './length_normalized_docs_txt/normalized_doc' + str(docID) + '.txt'
            df.to_csv(path, encoding='utf-8-sig')
            print("finished normalize doc" + str(docID))
            count += 1
```

实现文档长度归一化后的结果如下，以data文件夹中的第一个文档为例。

	词项	归一化长度
0	想	0.030245
1	吃	0.032106
2	包子	0.010702
3	馋嘴	0.020914
4	转发	0.04397
5	频道	0.010702
6	博	0.040631

boolean_retrieval() 实现and、or、not和混合检索

1. 读取倒排索引表，处理词项、docID等在后续需要用到的数据。

``

```
data = pd.read_csv('inverted_index_table.csv', sep=',', header=0, usecols=[1, 3],
encoding='utf-8')
term = list(data['词项'])
docID_tf = list(data['docID-tf'])
key_list_of_docID_tf = []  # 存放docID_tf中每一项字典的key构成的列表
# 将从csv读取的字符串形式的字典转换成字典
for i in range(len(docID_tf)):
    docID_tf[i] = eval(docID_tf[i])
    key_list_of_docID_tf.append(list(docID_tf[i].keys()))
# 处理完后docID—tf是字典构成的列表
normalized_doc_list = []
dic1 = dict(zip(term, key_list_of_docID_tf))  # 是一个key为词项 value为出现该词项的文件序号的列表
files = os.listdir('./data/')
```

2. load_doc()函数

   载入经过长度归一化后的文档数据

``

```
def load_doc():
    """
    在程序准备阶段读取532个长度归一化后的文件到内存
    :return: normalized_doc_list
    """
    print("loading docs")
    base = './length_normalized_doc_with_stopwords/'
    cnt = 1
    while cnt <= 532:
        fullname = base + 'normalized_doc' + str(cnt) + '.csv'
        docID = cnt
        # print("loading doc" + str(docID))
        df_1 = pd.read_csv(fullname)
        key = []
        value = []
        for item in df_1["词项"]:  # “词项”用作键
            key.append(item)
        for j in df_1["归一化长度"]:  # “归一化长度”用作值
            value.append(j)
        dic = dict(zip(key, value))
        normalized_doc_list.append(dic)
        cnt += 1
```

3. and_retrieval()函数

   实现简单的and检索，没有查询词数量的限制，只需要连接词是小写的and即可，对查询结果根据tf-idf权重进行排序，返回查询结果，并写入txt文件

``

```
def and_retrieval():
```

```
    """
     求所有词项docID的交集
     :return:
     """
    print("请输入以and连接的查询项：", end='')
    s = input().replace(' ', '')  # 去除用户输入可能多输的空格
    start_time = time.perf_counter()  # 用户输入结束，计时开始
    input_term_list = list(s.split("and"))  # 以AND分割词项存入列表
    if len(input_term_list) == 1:
        print("参数不足，请重新输入")
        return

    # 将输入当成文档 构造长度归一化的输入文档
    input_term_dic = {}
    for word in input_term_list:
        input_term_dic[word] = 1
    L2 = math.sqrt(len(input_term_list))
    for word in input_term_dic:
        input_term_dic[word] /= L2

    # 找到词项的索引 从0开始
    index_of_terms = []

    try:
        for t in input_term_list:
            index_of_terms.append(term.index(t))

    except ValueError:
        print("查询失败！ 查询词不存在！")
        return

    #初始化集合为检索的第一个词出现的文档集合
    docID_set = set()
    for item in key_list_of_docID_tf[index_of_terms[0]]:
        docID_set.add(item)
    docID_set.update(key_list_of_docID_tf[index_of_terms[0]])

    # 用intersection方法求交集
    for j in range(len(index_of_terms)):
        s2 = set()
        s2.update(key_list_of_docID_tf[index_of_terms[j]])
        docID_set = docID_set.intersection(s2)

    # 把求得的目标文档列表和输入查询构成的文档传入排序函数 输出排序后的结果
    sorted_res = relevance_sort(index_of_terms, input_term_dic, docID_set)
    write_query(sorted_res, input_term_list)

    print('查询时间:%s毫秒' % ((end_time - start_time) * 1000))
```

and检索的部分实例结果如下：

```
请输入以and连接的查询项：夏天and西瓜
共找到398篇文档
查询结果如下：
No. 1 doc_No.402 data.zdqk.txt
No. 2 doc_No.403 data.zdqq.txt
No. 3 doc_No.11 data.cz.txt
No. 4 doc_No.340 data.zcyv.txt
No. 5 doc_No.214 data.zbot.txt
No. 6 doc_No.491 data.zesq.txt
No. 7 doc_No.57 data.sq.txt
No. 8 doc_No.333 data.zcwn.txt
No. 9 doc_No.261 data.zccl.txt
No. 10 doc_No.377 data.zdji.txt
```

4. or_retrieval()函数

实现简单的or检索，没有查询词数量的限制，只需要连接词是小写的or即可，返回查询结果

``

```
def or_retrieval():
    """
    多词or查询
    :return:
    """
    print("请输入以or连接的查询项：", end='')
    s = input().replace(' ', '')  # 去除用户输入可能多输的空格
    start_time = time.perf_counter()  # 用户输入结束，计时开始
    input_term_list = list(s.split("or"))  # 以AND分割词项存入列表
    if len(input_term_list) == 1:
        print("参数不足，请重新输入")
        return
    # 将输入当成文档 构造长度归一化的输入文档
    input_term_dic = {}
    for word in input_term_list:
        input_term_dic[word] = 1
    L2 = math.sqrt(len(input_term_list))
    for word in input_term_dic:
        input_term_dic[word] /= L2

    # 找到词项的索引 从0开始
    index_of_terms = []

    try:
        for t in input_term_list:
            index_of_terms.append(term.index(t))

    except ValueError:
        print("查询失败！查询词不存在！")
        return

    docID_set = set()

    for j in index_of_terms:
```

```
            docID_set.update(key_list_of_docID_tf[j])

        docID_list = list(docID_set)
        docID_list.sort()
        # 把求得的目标文档列表和输入查询构成的文档传入排序函数 输出排序后的结果
        final_res,end_time=relevance_sort(index_of_terms, input_term_dic, docID_list)

        print('查询时间:%s毫秒' % ((end_time - start_time) * 1000))
```

or检索的部分实例测试结果如下：

```
请输入以or连接的查询项：飞机or汽车
共找到522篇文档
查询结果如下：
No. 1 doc_No.134 data.zasj.txt
No. 2 doc_No.490 data.zesi.txt
No. 3 doc_No.126 data.zaql.txt
No. 4 doc_No.237 data.zbup.txt
No. 5 doc_No.128 data.zaqr.txt
No. 6 doc_No.364 data.zdeq.txt
No. 7 doc_No.159 data.zayv.txt
No. 8 doc_No.131 data.zard.txt
No. 9 doc_No.470 data.zejk.txt
No. 10 doc_No.312 data.zcqg.txt
```

5. not_retrieval()函数

实现简单的not检索，以not开头连接词项进行查询

``

```
def not_retrieval():
    """
    not查询 可以只有一项 not a1 不需要排序，则不需要长度归一化
    :return:
    """
    print("请输入以not开头 not连接的查询项（形式如：not a1 not a2)：", end='')
    s = input().replace(' ', '')  # 去除用户输入可能多输的空格
    start_time = time.perf_counter()  # 用户输入结束，计时开始
    input_term_list = list(s.split("not"))  # 以AND分割词项存入列表
    del input_term_list[0]
    res = set(range(1, 532))  # 初始化集合为全集 然后逐个求差集得到结果
    r = set()
    # 对出现的词项逐个求差集
    for i in input_term_list:
        try:
            tmp = set(dic1[i])
            r = res.difference(tmp)
        except KeyError:  # 如果词项i不在语料库中 在出现keyerror时抛出异常 将tmp置为全集
            tmp = set(range(1, 532))
            r = res.difference(tmp)

    final_res = list(r)
    final_res.sort()
```

```
    end_time = time.perf_counter()

    print("共找到" + str(len(final_res)) + "篇文档")
    print("查询结果如下：")

    if len(final_res) == 0:
        print("查询结果为空！")
    else:
        t = 1
        for i in final_res:
            print("No. %d" % t, end=' ')
            t += 1
            print("doc_No.{}".format(i), end=" ")
            print(files[i - 1])

    print('查询时间:%s毫秒' % ((end_time - start_time) * 1000))
```

测试结果如下：

```
请输入以not开头 not连接的查询项（形式如：not a1 not a2）: not 想想 not 想象
共找到91篇文档
查询结果如下：
No. 1 doc_No.6 data.by.txt
No. 2 doc_No.9 data.co.txt
No. 3 doc_No.11 data.cz.txt
No. 4 doc_No.14 data.ek.txt
No. 5 doc_No.20 data.ff.txt
No. 6 doc_No.24 data.fz.txt
No. 7 doc_No.36 data.jf.txt
No. 8 doc_No.41 data.mi.txt
No. 9 doc_No.46 data.mx.txt
No. 10 doc_No.55 data.qn.txt
```

6. mix_retrieval() 函数

实现简单的混合检索，输入格式限制比较严格，需要根据括号“）”的位置对输入字符串进行划分操作

``

```
def mix_retrieval():
    """
    and or not 混合检索 不需要排序
    :return:
    """
    print("请按如下格式输入(括号请按全角形式输入): (A1 and A2...AN) and (B1 or B2...BM) and not
(C1 or C2...CK)")
    s = input().replace(' ', '')  # 去除用户输入可能多输的空格
    start_time = time.perf_counter()  # 用户输入结束，计时开始
    try:
        lst = list(s.split("）"))  # 将输入以）分割为三部分每一部分形式为 （a1 and a2    （b1 or
b2  （c1 or c2
        # 处理第一个括号
        part1 = lst[0][1:]  # 从a1的第一个字符开始
        lstpart1 = list(part1.split("and"))  # 提取第一部分的所有词项
```

```
        r = dic1[lstpart1[0]]  # 先初始化r集合为第一部分中的第一个词项出现的所有文档的列表
        if len(lstpart1) > 1:  # 如果第一部分的词项数大于一个
            for i in range(1, len(lstpart1)):
                r = list(set(r).intersection(dic1[lstpart1[i]]))  # 对包含剩下词项的文件序号列表求交集

        # 处理第二个括号
        part2 = lst[1][4:]  # 从b1的第一个字符开始 不读前面的and
        lstpart2 = list(part2.split('or'))  # 第二个括号内的词项
        r1 = dic1[lstpart2[0]]
        if len(lstpart2) > 1:  # 如果第二部分长度大于一个词项
            for i in range(1, len(lstpart2)):
                r1 = list(set(r1).union(dic1[lstpart2[i]]))  # 对包含剩下词项的文件序号列表求并集
        r = list(set(r).intersection(r1))  # 第一第二个括号对应的文档序号列表求交集

        # 处理第三个括号
        part3 = lst[2][7:]  # 从c1的第一个字符开始 不读前面的and not
        lstpart3 = list(part3.split('or'))  # 第三个括号内的词项
        r2 = dic1[lstpart3[0]]  # 先求第三项的并集
        if len(lstpart3) > 1:
            for i in range(1, len(lstpart3)):
                r2 = list(set(r2).union(dic1[lstpart3[i]]))  # 对包含剩下词项的文件名列表求并集
        r = list(set(r).difference(r2))  # 第一第二个括号和第三个括号对应的文档列表求差集

        end_time = time.perf_counter()  # 找到所有文档，计时结束

        print("共找到" + str(len(r)) + "篇文档")
        t = 1
        for i in r:
            print("No. %d" % t, end=' ')
            t += 1
            print("doc_No.{}".format(i), end=" ")
            print(files[i - 1])
        # 相关度排序

        print('查询时间:%s毫秒' % ((end_time - start_time) * 1000))

    except:
        print("查询失败！查询词不存在！")
```

7. relevance_sort() 函数

对and检索和not检索得到的结果进行排序并返回最终结果的函数

``

```
def relevance_sort(index_of_terms, input_term_dic: dict, doc_set):
    """
    文档相似度排序
    :param index_of_terms:查询词项在倒排索引表中的索引
    :param input_term_dic:输入的词项构成的字典 values为归一化后的长度 1 + log10(tf)
    :param doc_set: 按照要求求得的词项的集合
    :return:
    """
    cos = {}  # 记录文档doc和输入相似度的字典
    doc_term = {}
    for doc in doc_set:
```

```
            doc_term[doc] = []
            cos[doc] = 0.0
            # 对每篇文档求其和输入文档的相似度cos(doc,input)
            for i in index_of_terms:
                # 从所有词项的列表中根据索引找到i词项在当前文档中的tf_idf （已读入docID_tf)
                tmp = 0
                if term[i] in normalized_doc_list[doc - 1]:
                    tmp = normalized_doc_list[doc - 1][term[i]]
                    doc_term[doc].append(tmp)
                cos[doc] += tmp * input_term_dic[term[i]]

        if len(cos) > 1:
            # 结果序列长度大于一时输出排序后的结果
            cos_tuple = sorted(cos.items(), key=lambda x: x[1], reverse=True)
            cos_dic_list = []
            for cos_pair in cos_tuple:  # 将元组转化为字典
                cos_pair_dic = {}
                cos_pair_list = list(cos_pair)
                cos_pair_dic[cos_pair_list[0]] = cos_pair_list[1]
                cos_dic_list.append(cos_pair_dic)
            final_res = []

            end_time = time.perf_counter()

            print("共找到" + str(len(cos_dic_list)) + "篇文档")
            t = 1
            print("查询结果如下：")
            for item in cos_dic_list:
                for j in set(item):
                    final_res.append(files[j - 1])
                    print("No. %d" % t, end=' ')
                    t += 1
                    print("doc_No.{}".format(j), end=" ")
                    print(files[j - 1])
            return final_res,end_time  # 返回排序后的文件列表
        else:
            end_time = time.perf_counter()
            # 结果序列长度小于等于1直接输出
            if len(cos) == 0:
                print("查询结果为空！")

            else:
                final_res = []
                for i in list(cos.keys()):
                    final_res.append(files[i - 1])
                print("共找到" + str(len(final_res)) + "篇文档")
                print("查询结果如下：")
                t = 1
                print("No. %d" % t, end=' ')
                t += 1
                print(final_res[0])
                return final_res,end_time
```

8. wirte_query()函数

将and检索的最终结果写入查询词同名txt文件的函数，为后续通过Java程序评测做准备。

```
```

```
def write_query(sorted_res, input_term_list):
    """
    将最终排序后的结果写到指定文件夹中 文件名为 词项+空格+词项...txt
    :param sorted_res: 排序后的查询结果文档
    :param input_term_list: 输入的词项表
    :return:
    """
    path = './query/'  # 写入到query文件夹
    filename = input_term_list[0]
    for i in range(1, len(input_term_list)):
        filename += ' ' + input_term_list[i]
    filename += '.txt'
    fp = open(path + filename, 'w+')
    for item in sorted_res:
        str1 = item + '\n'
        fp.write(str1)
    fp.close()
```

9. mode_choose()函数

显示查询模式供用户选择

``

```
def mode_choose():
    """
    选择需要查询的模式
    :return: None
    """
    print("请输入对应的数字选择所需的检索")
    print("[1]AND查询")
    print("[2]OR查询")
    print("[3]NOT查询")
    print("[4]多词的AND、OR、NOT查询")
    print("输入其他数字退出系统")
```

10. main()函数

```
if __name__ == "__main__":
    load_doc()  # 载入语料库和准备过程
    while 1:
        mode_choose()
        n = input()
        if n == '1':
            and_retrieval()
            print("按下任意键继续。", end="")
            t = input()
        elif n == '2':
            or_retrieval()
            print("按下任意键继续。", end="")
            t = input()
        elif n == '3':
            not_retrieval()
            print("按下任意键继续。", end="")
            t = input()
        elif n == '4':
            mix_retrieval()
            print("按下任意键继续。", end="")
```

```
            t = input()
        else:
            break
```

三、总结

在本次智能信息检索的实验中学习到了专业领域的信息检索知识，打破了局限于单一的字符串匹配实现查询的认识。实验中仍存在着许多不足，数据结构存在冗余情况，没有实现最佳的算法效率，仍有待改善。