**《数字内容安全》实验报告**

**姓 名：** 丁昭旭  **学 号：** 16020031011

**年 级：** 2016级 **专 业：** 计算机科学与技术

1. **实验名称**

实验四 文本信息检索

1. **实验要求**

* 针对“语料库.txt”文件，实现基于布尔模型的检索系统和基于TF-IDF的检索系统。（每一行看做是一个文档）
* 基于布尔模型的检索系统，

输入 以布尔表达式形式输入，一次检索单词数量不超过三个（AA and BB or CC ）

返回所有的击中的结果（超过10项的只显示前10项）

* 基于TF-IDF的检索系统，

输入 不超过8个字的短语，系统首先自动进行分词，按照TF-IDF的值求和排序返回前10项结果。

1. **实验结果**

由于每一行看成一个文档的话导致文档太多而每个文档中的词语太少，也破坏了段落的语义完整性，效果不佳。故我将每一篇文章看做一个文档，希望老师理解。

1. 布尔模型

首先将输入的句子中的括号和and,not以及or去除，然后通过split函数将输入句子分割成词语组成的数组，这样字符串内仅仅含有具有实际意义的词语了，就能够以数组的形式来获取其中的词语：

*parsed\_set = \_string.replace(')', '').replace('(', '').split(' ')*

*bool\_word = ['and', 'not', 'or']*

*for word in bool\_word:*

*if word in parsed\_set:*

*parsed\_set.remove(word)*

*for word in parsed\_set:*

*if word == 'and' or word == 'or' or word == 'not' or word == '':*

*parsed\_set.remove(word)*

吸取实验中的教训，我通过sort函数根据字符串长度来由长到短进行排序，以避免将较长的单词看成短的词语而破坏其完整性：

*parsed\_set.sort(key=len, reverse=True)*

这样在替换词语的过程中就会从更长的单词入手，就能够正确处理“我们”和“我”并存的情况了，而不会将“我们”中的“我”看成独立的一个词，导致错误的出现。

得到句子所包含的词语之后，我们需要得到这个布尔表达式的所有可能布尔取值，为此我们需要取得与词语个数长度等同的所有布尔取值数组，故我编写了get\_all\_possible\_bool函数：

*def get\_all\_possible\_bool(size):*

*bool\_arr = []*

*if size == 2:*

*return [[False], [True]]*

*for i in range(size):*

*bi\_num = bitarray(bin(i)[2:].zfill((math.ceil(math.sqrt(size)))))*

*bool\_arr.append(bi\_num.tolist())*

*return bool\_arr*

根据这一函数，我通过二进制格式的数字循环来得到所有的可能取值，然后通过bitarray库十分方便的tolist函数将二进制数转化为布尔数，通过这样的循环我们可以得到所有可能取得的布尔数组，后面可以将它传入循环，筛选出对于输入的句子可能取得的布尔数组。

对所有的可能布尔取值，我设置一个循环对其进行遍历，筛选出对于句子能够取到的数值，可以通过eval函数来十分方便地获取替换过布尔值的字符串：

*for i in range(len(all\_bool)):*

*string\_to\_parse = str(\_string)*

*for j in range(len(parsed\_set)):*

*string\_to\_parse = string\_to\_parse.replace(parsed\_set[j], str(all\_bool[i][j]))*

***result = eval(string\_to\_parse)***

*if result == True:*

*right\_bool.append(all\_bool[i])*

我们就将字符串中的字符替换为所有可能布尔数组中的布尔值（‘True’和‘False’），然后传入eval函数进行判断，若得到True这说明这一布尔取值是其中的一种可能情况，保存之。

取得了所有可能的布尔取值之后，我们在语料库中进行循环，通过以下语句判断是否符合条件：

*for paragraph\_dict in whole\_dict.values():*

*word\_set = set()*

*for char in paragraph\_dict.keys():*

*word\_set.add(char)*

*ok = True*

*for bool\_list in bool\_values:*

*for i in range(len(bool\_list)):*

***if bool\_list[i] == True and char\_arr\_to\_match[i] not in word\_set or bool\_list[i] == False and char\_arr\_to\_match[i] in word\_set:***

***ok = False***

*unmatched\_num = unmatched\_num +1*

*break*

*if ok:*

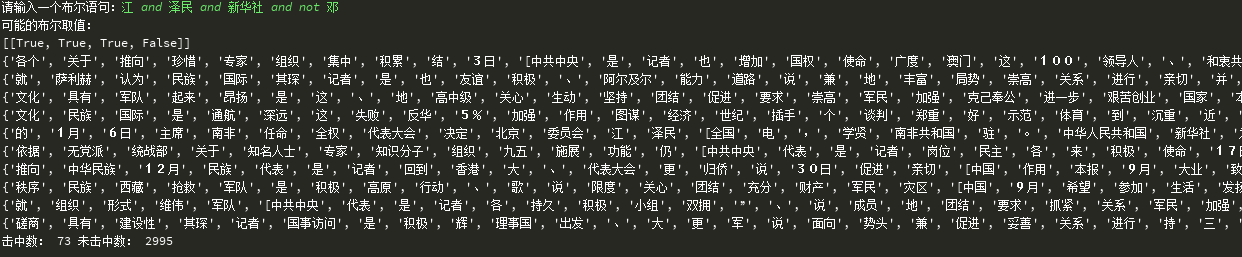
*matched\_num = matched\_num + 1*

*all\_matched.append(word\_set)*

在循环过程中，程序通过加粗的这一个判断来对照布尔取值、对应的词语和语料库中某一段落的词语集合：

对于每一个段落进行如下循环：在所有可能的布尔取值中，若布尔对应0但是这一段落包含布尔数对应的词语，则说明不匹配，结束对这一个段落的循环；若布尔对应1但是段落中不包含这个词，则也是不匹配。

循环过后，我们就能够得到所有击中的段落，保存在all\_matched数组中，等待后续输出。

运行结果如下：

输出的信息是命中文档的词语数组以及命中数、未命中数的统计情况。

1. TF-IDF检索系统

若要计算出所有段落的TF-IDF，我们应该首先分别得到TF和IDF值。由于TF值比较容易取得，故我们首先计算每篇文档的TF值：

1. TF的计算

首先我们通过一个函数来得到语料库中的所有词语，通过一个字典类变量保存，关键代码如下：

*while line:*

*line = f.readline()*

***info = re.findall(r'([\S]\*?)-([\S]\*?)-([\S]\*?)-([\S]\*?)', line)***

*if len(info) == 0:*

*if len(paragraph\_dict) != 0:*

*whole\_dict[info\_str] = paragraph\_dict*

*paragraph\_dict = dict()*

*continue*

***info\_str = info[0][0] + '-' + info[0][1] + '-' + info[0][2]***

***match = re.findall(r'([\S]\*?)/([\S]\*?)', line)***

*for item in match:*

*word = item[0]*

*if word in stop\_words or re.match(r'([\S]\*?)-([\S]\*?)-([\S]\*?)-([\S]\*?)', word):*

*continue*

*if not word in paragraph\_dict.keys():*

*paragraph\_dict[word] = 1*

*else:*

*paragraph\_dict[word] = paragraph\_dict[word] + 1*

这样我们可以得到语料库中的所有词语，每篇文档通过前面‘19980101-01-003’这样的字符串来标记，分别保存每篇文章中的词频。

TF值计算的相关代码如下：

*for str\_info, paragraph\_dict in whole\_dict.items():*

*child\_tf\_dict = dict()*

*whole\_count = sum(paragraph\_dict.values())*

*for word, count in paragraph\_dict.items():*

*word\_tf = count / whole\_count*

*child\_tf\_dict[word] = word\_tf*

*tf\_dict[str\_info] = child\_tf\_dict*

即对于语料库中每一篇文档，我们通过sum\_count来得到所有的词频之和，然后将每个词语的词频除以词频和来得到TF值，将其以文档为单位保存在一个TF字典中。

1. IDF的计算

IDF是针对所有文档得到的，是根据所有出现的词语来计算其出现的频率。对此我首先编写一个函数来得到语料库中所有的词语：

*words\_set = set()*

*for child\_dict in whole\_dict.values():*

*for word in child\_dict.keys():*

*words\_set.add(word)*

*return words\_set*

通过以下代码我们计算每一个词语的IDF值：

*for word in words\_set:*

*count\_document = len(whole\_dict)*

*exists\_doc\_count = 0*

*for words in whole\_dict.values():*

*words\_in\_doc = [item for item in words.keys()]*

*# print(words\_in\_doc)*

*if word in words\_in\_doc:*

*exists\_doc\_count = exists\_doc\_count + 1*

*idf\_value = math.log(count\_document / exists\_doc\_count)*

*idf\_dict[word] = idf\_value*

即对于每个词语，在语料库中进行检索，若某篇文档包含这个词语，则将其出现的次数加一，最终根据所有文章数和出现次数来计算出该词语IDF的值

1. TF-IDF的计算

已经得到了TF和IDF之后，计算TF-IDF就比较简单了，我们针对每篇文章的TF值，从IDF字典中寻找单个词语对应的IDF值，然后相乘计算TF-IDF值，保存在一个新的字典中：

*for item in tf\_collection.find():*

*info\_str = item['info\_str']*

*tf\_dict = item['tf\_dict']*

*tf\_idf\_dict = dict()*

*for word, tf in tf\_dict.items():*

*try:*

*idf = idf\_dict[word]*

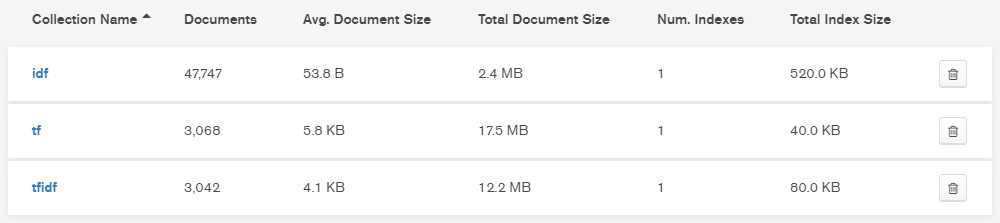
*tf\_idf = idf \* tf*

*tf\_idf\_dict[word] = tf\_idf*

*except:*

*continue*

由于语料库体量比较大，我尝试将TF-IDF值保存在MongoDB数据库中，并在计算的过程中为了避免重复计算也将TF和IDF值保存在了数据库中，即存在三个collection中：



下面是对于句子的分析，首先我们通过实验三的fmm分词函数来对输入进行分词，若遇到停词则删除，这里我使用了github上的常用中文分词表。

对于每篇文章的TF-IDF数值，我们对于分词结果中的每一个词语在TF-IDF表中进行查找，若存在则加上其对应的TF-IDF值，最终将所有计算出来的TF-IDF和进行排序处理：

for info\_str, values\_dict in tf\_idf\_dict.items():

sum = 0

for word in fmm\_result:

try:

sum = sum + values\_dict[word]

except:

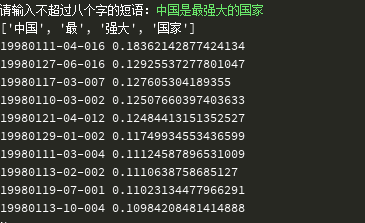
continue

calculated\_sum\_dict.append((info\_str, sum))

sorted\_dict = sorted(calculated\_sum\_dict, key=lambda d: d[1], reverse=True)

排序之后结果就会按照TF-IDF值由大到小的顺序排列，之后我们就能够根据排序结果进行输出了。

运行结果如下：



首先输出的结果是去除停词后的分词情况，然后是TF-IDF和排名在前十位的文档编号和TF-IDF值。

1. **实验总结**

这次实验有比较大的难度，我在实验过程中也遇到了大大小小很多困难和挑战，既有想不出可行的算法的困难，比如布尔模型的实现方法是我绞尽脑汁才实验出来的，也有实际操作上的困难，比如tf-idf计算的方法有不妥，导致时间效率低下，最终想到了通过数据库来解决。

在这次实验过程中，我也学到了很多有价值的东西，比如神奇的eval函数、用于弥补Python二进制处理过程中的困难的bitarray库、中文停词表的使用、进度条库tqdm等。这些方法体现了Python这门语言的多样性和全面性，熟悉它们一定会让我受益很久。

除去以上的收获，我对于布尔模型和TF-IDF模型的理念和实践方法有了深层次的认识。这一认识已经超越了课本上的理论知识和熟练运用出色的库函数比如jieba库，而是有了自己的理解和体验，这也是此次实验的价值所在。