武汉大学国家网络安全学院

实验报告

课程名称 信 息 检 索

专业年级 2018 级网络空间安全

姓　　名 李伟诚

学　　号 2018302080100

协 作 者 无

实验学期 2020-2021 学年 第二 学期

课堂时数 32 课外时数

填写时间 2021 年 6 月 1 日

实验介绍

【实验名称】：基于带位置信息倒排索引的多功能新闻信息检索系统

【实验目的】：

信息检索是从大规模非结构化数据（通常是文本）的集合（通常保存在计算机上）中找出满足用户信息需求的资料（通常是文档）的过程。在此定义下，信息检索在过去往往是某些特定人士才能从事的一项活动。然而，当今世界发生了巨大的变化，当上亿的用户每天使用 Web 搜索引擎或者查找邮件时，他们实际上都在从事信息检索活动。信息检索已经替代传统的数据库式搜索迅速成为信息访问的主要形式。

信息检索按照所处理数据的规模进行区分，可以分为 3 个级别，即以 Web 搜索为代表的大规模级别、以个人信息检索为代表的小规模级别和介于二者之间的中等规模级别（包括面向企业、机构和特定领域的搜索）。本项目主要聚焦于小规模级别和中等规模级别的搜索，在这种情况下，文档存储在集中的文件系统中，关注重点不在如何处理大规模数据集，而在如何提供检索服务本身。

本项目实现的是特定领域的搜索，检索对象是一系列路透社的新闻报道文本，通过键入搜索关键词可以返回相关的新闻，因而在某种意义上也可以理解成一个关注特定新闻媒体动向的新闻检索系统。在具体的技术实现上，我构建了带位置信息的倒排索引来快速定位所需的新闻文档，并使用了 tf-idf 及其变体 wf-idf 作为查询评分函数，最终实现了 Top K 查询、布尔查询、短语查询、通配符查询、同义词查询、词干还原、拼写校正等功能，具有了实际应用的价值。

【实验环境】：

（1）电脑配置：

处理器：Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz

机带 RAM：16.0 GB

系统类型：64 位操作系统，基于 x64 的处理器

（2）操作系统

Windows 10 家庭版

（3）开发工具与版本

PyCharm 2017.2.3（配置了 anaconda 环境）

编程语言：Python 3.7.4

【参考文献】：

[1]曼宁. 信息检索导论[M]. 修订版. 北京:人民邮电出版社, 2019

[2]位置信息倒排索引, https://blog.csdn.net/Necstyle/article/details/38156991

[3]布尔检索及其查询优化, https://blog.csdn.net/qll125596718/article/details/8437670

[4]词典及容错式检索, https://blog.csdn.net/djl806943371/article/details/103630880/

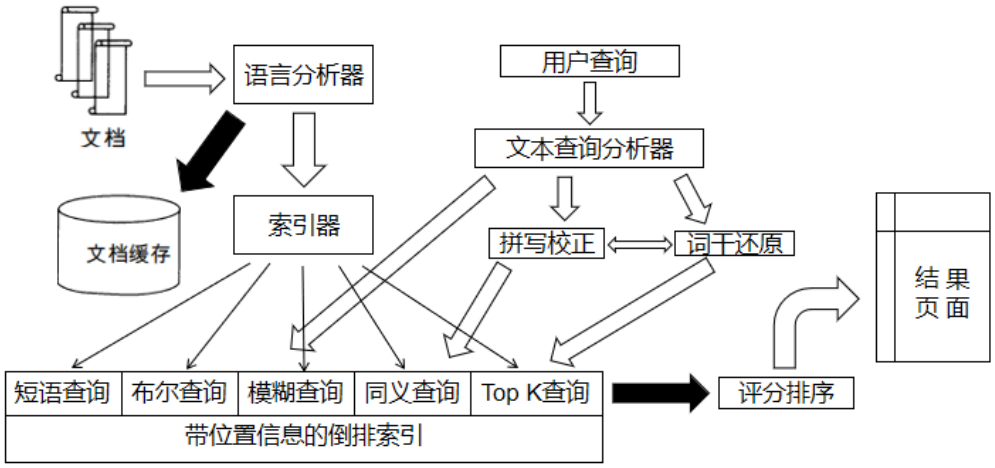
[5]python 时间函数, https://blog.csdn.net/weixin_30569303/article/details/112357544

[6]信息检索系统评价指标, https://blog.csdn.net/fengjiancangyue/article/details/43746651

实验内容

【实验方案设计】：

本项目实现的是一个新闻检索系统，检索对象是一系列路透社（Reuters）的新闻报道文本。通过键入关键词，系统会返回与此相关的新闻，因而在某种意义上也可以理解成一个关注特定新闻媒体动向的新闻检索系统。在技术层面，我的总体思路是利用 tf-idf 及其亚线性尺度变换（wf-idf）来表征词项的权重，然后将词项权重求和得到文档的评分；构建带位置信息的倒排索引来快速定位所需的新闻文档，之所以要加上位置信息，是为了支持短语查询；最终实现 Top K 查询、布尔查询、短语查询、通配符查询、同义词查询、词干还原、拼写校正等功能，具有了实际应用的价值。下图是本项目的流程图：



接下来对整个项目的运行方法以及各部分的方案、代码设计进行详细说明。

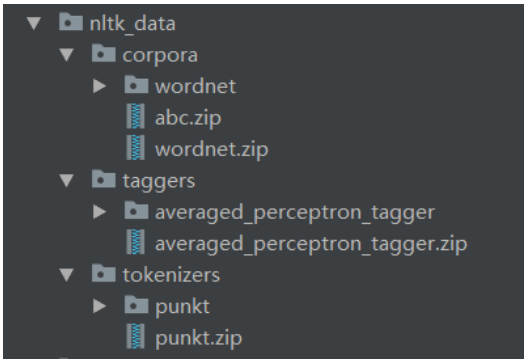
一、 运行方法

1、下载依赖的语料库。

在初次运行程序之前，需要下载词干还原所依赖的语料库，具体代码如下：

```
# 下载需要的依赖文件
nltk.download("wordnet")
nltk.download("averaged_perceptron_tagger")
nltk.download("punkt")
nltk.download("maxnet_treebank_pos_tagger")
```

下载完毕后，在/user/AppData/Roaming 目录下会出现一个 nltk_data 文件夹，如下所示：



2、开始运行。

进入 SearchSystem 目录，运行以下命令即可：

```
python main.py
```

main.py 还支持以下命令行参数：

```
usage: main.py [-h] [-s] [-c] [-k TOPK] [-i]

SearchSystem

optional arguments:
  -h, --help            show this help message and exit
  -s, --stemming        run stemming or not
  -c, --spelling_correcting
                        correct spelling or not
  -k TOPK, --topk TOPK  return top k results
  -i, --indexing        build a new index or not
```

参数添加代码如下：

```
parser = argparse.ArgumentParser(description='SearchSystem')
parser.add_argument('-s', '--stemming', default='False', action='store_true', help='run
stemming or not')
parser.add_argument('-c', '--spelling_correcting', default='False', action='store_true',
                    help='correct spelling or not')
parser.add_argument('-k', '--topk', default=5, type=int, help='return top k results')
parser.add_argument('-i', '--indexing', default='False', action='store_true', help='build a new
index or not')

args = parser.parse_args()
```

其中，“-s”（或“--stemming”）参数代表进行词干还原；“-c”（或“--spelling_correcting”）参数代表进行拼写校正；“-k”（或“--topk”）参数指定 TopK 查询中返回的文档数；“-i”（或“--indexing”）代表建立索引，在文档收集完毕之后、开始检索之前进行。

二、 实现功能的原理及代码

1、带位置信息的倒排索引

1.1 概述

很多复杂的或技术性的概念、机构名和产品名等都是由多个词语组成的复合词或短语。例如，用户希望能够将类似于 Stanford University 的查询中的两个词看成一个整体，所以一篇含有句子 The inventor Stanford Ovshinsky never went to university 的文档不会与该查询匹配。要支持短语查询，只列出词项所在的文档列表的倒排记录表不足以满足要求。在实际场景中比较常用的一种方式是采用位置信息索引，在这种索引中，对每个词项，以如下方式存储倒排记录：文档 ID:[位置 1,位置 2,...]。

下面是一个例子：

```
{
    "word1":{
        "1":[5,6,10],
        "5":[10,20,30]
    },
```

```
        "word2":{
            "2":[5,6,10],
            "33":[15,28,30]
        }
    }
```

以“word1”为例，该词项出现在了 ID 为 1 和 5 的文档中，其中文档 1 中出现的位置是 5、6、10。

1.2 实现

在本项目中，我们一开始会事先建立好索引并保存在文件中，每次运行程序时将索引加载到内存即可。

建立索引的代码如下：

```
def createIndex(directname):
    invertedIndex = {}
    path = tools.projectpath.replace("SearchSystem", "Reuters")
    files = os.listdir(path)
    for file in files:
        print("analyzing file: ", file)
        # 每个文档的词项 list
        content = PreprocessFile.preProcess(path + '/' + file)
        docId = getDocID(file)

        num = 0  # word 在文档中的位置
        for word in content:
            # if word.isdigit():
            #     num += 1
            #     continue
            if word not in invertedIndex:
                docList = {}
                docList[docId] = [num]
                invertedIndex[word] = docList
            else:
                if docId not in invertedIndex[word]:
                    invertedIndex[word][docId] = [num]
                else:
                    invertedIndex[word][docId].append(num)
            num += 1

    # 给倒排索引中的词项排序
    invertedIndex = sortTheDict(invertedIndex)
    # 获取词项列表
    wordList = getWordList(invertedIndex)
    printIndex(invertedIndex)
    # 将数据写入文件中
```

```
    tools.writeToFile(invertedIndex, tools.projectpath + 'invertIndex.json')
    tools.writeToFile(wordList, tools.projectpath + 'wordList.json')
```

思路很清晰，就是对每一个文件进行处理和分词，将某篇文档中包含的所有词项写入倒排索引中，同时标上位置信息（若倒排索引中已经有该词项则仅需添加位置信息）。接下来给倒排索引中的词项排序并获取词项列表，最后将生成的倒排索引和词项列表保存到本地。

最终建立的索引如下所示：


```
{"ANTI-US": {"12472": [3]}, "ANTIBIOTICOS": {"497": [4], "7354": [4]}, "ANTIBIOTICS": {"2168": [0]}, "ANTIBODIES":
[4]}, "ANTONSON": {"474": [6]}, "ANW": {"1673": [83]}, "ANWAT": {"6961": [12], "7030": [12]}, "ANWI": {"8965": [4]
"12731": [0, 59], "12732": [0, 57], "16108": [5, 34, 125, 197, 290], "16119": [0, 67], "19020": [0, 17], "19022":
"16108": [39], "16119": [14], "18362": [101], "19022": [18]}, "ANova": {"3643": [8, 34, 58, 103]}, "AO": {"15780":
{"1393": [5, 42], "2881": [21, 38]}, "APD": {"5396": [26, 56], "7266": [6], "21275": [6]}, "APEA": {"8117": [27, 5
"APEX": {"19223": [3]}, "API": {"1331": [0, 14], "1343": [0, 40, 97, 147], "3015": [0, 31], "3364": [0], "6294": [
"8209": [39, 40, 102], "8882": [108, 138, 269], "9039": [0, 16], "11007": [124, 171, 329], "11491": [109, 152, 219
15], "14732": [0, 102, 148], "18146": [384], "18311": [0, 16], "18840": [58], "19832": [63], "20093": [100], "2070
15220": [74]}, "APPAREL": {"2567": [2], "2645": [3]}, "APPEAL": {"3166": [3], "10238": [3], "17823": [1], "20458
[4], "2647": [5], "10830": [2], "20649": [1]}, "APPEARS": {"4824": [3], "10780": [1]}, "APPLAUD": {"11213": [1]},
{"7447": [2]}, "APPLIANCES": {"2660": [2]}, "APPLICATION": {"2939": [6, 16], "2940": [4, 12], "2948": [8], "5664":
{"64": [1], "134": [1], "1643": [1], "2275": [0], "3038": [0], "5043": [0], "6042": [0], "18828": [0], "19379": [0
"APPOINTED": {"12503": [6]}, "APPOINTEES": {"9526": [5]}, "APPOINTS": {"1588": [1]}, "APPRECIATE": {"5183": [7],
17], "8308": [2], "19378": [4]}, "APPROACHED": {"736": [4], "2512": [5]}, "APPROACHES": {"13966": [1]}, "APPROPRIA
"6108": [2], "9054": [8, 18], "10075": [7, 15], "10081": [6], "17943": [6]}, "APPROVAL": {"162": [6], "2983": [6],
```


2、查询评分

2.1 概述

在文档集规模很大的时候，满足查询条件的结果文档数量可能会非常多，往往会大大超过用户能够浏览的文档数目。因此，对搜索引擎来说，对文档进行评分和排序非常重要。为此，对于给定的查询，搜索引擎应当能够计算出每个匹配文档的得分。

评分排序的具体方法是：通过查询文档对应的向量计算出其评分，然后排序。

这里采用了两种方法进行评分。一种是 tf-idf，一种是 wf-idf。其中，tf-idf 由两部分组成：词项频率（tf）和逆文档频率（idf）。wf-idf 中的 wf 则是 tf 的亚线性尺度变换。

下图是 tf-idf 的计算公式：

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

TF-IDF

Term x within document y

$tf_{x,y}$ = frequency of x in y

df_x = number of documents containing x

N = total number of documents

首先介绍词项频率 tf。在具体查询中，我们不能仅考虑词项在文档域中出现与否这两种情形，还应该考虑词项出现的频率。在自由文本查询中，系统将查询简单地看成是多个词组成的集合。在查询时，我们先基于每个查询词项与文档的匹配情况对文档打分，然后对所有查询词项上的得分求和。这样就对文档中的每个词项都赋予了一个权重，它取决于该词项在文档中出现的次数，这种权重计算的结果称为词项频率 tf。对于文档 d，利用上述 tf 权重计算方式得到的权重集合可以看成是文档的一个经过量化以后得到的浓缩版本。

接下来介绍逆文档频率 idf。原始的词项频率会面临一个问题，即在和查询进行相关度计算时，所有的词项都被认为是同等重要的。实际上某些词项对于相关度计算来说并没有太大的作用（即区分能力）。我们需要使用一种机制来降低某些出现次数过多的词项在相关性

计算中的重要性，一个很直接的想法就是给文档集频率较高的词项赋予较低的权重，其中文档集频率指的是词项在文档集中出现的次数。实际中，一个更常用的因子是文档频率 df，它表示的是出现某一词项的所有文档的数目。这是因为文档评分的目的是区分文档，所以最好采用基于文档粒度的统计量而不是基于整个文档集的统计量来计算。由于 df 本身往往较大，所以通常需要将它映射到一个较小的取值范围中去。为此，假定所有文档的数目为 N，词项 t 的 idf（inverse document frequency, 逆文档频率）的定义如下：

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

介绍完 tf 和 idf 的概念后，下面谈谈 tf-idf 权重的计算：对于每篇文档中的每个词项，可以将其 tf 和 idf 组合在一起形成最终的权重。tf-idf 权重机制对文档 d 中的词项 t 赋予的权重如下：

$$\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

换句话说，$\text{tf-idf}_{t,d}$ 按照如下的方式对文档 d 中的词项 t 赋予权重：

(1) 当 t 只在少数几篇文档中多次出现时，权重取值最大（此时能够对这些文档提供最强的区分能力）；

(2) 当 t 在一篇文档中出现次数很少，或者在很多文档中出现，权重取值次之（此时对最后的相关度计算作用不大）；

(3) 如果 t 在所有文档中都出现，那么权重取值最小。

这样，就可以把文档看成是一个向量，其中的每个分量都对应词典中的一个词项，分量值为 tf-idf 权重值。这种向量形式对于评分和排序十分重要。在这里我们采用的评分指标如下所示：

$$Score(q,d) = \sum_{t \in q} \text{tf-idf}_{t,d}$$

2.2 实现

项目中计算 tf-idf 权重的代码如下：

```
def get_tfidf_Score(index,fileNum,docID,wordList):
    score = 0
    docID = str(docID)

    for word in wordList:

        if word not in index or docID not in index[word]:
            continue
        tf = len(index[word][docID])
        df = len(index[word])
        idf = cmath.log10(fileNum / df).real
        # print("filenum / df",fileNum / df, "df: ",df, " idf: ", idf )
        score += tf * idf
    return score
```

wf-idf 则在 tf-idf 的基础上进行改进，属于 tf 的一种亚线性尺度变换：通过 log 计算削弱词项频率对评分的影响，因为一篇文章中单词出现 n 次不代表其权重扩大 n 倍。其计算公式

如下：

$$wf_{t,d}=\begin{cases}1+\log tf_{t,d}, & tf_{t,d}>0,\\ 0, & \text{其他。}\end{cases}$$

下图为 wf 和 tf 的对比：

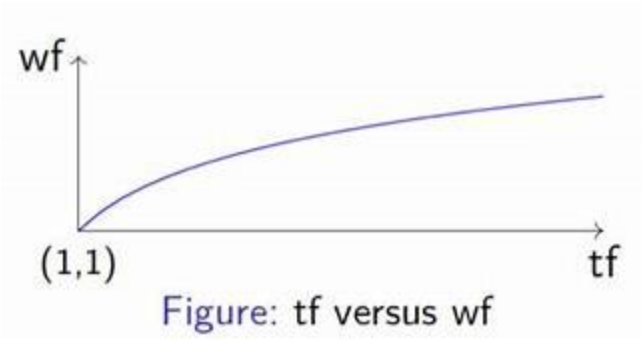


Figure: tf versus wf

将 tf 替换成 wf 后，tf-idf 就变成了 wf-idf：

$$\text{wf-idf}_{t,d}=\text{wf}_{t,d}\times\text{idf}_{t}。$$

本项目中使用 wf-idf 的流程如下：

（1）首先对 query 中出现的单词对应的文档列表取并集。

（2）随后对 query 中出现的单词对文档进行 wf-idf 计算并评分。

（3）得到所有文档对该查询的评分后再对所有文档进行排序。

wf-idf 的代码如下：

```
def get_wfidf_Score(index,fileNum,docID,wordList):
    score = 0
    docID = str(docID)
    for word in wordList:
        if word not in index or docID not in index[word]:
            continue
        tf = len(index[word][docID])
        df = len(index[word])
        wf = 1 + cmath.log10(tf).real
        idf = cmath.log10(fileNum / df).real
        score += wf * idf
    return score
```

3、Top K 查询

3.1 概述

在文档集规模很大的时候，满足查询条件的结果文档数量可能会非常多，往往会大大超过用户能够浏览的文档数目。因此，对搜索引擎来说，对文档进行评分和排序非常重要。为此，对于给定的查询，搜索引擎应当能够计算出每个匹配文档的得分。前面已经介绍了查询评分的方法，即 tf-idf 和 wf-idf。在具体的检索中，每篇文档被表示为上述权重计算结果的向量，通过它可以计算查询和每篇文档的相似度。这就是众所周知的向量空间方法。通过该方法，检索系统可以返回与查询相似度最高的 K 篇文档，这就是 Top K 查询。

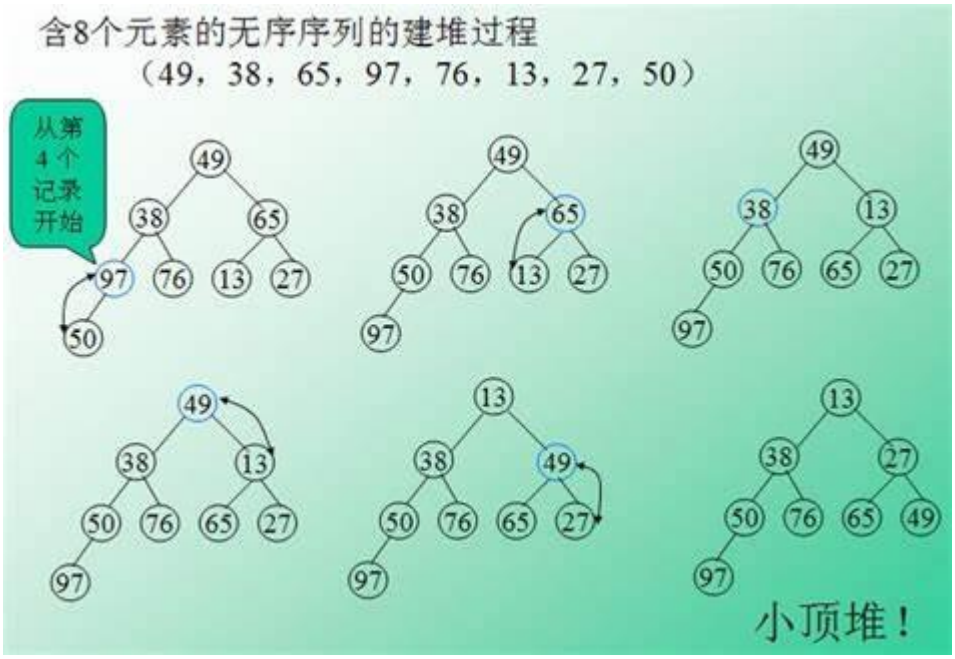
在这一部分我们主要讨论自由文本查询，即词项之间的相对次序、重要性以及其在文档

中出现的位置并不作特别指定。在这种概念下，查询实际上就是多个词项构成的集合。

3.2 实现

首先，通过前面设计的查询评分函数来计算得到所有文档的评分。然后，通过排序建好最小堆以后进行 K 次 precDown 操作。堆最后的 K 个元素即为评分前 K 大的元素。这里利用了堆排序的思想来找到 Top K 个新闻文档，主要是为了提高查找效率。

（最小堆）堆排序中的 precDown 操作如下图所示：



可见就是将小的上移，大的下移，最终形成一个小顶堆。这时堆最后的 K 个元素即为评分前 K 大的元素。

整体代码如下：

```
def TopKScore(K,index,fileNum,words,docList):
    scoreDocList = getScoreDocList(index, fileNum, words, docList)
    N = len(scoreDocList)
    if N is 0:
        return []
    scoreDocList = heapsort(scoreDocList,N,K)
    L = K
    if N < K: L = N
    return [scoreDocList[N - x - 1] for x in range(0,L)]
```

其中包含了 getScoreDocList、heapsort 两个核心函数，下面分别进行说明。

（1）getScoreDocList：

```
def getScoreDocList(index,fileNum, words,docList):
    scoreDocList = []
    for doc in docList: #对每一篇文档计算一个 score
        score = getScore.get_wfidf_Score(index,fileNum,doc,words)
        #scoreDocList 是 score 和 doc 的 list
        scoreDocList.append([score,doc])
    #print(scoreDocList)
```

```
    return scoreDocList
```

代码逻辑很清晰，就是对每一篇文档调用 get_wfidf_Score 方法计算出一个得分，最后将每篇文档的得分和文档 ID 返回。

（2）heapsort：

```
def heapsort(A, N, K):
    i = int(N / 2)
    while i >= 0:
        A = precDown(A, i, N)
        i -= 1
    i = N - 1
    end = 0
    if N - 1 > K:
        end = N - 1 - K
    while i > end:
        tem = A[0]
        A[0] = A[i]
        A[i] = tem
        precDown(A, 0, i)
        i -= 1
    return A
```

以上代码的逻辑是通过排序建好最小堆以后进行 K 次 precDown 操作。

precDown 操作的代码如下：

```
def leftChild(i):
    return 2 * i + 1
def precDown(A, i, N):
    tmp = A[i]
    while leftChild(i) < N:
        child = leftChild(i)
        if child != N - 1 and A[child + 1][0] > A[child][0]:
            child += 1
        if tmp[0] < A[child][0]:
            A[i] = A[child]
        else:
            break
        i = child
    A[i] = tmp
    return A
```

此后堆中最后的 K 个元素就是评分前 K 大的元素。具体如下：

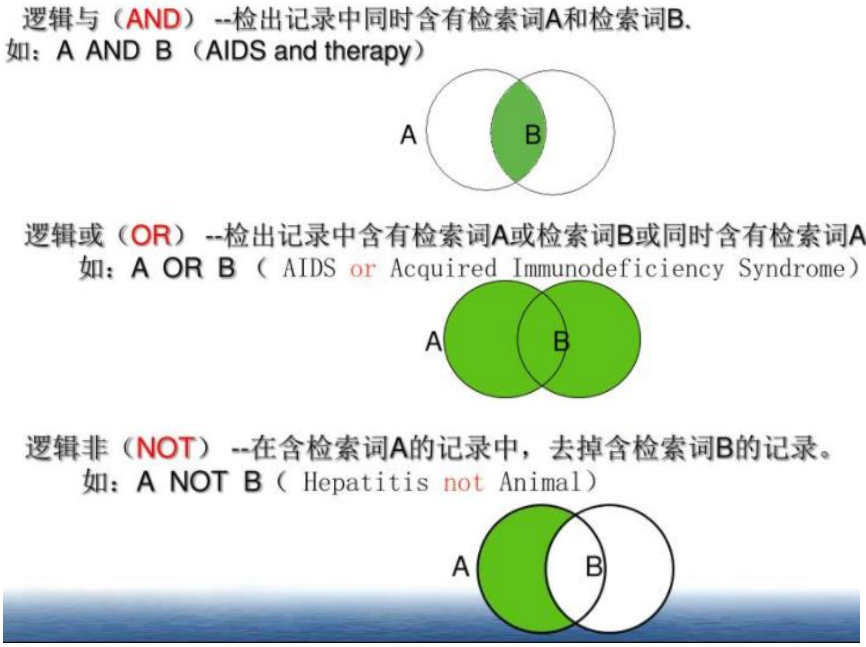
```
scoreDocList = heapsort(scoreDocList, N, K)
L = K
if N < K: L = N
return [scoreDocList[N - x - 1] for x in range(0, L)]
```

4、Bool 查询

4.1 概述

前面介绍了支持自由文本查询的向量空间模型，在自由文本查询中，查询表示为一系列词的集合，词之间没有任何操作符连接。在这种模型下，能够对文档进行评分和排序。而接下来讨论的布尔查询、短语查询以及通配符查询的处理方式与自由文本查询有所不同，下面会分别说明。

布尔检索模型接受布尔表达式查询，即通过 AND、OR 以及 NOT 等逻辑运算符将词项连接起来的查询，这几种运算符的含义如下图所示。在该模型下，每篇文档只被看成是一系列词的集合。



向量空间模型的索引结构能够用于处理布尔查询，反之则不成立，因此我们这里继续使用前面构建的倒排索引。然而，布尔模型的索引在默认情况下并不包含词项的权重信息，站在用户的角度讲，将向量空间模型和布尔查询融合并非易事：一方面，向量空间查询的处理基本上是证据累加的方式，即多个查询词项的出现会增加文档的得分；另一方面，布尔检索需要用户指定一个表达式，通过词项的出现与不出现的组合方式来选择最终的文档，文档之间并没有次序可言。数学领域中实际上存在一种称为 P 范式的方法（该方法是扩展布尔模型的一种，基本思路是对布尔检索的结果进行排序），但目前还没有实际系统这样做，因此本项目也并未对其进行实现，最终仅返回满足布尔表达式的文档集合。

4.2 实现

在本项目中，我们提供了 AND/OR/NOT 三种布尔操作符。如果选择 Bool 查询，并在命令行中输入包含 OR/AND/NOT 的查询表达式，则程序会先将查询表达式转为后序表达式，如“A OR B AND C”转为“A B C AND OR”，并使用一个栈来计算后序表达式。

```
def BoolSearch(query, index):
```

（1）将查询表达式转为后序表达式：

```
pofix = InfxiToPofix(query)
```

InfxiToPofix 函数如下所示（可以看到符号的优先级关系是 NOT>AND>OR，且支持括号）：

```
#得到后序表达式的栈
def InfxiToPofix(inputList):
    #符号优先级
```

```
precedence = ['OR','AND','NOT']
precedence ={}
precedence['OR'] = 0
precedence['AND'] = 1
precedence['NOT'] = 2
pofix_res = []
tmp = []
queries = []
for word in inputList:
    if(word == '('):
        tmp.append('(')
    elif word == ')':
        if len(queries) > 0:
            pofix_res.append(queries)
            queries = []
        sym = tmp.pop()
        while sym != '(':
            pofix_res.append(sym)
            if len(tmp) == 0:
                print("Incorrect query")
                exit(1)  #查询错误退出
                break
            sym = tmp.pop()
    elif word == 'NOT' or word == "OR" or word == 'AND':
        if len(queries) > 0:
            pofix_res.append(queries)
            queries = []
        if(len(tmp) <= 0):
            tmp.append(word)
        else:
            sym = tmp[len(tmp)-1]
            #弹出到左括号为止
            while len(tmp) >0 and sym != '('and precedence[sym] >= precedence[word]:
                # pop out
                pofix_res.append(tmp.pop())
                if(len(tmp) == 0):
                    break
                sym = tmp[len(tmp) - 1]
            # push in
            tmp.append(word)
    else:
        # put it into a
        queries.append(word)
        #pofix_res.append(word)
```

```
    if len(queries) > 0:
        pofix_res.append(queries)
    while len(tmp) > 0:
        pofix_res.append(tmp.pop())
    return pofix_res
```

（2）根据当前词项来做不同的处理。

遍历每个词项：

```
limit = len(pofix)
i = 0
while i < limit:
    item = pofix[i]
    if item != 'AND'and item !='OR': ...
    elif item == 'AND': ...
    elif item == 'OR': ...
    i += 1
```

若当前词项不是“AND”或者“OR”，则根据其是否为“NOT”分别进行处理：

```
if item != 'AND'and item !='OR':
    #print(result)
    #lookforword to see if is item
    if i < limit - 1:
        if pofix[i+1] == "NOT":
            i = i + 1
            result.append(search.serarchPhraseForBool(index, item, flag=False))
            #result.append(serachtest(index, item, flag=True))
        else:
            result.append(search.serarchPhraseForBool(index, item, flag=True))
            #result.append(serachtest(index, item, flag=False))
    else:
        result.append(search.serarchPhraseForBool(index, item, flag=True))
```

若当前词项是“AND”，则调用 listSort.andTwoList 方法对两个文档集合做“与操作”：

```
elif item == 'AND':
    if len(result) < 2:
        print("illegal query")
        return nullReturn
    else:
        list1 = result.pop()
        list2 = result.pop()
        result.append(listSort.andTwoList(list1,list2))
```

若当前词项是“OR”，则调用 listSort.mergeTwoList 方法将两个文档集合合并（即做“或操作”）：

```
elif item == 'OR':
    if len(result) < 2:
        print("illegal query")
        return nullReturn
```

```
    else:
        list1 = result.pop()
        list2 = result.pop()
        result.append(listSort.mergeTwoList(list1,list2))
```

由以上的展示可知，这一部分涉及到的两个比较重要的函数是 listSort.andTwoList 和 listSort.mergeTwoList，这两个函数的功能分别是对文档集取交集和并集：

（1）“AND”取交集：

以“A AND B”为例，我们要做的工作是：在词典中定位“A”，返回其倒排记录表；在词典中定位“B”，返回其倒排记录表；对两个倒排记录表求交集。代码如下所示：

```
def andTwoList(list1,list2):
    rlist = []
    len1 = len(list1)
    len2 = len(list2)
    n1 = 0
    n2 = 0
    while n1 < len1 and n2 < len2:
        if list1[n1] < list2[n2]:
            n1 += 1
        elif list1[n1] > list2[n2]:
            n2 += 1
        else:
            rlist.append(list1[n1])
            n1 += 1
            n2 += 1
    return rlist
```

（2）“OR”取并集：

以“A AND B”为例，我们要做的工作是：在词典中定位“A”，返回其倒排记录表；在词典中定位“B”，返回其倒排记录表；对两个倒排记录表求并集。具体做法是通过移动每个列表中的指针而将多个有序列表进行迭加。代码如下所示：

```
def mergeTwoList(list1,list2):
    rlist = []
    len1 = len(list1)
    len2 = len(list2)
    n1 = 0
    n2 = 0
    while n1 < len1 and n2 < len2 :
        if list1[n1] < list2[n2]:
            rlist.append(list1[n1])
            n1 += 1
        elif list1[n1] > list2[n2]:
            rlist.append(list2[n2])
            n2 += 1
        else:
            rlist.append(list1[n1])
```

```
            n1 += 1
            n2 += 1

    if n1 < len1:
        rlist.extend(list1[n1 : len1])
    if n2 < len2:
        rlist.extend(list2[n2 : len2])
    return rlist
```

（3）“NOT”取出 list1 中不属于 list2 的元素：

```
#list1 中不属于 list2 的
def listNotcontain(list1,list2):
    rlist = []
    len1 = len(list1)
    len2 = len(list2)
    n1 = 0
    n2 = 0
    while n1 < len1 and n2 < len2:
        if list1[n1] < list2[n2]:
            rlist.append(list1[n1])
            n1 += 1
        elif list1[n1] > list2[n2]:
            n2 += 1
        else:
            n1 += 1
            n2 += 1
    return rlist
```

5、短语查询

5.1 概述

在短语查询中，我们不能只是简单地判断两个词项是否出现在同一文档上，而且还需要检查它们的出现位置关系与查询当中是否保持一致。下图是一个例子：

这需要计算出词之间的偏移距离，我们可以利用带位置信息的倒排索引方便地实现。这

里需要说明的是，因为我们仅仅知道每个词项在文档中的相对权重，所以无法得到短语查询的向量空间评分结果。以下是一个例子：对于查询 apple computer，可以通过向量空间检索来找到这两个词项权重较高的文档，但是无法要求这两个词项连续出现；另一方面，短语检索告诉我们短语 apple computer 在文档中的出现，但是并没有任何有关短语相对频率及权重的提示信息。因此，本项目并未实现对短语查询的评分，最终仅返回包含查询短语的文档集合。

5.2 实现

利用带位置信息的倒排索引，首先得到包含 query 中所有单词的文档列表的交集，然后从这些文档集中根据位置索引查找是否有匹配的短语，具体做法是：遍历第一个词项在文档中的位置，依次检测后面的词项位置中是否包含与其匹配的位置。

```
def searchPhrase(index, words, inputList):
    if len(words) == 0:
        return []
    docQueue = queue.Queue()
    for word in words:
        docQueue.put(searchOneWord(index, word))

    while docQueue.qsize() > 1:
        list1 = docQueue.get()
        list2 = docQueue.get()
        docQueue.put(operateDocList.andTwoList(list1, list2))
    doclist = docQueue.get()

    resultList = {}

    if len(inputList) == 1:
        for doc in doclist:
            resultList[doc] = index[inputList[0]][str(doc)]
        return resultList

    # print(doclist)
    for docid in doclist:
        docid = str(docid)
        locList = []
        x = index[inputList[0]][docid]
        for loc in index[inputList[0]][docid]:
            floc = loc
            n = len(inputList)
            hasFind = True
            for word in inputList[1:n]:
                floc += 1
                try:
                    #print(index[word][docid])
```

```
                    index[word][docid].index(floc)
                except:
                    hasFind = False
                    break
            if hasFind:
                locList.append(loc)
        if len(locList) > 0:
            resultList[docid] = locList
    return resultList
```

该函数有几个值得注意的地方：

（1）得到包含 query 中所有单词的文档列表的交集

```
while docQueue.qsize() > 1:
    list1 = docQueue.get()
    list2 = docQueue.get()
    docQueue.put(operateDocList.andTwoList(list1, list2))
doclist = docQueue.get()
```

（2）遍历第一个词项在文档中的位置，依次检测后面的词项位置中是否包含与其匹配的位置。

```
for docid in doclist:
    docid = str(docid)
    locList = []
    x = index[inputList[0]][docid]
    for loc in index[inputList[0]][docid]:
        floc = loc
        n = len(inputList)
        hasFind = True
        for word in inputList[1:n]:
            floc += 1
            try:
                # print(index[word][docid])
                index[word][docid].index(floc)
            except:
                hasFind = False
                break
        if hasFind:
            locList.append(loc)
    if len(locList) > 0:
        resultList[docid] = locList
return resultList
```

6、通配符查询

6.1 概述

通配符查询又可称为模糊查询。它主要适用于以下场景：（1）用户对查询的拼写不太确定（比如，如果不太确定是 Sydney 还是 Sidney，就采用通配符查询 S*dney）；（2）用户

知道某个查询词项可能有不同的拼写版本，并且要把包含这些版本的文档都找出来（比如 color 和 colour）；（3）用户查找某个查询词项的所有变形，这些变形可能做了词干还原，但是用户并不知道搜索引擎是否进行了词干还原；（4）用户不确定一个外来词或者短语的正确拼写方式。

对于所有匹配的查询，我们可以分别得到 Top K 查询结果，也可以综合考虑各查询语句，比如，对于输入的查询 rom*，rome 和 roman 是两个可能的查询词项，那么一篇同时包含 rome 和 roma 的文档要比只包含其中一个词的文档的得分要高，当然最终文档的精确排序主要取决于不同词项在文档中的相对权重。在本项目中，我们的目的仅仅是检验通配符查询的返回结果是否正确（即是否能够返回与通配符匹配的词项的查询结果），因此并未进行评分和排序。最终的返回结果是包含各匹配词项的结果文档集合（按词项划分）。

本项目实现了“*”和“？”这两种通配符，它们的含义如下所示：

符号	意义
*	代表【0 到无穷多个】任意字符
?	代表【有且只有一个】任意字符

6.2 实现

在本项目中，我们处理带通配符的查询语句的做法是：利用正则表达式找到所有匹配的词项，利用倒排索引检索出词项对应的文档。构建支持通配符的短语查询：首先将检索到的词项保存在一个二维数组 forSearchList 中，随后对出现的每个短语组合进行短语查询。

```
# 模糊查询
elif choice == 5:
    start = time.perf_counter()
    list = searchWord.wildcardSearch(STATEMENT, INDEX, WORDLIST)
    duration = (time.perf_counter() - start) * 1000  # ms
    print("(took %.4f ms)" % duration)
```

下面介绍核心函数 wildcardSearch。

```
def wildcardSearch(statement, index, wordList):
```

该函数主要包含了以下步骤：

（1）对于输入的每一个词项，利用正则表达式找到与其匹配的所有词项（通过 searchBasedOnWildcard 这个函数，最终得到一个列表 rset），最终得到一个二维数组 forSearchList，格式为[[rset1],[rset2],[rset3]...[rsetN]]。

```
words = statement.split(' ')
# print(words)
forSearchList = []

for word in words:
    rset = searchBasedOnWildcard(word, wordList)  # word:rset
    if len(rset) > 0:
        forSearchList.append(rset)  # [[rset1],[rset2],[rset3]...[rsetN]]
    else:
        print(word, "doesn't find matching words in these articles.")
```

```
        return []
```

searchBasedOnWildcard 函数如下所示：

```
def searchBasedOnWildcard(wildcard, wordList):
    result = []
    regex = wildcard2regex(wildcard)
    if (regex == None):
        return result
    pattern = re.compile(regex, flags=re.IGNORECASE)
    for word in wordList:
        if (pattern.match(word)):
            result.append(word)
            # print(word)
    return result
```

思路很简单，首先调用 wildcard2regex 函数将通配符表达式转换成正则表达式，然后再使用 python 的 re 库进行正则表达式匹配，找到所有匹配的词项。

wildcard2regex 函数如下：

```
def wildcard2regex(wildcard):
    regex = '^'
    for i in range(wildcard.__len__()):
        if (i == 0):
            if (wildcard[i] == '*'):
                regex = regex + '[a-z]*'
            elif (wildcard[i] == '?'):
                regex = regex + '[a-z]'
            elif (not wildcard[i].isalpha()):
                return None
            else:
                regex = regex + wildcard[i]
        else:
            if (wildcard[i] == '*'):
                regex = regex + '[a-z]*'
            elif (wildcard[i] == '?'):
                regex = regex + '[a-z]'
            elif (not wildcard[i].isalpha()):
                return None
            else:
                regex = regex + wildcard[i]
    regex = regex + '$'
    return regex
```

可见能够支持“*”和“?”这两个通配符。

（2）对二维数组 forSearchList 中所有可能出现的短语组合进行短语查询。

```
i = 0
numList = []
```

```
N = len(forSearchList)
while i < N:
    numList.append(0)  # [0,0,0...,0]  N 个 0
    i += 1

resultList = {}

while 1:
    searchList = []
    statement = ""
    j = 0
    while j < N:  # j =0,1,2...,N-1
        searchList.append(forSearchList[j][numList[j]])
        statement += searchList[j] + " "
        j += 1

    docList = searchPhrase(index, set(searchList), searchList)

    resultList[statement] = docList
    if len(docList) > 0:
        print(statement, ":")
        print("    DocList: ", docList)

    j = 0
    while j < N:
        if numList[j] < len(forSearchList[j]) - 1:
            numList[j] += 1
            m = 0
            while m < j:
                numList[m] = 0
                m += 1

            break
        j += 1

    if j >= N:
        break

return resultList
```

7、同义词查询

7.1 概述

一个优秀的信息检索系统应该支持同义词查询，因为这能够返回更多满足用户查询需求的文档。同义词主要包括以下几类：

下面是一个例子：缅甸的英文名称是 Myanmar 和 Burma，这两个词就是同义词。当用户搜索“Myanmar”时，他的需求是查询跟缅甸相关的新闻，这时检索系统应该不仅要返回“Myanmar”的查询结果，还要返回“Burma”的查询结果，这样才能使得搜索结果更完整。在具体项目中可以利用同义词表来实现。

7.2 实现

利用 nltk 语言处理库获取单个单词的同义词列表（有可能是短语），随后对每个单词或者短语进行检索，获取文档集。

首先是获取单个单词的同义词列表：

```
def getSynonyms(index, word):
    stem = word.lower()
    result = [[word]]
    for synset in wordnet.synsets(stem):
        for lemma in synset.lemmas():
            if lemma.name() != stem:
                wordlist = lemma.name().split('_')
                result.append(wordlist)
    return result
```

对每个单词或短语进行检索以获取文档集：

```
def searchSynonymsWord(index, word):
    wordlist = getSynonyms(index, word)
    resultList = {}

    for phraseList in wordlist:
        # print(phraseList)
        wordset = set(phraseList)
        list = searchPhrase(index, wordset, phraseList)
        phrase = ''
        for w in phraseList:
            phrase += w + " "
        if len(list) > 0:
            print(phrase, ":")
            print("    ", list)
```

```
            resultList[phrase] = list

    return resultList
```

8、拼写校正

8.1 概述

拼写校正主要应用于容错式检索。容错式检索的处理对象是非精确的查询形式，即拼写上存在错误的查询。用户可能无意中会出现查询的拼写错误，或者用户在检索查询词时，并不清楚其在文档集中的正确拼写方式。

对于大多数拼写校正算法而言，存在以下两个基本的原则：（1）对于一个拼写错误的的查询，在其可能的正确拼写中，选择距离“最近”的一个。这就要求在查询之间有距离或者邻近度的概念。（2）当两个正确拼写查询邻近度相等（或相近）时，选择更常见的那个。

拼写校正主要包含两种方法：一种是词项独立的校正，另一种是上下文敏感的校正。在词项独立的校正方法中，不管查询中包含多少个查询词项，每次只考虑一个词项的校正。词项独立的拼写校正有两种主要的方法：编辑距离方法以及 k-gram 重合度方法。本项目中，我采用了编辑距离方法。

以下是编辑距离的定义。给定两个字符串 s_1 及 s_2，两者的编辑距离（edit distance）定义为将 s_1 转换成 s_2 的最小编辑操作（edit operation）数。通常，这些编辑操作包括： (i) 将一个字符插入字符串； (ii) 从字符串中删除一个字符； (iii) 将字符串中的一个字符替换成另外一个字符。

8.2 实现

下面具体介绍拼写校正在本项目中的具体实现方法。

总的来说，我们采用了以下公式进行计算：

$$\operatorname{argmax}_{c \in candidates} P(c|w) \quad \text{..... (1)}$$

其中 c 代表“编辑距离”为 0、1 或 2 的词，且属于给定的词典。

P(c)指的是每个词在字典中出现的频率，P(w)则是每一种拼写错误可能出现的概率，这里我们将其视为一个常数。

根据贝叶斯理论，我们计算的 P(c|w)就是在拼写错误的前提下得到词项 c 的概率，我们要做的是选出使得上述概率最高的词项 c。式子(1)与以下表达式等价：

$$\operatorname{argmax}_{c \in candidates} P(c)\, P(w|c) / P(w) \quad \text{..... (2)}$$

因为 P(w)对于每一个词项 c 来说都是一样的，因此可以将其去除，最终表达式为：

$$\operatorname{argmax}_{c \in candidates} P(c)\, P(w|c) \quad \text{..... (3)}$$

上式中 P(w|c)指的是给定词项 c，c 就是输入者想要的单词的概率。

由于没有错误模型数据，我们做出以下假设：

P(w|c0) >> P(w|c1) >> P(w|c2) .....(4)

其中 c0、c1、c2 分别表示“编辑距离”为 0、1、2 的词。由该式子可知“编辑距离”越小，优先级越高。

模型的工作流程如下：

（1）如果输入的词在词典里找到了原词，则返回；（2）从“编辑距离”为 1 的词中选出频率最高的返回；（3）从“编辑距离”为 2 的词中选出频率最高的返回。

调用的相关函数如下所示：

```
def correctSentence(input):
    #words = input.split(' ')
    res = []
    for word in input:
        #print(word)
        if word =='(' or word == ')':
            res.append(word)
        else:
            res.append(correction(word))
        # print(correction(word))
    return res
```

其中 correction 函数如下所示，该函数就是式子(3)的代码表达：

```
def correction(word):
    "Most probable spelling correction for word."
    return max(candidates(word), key=P)
```

P 函数如下：

```
def P(word, N=sum(WORDS.values())):
    "Probability of `word`."
    return WORDS[word] / N
```

candidates 函数如下所示。

```
def candidates(word):
    "Generate possible spelling corrections for word."
    return (known([word]) or known(edits1(word)) or known(edits2(word)) or [word])
```

known 函数会在词典中寻找指定词项，若找到则返回。因此上面 candidates 函数的逻辑就是按照“编辑距离”0、1、2 的顺序在词典中寻找，若在某一步能找到则直接返回；如果“编辑距离”0、1、2 均找不到，则直接返回输入词项。

下面说明一下 known、edits1 和 edits2 这三个函数。

（1）known:

```
def known(words):
    "The subset of `words` that appear in the dictionary of WORDS."
    return set(w for w in words if w in WORDS)
```

在词典中找到输入中包含的词项并返回。

（2）edits1：

```
def edits1(word):
    "All edits that are one edit away from `word`."
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
```

```
    inserts    = [L + c + R               for L, R in splits for c in letters]
    return set(deletes + transposes + replaces + inserts)
```

该函数给出了“编辑距离”为 1 的定义，可见包括了删除一个字符、交换两个连续字符的位置、替换一个字符和插入一个字符。

（3）edits2：

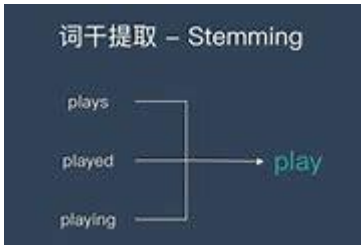
```
def edits2(word):
    "All edits that are two edits away from `word`."
    return (e2 for e1 in edits1(word) for e2 in edits1(e1))
```

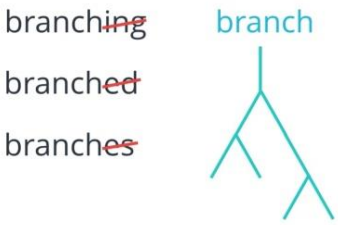
该函数给出了“编辑距离”为 2 的定义，可见就是做出了两次“编辑距离”为 1 的操作。

9、词干还原

9.1 概述

由于语法上的要求，文档中常常会使用词的不同形态，比如 organize、organizes 和 organizing。另外，语言中也存在大量意义相近的同源词，比如 democracy、democratic 和 democratization。在很多情况下，如果输入其中一个词能返回包含其同源词的文档，那么这样的搜素似乎非常有用。词干还原通常指的是去除单词两端词缀的启发式过程，其目的是为了减少词的屈折变化形式。下图是两个例子：





9.2 实现

本项目利用 python 中自然语言处理的库 nltk 来对文章中的单词进行词干还原。在词干还原的过程中还会去除无用的标点符号、排除空字符串等。

```
def lemmatize_sentence(sentence,forinput):
    res = []
    result = []
    lemmatizer = WordNetLemmatizer()
    for word, pos in pos_tag(word_tokenize(sentence)):
        wordnet_pos = get_wordnet_pos(pos) or wordnet.NOUN
        res.append(lemmatizer.lemmatize(word, pos=wordnet_pos))

    for word in res:
        #如果是 's 什么的，直接排除
        if word[0] is '\'':
            continue

        #去除标点符号
        if not forinput:
            for c in deleteSignal:
                word = word.replace(c,'')
```

```
        else:
            for c in deleteSignalForInput:
                word = word.replace(c,'')

        #排除空的字符串
        if len(word) is 0 or word[0] is '-':
            continue

        #如果分解的单词中有/,则将其中的每个单词添加到结果中
        if word.find('/') > 0:
            rs = word.split('/')
            for w in rs:
                w = getWord(w)
                result.append(w)
        else:
            word = getWord(word)
            result.append(word)

    return result
```

10、主函数逻辑

下面对该程序的主函数逻辑进行说明。

（1）定义命令行参数：

```
parser = argparse.ArgumentParser(description='SearchSystem')
parser.add_argument('-s', '--stemming', default='False', action='store_true', help='run
stemming or not')
parser.add_argument('-c', '--spelling_correcting', default='False', action='store_true',
                    help='correct spelling or not')
parser.add_argument('-k', '--topk', default=5, type=int, help='return top k results')
parser.add_argument('-i', '--indexing', default='False', action='store_true', help='build a new
index or not')

args = parser.parse_args()
print(args)
```

相关命令行参数的含义在前面已经介绍。

（2）对给定文档集建立倒排索引：

```
if args.indexing == True:
    print("establishing the INDEX...")
    establishIndex.createIndex(DIRECTNAME)
```

（3）加载全局的词项表 WORDLIST、全局倒排索引 INDEX 以及词干还原所依赖的语料库，同时指定文档集和它所在的路径：

```
print("getting word list...")
```

```
WORDLIST = getIndex.getWordList()
print("getting index...")
INDEX = getIndex.getIndex()
print("loading the wordnet...")
stemming.lemmatize_sentence("a", False)

PATH = tools.projectpath + DIRECTNAME
FILES = os.listdir(tools.reuterspath)
FILENUM = len(FILES)
```

（4）正式进入查询的循环：

```
LOOP = True
print("================Searching System================")

while LOOP:
    print("searching operation: ")
    print("[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit")
    print("your choice(int):")
    try:
        choice = int(input())
        if choice == 7:
            break
    except:
        print()
        continue
```

可以自行选择采用何种查询模式。

（5）检查输入：

```
if choice >= 1 and choice <= 6:
    print("input the query statement:")
    STATEMENT = input()
    INPUTWORDS = []
    if STATEMENT == "EXIT":
        break
```

检查输入的数字是否合法，若合法则进行相应的操作。

（6）全局查询排序：

```
if choice == 1:
    start = time.perf_counter()
    if (args.stemming == True):
        print("stemming...")
        INPUTWORDS = stemming.lemmatize_sentence(STATEMENT, True)
        print(INPUTWORDS)
    if (args.spelling_correcting == True):
```

```
        print("spelling correcting...")
        if args.stemming != True:
            INPUTWORDS.extend(STATEMENT.split())
        INPUTWORDS = spell.correctSentence(INPUTWORDS)
        print(INPUTWORDS)
    else:
        INPUTWORDS.extend(STATEMENT.split())
        print(INPUTWORDS)

    WORDSET = set(INPUTWORDS)

    DOCLIST = searchWord.searchWords(INDEX, WORDSET)
    SORTEDDOCLIST = sortDoc.sortScoreDocList(INDEX, FILENUM, WORDSET, DOCLIST)
    duration = (time.perf_counter() - start) * 1000  # ms
    rank = 0
    print("======================Overall Result=======================")
    for doc in SORTEDDOCLIST:
        rank += 1
        # print("doc ID: ", doc[1], " score: ", "%.4f" % doc[0])
        print(str(rank), " [", "%.3f]" % doc[0], " doc ID:", str(doc[1]))
    print("===========================================================")
    print("(took %.4f ms)" % duration)
```

这一部分会返回与查询语句匹配的全部结果，包含排名、得分和文档 ID。

（7）TOP K 查询：

```
elif choice == 2:
    start = time.perf_counter()
    if (args.stemming == True):
        print("stemming...")
        INPUTWORDS = stemming.lemmatize_sentence(STATEMENT, True)
        print(INPUTWORDS)
    if (args.spelling_correcting == True):
        print("spelling correcting...")
        if args.stemming != True:
            INPUTWORDS.extend(STATEMENT.split())
        INPUTWORDS = spell.correctSentence(INPUTWORDS)
        print(INPUTWORDS)
    else:
        INPUTWORDS.extend(STATEMENT.split())
        print(INPUTWORDS)

    WORDSET = set(INPUTWORDS)

    DOCLIST = searchWord.searchWords(INDEX, WORDSET)
```

```
    SORTEDDOCLIST = sortDoc.TopKScore(args.topk, INDEX, FILENUM, WORDSET, DOCLIST)
    duration = (time.perf_counter() - start) * 1000  # ms
    rank = 0
    print("======================TopK Search Result=======================")
    for doc in SORTEDDOCLIST:
        rank += 1
        # print("doc ID: ", doc[1], " score: ", "%.3f" % doc[0])
        print(str(rank), " [", "%.3f]" % doc[0], " doc ID:", str(doc[1]))
        file_path = tools.reuterspath + str(doc[1]) + ".html"
        head(file_path)
    print("================================================================")
    print("(took %.4f ms)" % duration)
```

这一部分会返回与查询语句匹配的得分 Top K 的结果，包含了排名、得分、文档 ID 以及文档摘要。

其中文档摘要输出的是文档的前 5 行内容：

```
def head(file_name, line_num=5):
    f = open(file_name, 'r')
    line_count = line_num
    for line in f.readlines()[:line_num]:
        if line_count == 1:
            print(line.strip('\n') + ' ... ')
        else:
            print(line.strip('\n'))
        line_count -= 1
    f.close()
    print(" ")
```

（8）布尔查询：

```
elif choice == 3:
    start = time.perf_counter()
    if (args.stemming == True):
        print("stemming...")
        INPUTWORDS = stemming.lemmatize_sentence(STATEMENT, True)
        print(INPUTWORDS)
    if (args.spelling_correcting == True):
        print("spelling correcting...")
        if args.stemming != True:
            INPUTWORDS.extend(STATEMENT.split())
        INPUTWORDS = spell.correctSentence(INPUTWORDS)
        print(INPUTWORDS)
    else:
        INPUTWORDS.extend(STATEMENT.split())
        print(INPUTWORDS)
```

```
    DOCLIST = BoolSearchDel.BoolSearch(INPUTWORDS, INDEX)
    duration = (time.perf_counter() - start) * 1000  # ms
    print(len(DOCLIST), "DOCs :")
    print(DOCLIST)
    print("(took %.4f ms)" % duration)
```

该部分会返回满足布尔表达式的全部文档 ID。

（9）短语查询：

```
elif choice == 4:
    start = time.perf_counter()
    if (args.stemming == True):
        print("stemming...")
        INPUTWORDS = stemming.lemmatize_sentence(STATEMENT, True)
        print(INPUTWORDS)
    if (args.spelling_correcting == True):
        print("spelling correcting...")
        if args.stemming != True:
            INPUTWORDS.extend(STATEMENT.split())
        INPUTWORDS = spell.correctSentence(INPUTWORDS)
        print(INPUTWORDS)
    else:
        INPUTWORDS.extend(STATEMENT.split())
        print(INPUTWORDS)

    WORDSET = set(INPUTWORDS)

    PHRASEDOCLIST = searchWord.searchPhrase(INDEX, WORDSET, INPUTWORDS)
    duration = (time.perf_counter() - start) * 1000  # ms
    if 0 == len(PHRASEDOCLIST):
        print("Doesn't find \"", INPUTWORDS, '"')
    else:
        for key in PHRASEDOCLIST:
            print('docID: ', key, "   num: ", len(PHRASEDOCLIST[key]))
            print('    location: ', PHRASEDOCLIST[key])
    print("(took %.4f ms)" % duration)
```

该部分会返回严格包含查询短语的文档 ID 以及短语位置。

（10）模糊（通配符）查询：

```
elif choice == 5:
    start = time.perf_counter()
    list = searchWord.wildcardSearch(STATEMENT, INDEX, WORDLIST)
    duration = (time.perf_counter() - start) * 1000  # ms
    print("(took %.4f ms)" % duration)
```

该部分会返回所有与通配符匹配的词项（或短语）所在的文档 ID 以及词项（或短语）

所在位置。

（11）同义词查询：

```
elif choice == 6:
    start = time.perf_counter()
    if (args.stemming == True):
        print("stemming...")
        INPUTWORDS = stemming.lemmatize_sentence(STATEMENT, True)
        print(INPUTWORDS)
    if (args.spelling_correcting == True):
        print("spelling correcting...")
        if args.stemming != True:
            INPUTWORDS.extend(STATEMENT.split())
        INPUTWORDS = spell.correctSentence(INPUTWORDS)
        print(INPUTWORDS)
    else:
        INPUTWORDS.extend(STATEMENT.split())
        print(INPUTWORDS)

    WORDSET = set(INPUTWORDS)

    resultlist = searchWord.searchSynonymsWord(INDEX, INPUTWORDS[0])
    duration = (time.perf_counter() - start) * 1000  # ms
    print("(took %.4f ms)" % duration)
```

该部分会首先找到所有输入词项的同义词，然后返回所有包含这些词项的文档 ID 以及词项所在位置。

这里有几个地方需要说明一下：

（1）overall 查询和 Top K 查询属于自由文本查询，这种查询可以表示为一系列词的集合，词之间没有任何操作符连接。在这种模型下可以对文档进行评分和排序。而由于布尔查询、短语查询以及通配符查询的处理方式与自由文本查询有所不同，所以无法计算得分。

（2）在本项目中，我还调用了 python 中的 time 库来进行计时，以评估不同形式查询的用时。具体的，我使用了 time.perf_counter()来计算程序的运算时间。之所以使用 time.perf_counter()而不是 time.time()或者 time.clock()，是因为 time.perf_counter()返回性能计数器的值（以微秒为单位），具有最高可用分辨率，适用于测量短持续时间。

【实验结果分析】：

（1）全局查询排序：

以下是一例，在终端输入“retrieval”。

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
1
input the query statement:
retrieval
```

```
['retrieval']
======================Overall Result======================
1  [ 5.341]  doc ID: 733
2  [ 3.334]  doc ID: 21133
3  [ 3.334]  doc ID: 1584
4  [ 3.334]  doc ID: 706
5  [ 3.334]  doc ID: 424
==========================================================
(took 0.9451 ms)
```

返回全部相关的新闻文档，这些文档按照分数排序。给出了文档 ID，但并未给出摘要（为了避免当文本数量很大时输出内容过多），若要查看摘要则需选择 TOP K 查询。

（2）TOP K 查询：

以下是一例，在终端输入“new york san francisco”。

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
2
input the query statement:
new york san francisco
['new', 'york', 'san', 'francisco']
```

结果如下：

```
======================TopK Search Result======================
1  [ 9.410]  doc ID: 19828
SAN FRANCISCO, NOT REGION, HURT BY RESTRUCTURING
  Corporate mergers and acquisitions
  in and around San Francisco over the past seven years have had
  only a modest effect on the metropolitan area's economy, a
  leading business-backed organization said. ...

2  [ 7.405]  doc ID: 11863
U.S. SUPREME COURT ALLOWS DELTA-WESTERN MERGER
  U.S. Supreme Court Justice Sandra Day
  O'Connor early this morning lifted an Appeals Court injunction
  blocking the planned merger of Delta &lt;DAL> Airlines
  Inc and Western Airlines &lt;WAL>, the Court said. ...

3  [ 5.974]  doc ID: 14888
ANHEUSER-BUSCH JOINS BID FOR SAN MIGUEL
  Anheuser-Busch Companies Inc &lt;BUD.N> has
  joined several other foreign bidders for sequestered shares of
  the Philippines' largest food and beverage maker San Miguel
  Corp &lt;SANM.MN>, the head of a government panel which controls ...

4  [ 5.887]  doc ID: 13877
BALL &lt;BLL> TO SUPPLY PENNY BLANKS TO MINTS
  Ball Corp said it was awarded a
  one-year 12,750,000 dlr contract to supply copper-plated zinc
  penny blanks to the U.S. mints in Philadelphia and Denver.
      The new contract, effective in June, calls for shipping ...
```

```
5  [ 5.887]  doc ID: 15240
HARTMARX &lt;HMX> TARGETS EARNINGS GROWTH
  Hartmarx Corp, following a year of
  restructuring, continues to target record earnings for fiscal
  1987, Chairman John Meinert told the annual meeting.
      Meinert reiterated an earlier comment that earnings for the ...

============================================================
(took 9.7873 ms)
```

可见成功返回了 top k 个结果（此处 k 默认是 5），且按照分数从高到低排序，提供了排名、分数、文档 ID 以及摘要。

另外，我们可以在命令行中指定 k，下面是一例。

```
E:\大三下\信息检索\大作业\SearchingSystem\SearchSystem>python main.py -k 3
projectpath: E:\大三下\信息检索\大作业\SearchingSystem\SearchSystem\
Reuters path E:\大三下\信息检索\大作业\SearchingSystem\Reuters\
getting file list...
loading spell words
Namespace(indexing='False', spelling_correcting='False', stemming='False', topk=3)
getting word list...
getting index...
loading the wordnet...
================Searching System================
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
```

将 k 指定为 3 后，Top K 查询将只返回分数最高的三条结果。

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
2
input the query statement:
new york san francisco
['new', 'york', 'san', 'francisco']
====================TopK Search Result====================
1  [ 9.410]  doc ID: 19828
SAN FRANCISCO, NOT REGION, HURT BY RESTRUCTURING
  Corporate mergers and acquisitions
  in and around San Francisco over the past seven years have had
  only a modest effect on the metropolitan area's economy, a
  leading business-backed organization said. ...

2  [ 7.405]  doc ID: 11863
U.S. SUPREME COURT ALLOWS DELTA-WESTERN MERGER
  U.S. Supreme Court Justice Sandra Day
  O'Connor early this morning lifted an Appeals Court injunction
  blocking the planned merger of Delta &lt;DAL> Airlines
  Inc and Western Airlines &lt;WAL>, the Court said. ...

3  [ 5.974]  doc ID: 14888
ANHEUSER-BUSCH JOINS BID FOR SAN MIGUEL
  Anheuser-Busch Companies Inc &lt;BUD.N> has
  joined several other foreign bidders for sequestered shares of
  the Philippines' largest food and beverage maker San Miguel
  Corp &lt;SANM.MN>, the head of a government panel which controls ...

============================================================
(took 8.7316 ms)
```

（3）布尔查询（注意符号的优先级关系是 NOT>AND>OR，且支持括号）：

选择进行布尔查询。下面以“apple”、“computer”和“phone”这三个词项为例进行展示。

[1]首先是“AND”查询，输入“apple AND computer”，结果如下：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
3
input the query statement:
apple AND computer
['apple', 'AND', 'computer']
2 DOCs :
[361, 19828]
(took 1.7958 ms)
```

可见找到两篇文档，可以查看一下它们的内容：

361.html:

```
SCIENTIFIC MICRO SYSTEMS <SMSI> ACQUIRES SUPERMAC
  Scientific Micro Systems Inc said it
  has acquired Supermac Technology, a rapidly growing supplier of
  enhancement products and disc drive subsystems for the Apple
  personal computer market.
      Scientific Micro said it acquired all the common stock of
  Supermac in exchange for 1.05 mln shares of its own common
  stock. The stock closed at 5.50 dlrs bid on Friday.
      Supermac, a privately held firm based in Mountain View,
  California, as is Scientific Micro, reported a net profit of
  300,000 dlrs on revenue of 9.5 mln dlrs in fiscal 1986. It
  expects its revenue to approximately double in 1987.
```

19828.html:

```
  Francisco, he was the envy of many chief executives in other
  metropolitan area.
      "It's a great place to run a business," he added, "but it's
  a hell of a place to do business with government."
      The study concluded that Chevron, which mergered with Gulf
  Oil in 1984, would benefit in the long run from the
  restructuring activity.
      Among the corporate headquarters lost during the period
  were Crown Zellerbach, Memorex, Southern Pacific, Castle &
  Cooke and Rolm.
      Companies that grew enough during the period to make the
  Fortune 500 list included Apple Computer, Pacific Telesis,
  McKesson, Tandem Computer, U.S. Leasing and Amfac.
      The study by the management consulting firm said that,
  partly due to corporate restructuring, the rate of job growth
  in San Francisco has slowed since 1980.
      In addition, it said that the loss of corporate leadership
  had adversely affected some of the Bay Area's civic and
  charitable activities.
```

[2]“NOT”查询。输入“apple AND NOT computer”：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
3
input the query statement:
apple AND NOT computer
['apple', 'AND', 'NOT', 'computer']
5 DOCs :
[1361, 8692, 8699, 16094, 16357]
(took 4.6634 ms)
```

返回的文档 ID 包括 1361、8692、8699、16094 和 16357。

为了检验以上 AND 和 NOT 的效果，下面直接输入“apple”来查看返回内容：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
3
input the query statement:
apple
['apple']
7 DOCs :
[361, 1361, 8692, 8699, 16094, 16357, 19828]
(took 0.4171 ms)
```

可见返回的文档 ID 包括：361、1361、8692、8699、16094、16357 和 19828。

结果对比如下：

输入查询语句	返回文档 ID
apple AND computer	361,19828
apple AND NOT computer	1361,8692,8699,16094,16357
apple	361,1361,8692,8699,16094,16357,19828

可见“apple”查询返回的文档包括了“apple AND computer”查询和“apple AND NOT computer”查询的结果，检验成功。

[3]“OR”查询。输入“apple AND computer OR phone”：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
3
input the query statement:
apple AND computer OR phone
['apple', 'AND', 'computer', 'OR', 'phone']
10 DOCs :
[361, 2792, 4503, 4577, 6326, 8569, 12195, 16513, 19828, 21371]
(took 4.0589 ms)
```

返回的文档 ID 包括 361、2792、4503 等。

为了检验以上 AND 和 OR 的效果，下面直接输入“phone”来查看返回内容：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
3
input the query statement:
phone
['phone']
8 DOCs :
[2792, 4503, 4577, 6326, 8569, 12195, 16513, 21371]
(took 3.0660 ms)
```

结果对比如下：

输入查询语句	返回文档 ID
apple AND computer	361,19828
apple AND computer OR phone	361, 2792, 4503, 4577, 6326, 8569, 12195, 16513, 19828, 21371
phone	2792, 4503, 4577, 6326, 8569, 12195, 16513, 21371

可见“apple AND computer OR phone”查询返回的文档包括了“apple AND computer”查询和“phone”查询的结果，检验成功。

[4]符号优先级测试及带括号的布尔查询：

下面结合几个例子来测试符号的优先级以及布尔表达式中括号的作用：

输入查询语句	返回文档 ID
apple AND computer	361,19828
apple AND NOT computer	1361,8692,8699,16094,16357
NOT computer AND apple	1361,8692,8699,16094,16357
apple AND computer OR phone	361,2792,4503,4577,6326,8569,12195,16513,19828,21371
phone OR computer AND apple	361,2792,4503,4577,6326,8569,12195,16513,19828,21371
(phone OR computer) AND apple	361,19828

从上面的结果中不难看出符号的优先级是：NOT>AND>OR。如果带括号的话则优先处理括号中的表达式。

（4）短语查询：

输入“apple computer”进行查询，返回结果如下：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
4
input the query statement:
apple computer
['apple', 'computer']
docID:  19828    num:  1
    location:  [359]
(took 0.4664 ms)
```

仅返回了 ID 为 19828 的文档，其内容如下：

```
Francisco, he was the envy of many chief executives in other
metropolitan area.
    "It's a great place to run a business," he added, "but it's
a hell of a place to do business with government."
    The study concluded that Chevron, which mergered with Gulf
Oil in 1984, would benefit in the long run from the
restructuring activity.
    Among the corporate headquarters lost during the period
were Crown Zellerbach, Memorex, Southern Pacific, Castle &
Cooke and Rolm.
    Companies that grew enough during the period to make the
Fortune 500 list included Apple Computer, Pacific Telesis,
McKesson, Tandem Computer, U.S. Leasing and Amfac.
    The study by the management consulting firm said that,
partly due to corporate restructuring, the rate of job growth
in San Francisco has slowed since 1980.
    In addition, it said that the loss of corporate leadership
had adversely affected some of the Bay Area's civic and
charitable activities.
```

可见包含了“apple computer”短语。

这里与前面的布尔查询做个对比：

查询类型与输入	返回文档 ID
布尔查询：apple AND computer	361,19828
短语查询：apple computer	19828

短语查询并没有返回布尔查询中的 ID 为 361 的文档，这是因为 ID 为 361 的文档虽然同时包含了 apple 和 computer 这两个词，但并没有组成词组“apple computer”。从这个例子可以看出短语查询在“AND”布尔查询的基础上更进一步，不仅要求同时出现，还要求满足词组关系（即位置关系）。

（5）通配符查询：

在通配符查询的语句中支持“?”、“*”等符号，其中“?”代表任意一个字符，“*”则可以代替零个、单个或多个字符。通配符查询是具有实际意义的，当我们在进行信息的搜索时，如果不知道真正字符（比如忘记了某个词如何拼写）或者懒得输入完整名字时，常常使用通配符代替一个或多个真正的字符。

[1]“?”查询。

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
5
input the query statement:
appl?
APPLE  :
    DocList:  {20340: [0]}
APPLO  :
    DocList:  {20340: [5]}
APPLY  :
    DocList:  {2010: [2]}
apple  :
    DocList:  {361: [30], 1361: [129, 150, 163], 8692: [48], 8699: [48],
apply  :
    DocList:  {798: [71], 874: [79], 894: [65], 925: [368], 978: [61], 13
 [30, 46], 3847: [20], 4071: [366], 4145: [39], 4160: [103], 4640: [10],
```

我们输入“appl?”，最终返回了“apple”、“apply”等的搜索结果。

[2]“*”查询。

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
5
input the query statement:
appl*
APPLAUD  :
    DocList:  {11213: [1]}
APPLAUDS  :
    DocList:  {20045: [1]}
APPLE  :
    DocList:  {20340: [0]}
APPLIANCE  :
    DocList:  {7447: [2]}
APPLIANCES  :
    DocList:  {2660: [2]}
APPLICATION  :
    DocList:  {2939: [6, 16], 2940: [4, 12], 2948: [8], 5664: [2, 14], 11352: [6]}
```

我们输入“appl*”，最终返回了“applaud”、“apple”、“appliance”、“application”等的搜索结果。

除此之外，我们还可以借助通配符实现更丰富的查询功能，下面是一个例子：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
5
input the query statement:
appl? com*
apple computer  :
    DocList:  {'19828': [359]}
(took 126.4654 ms)
```

我们输入“appl? com*”，最终返回了“apple computer”短语的搜索结果。可见，通过巧妙地输入包含通配符的查询语句，我们也能实现短语查询的功能。

（6）同义词查询：

下面是一个例子：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
6
input the query statement:
happy
['happy']
happy  :
     {342: [438], 1312: [762], 1858: [162], 1967: [270], 3056: [177], 5523: [96], 8
130], 14890: [344], 14931: [130], 16784: [87], 17986: [515], 18029: [15], 20005: [5
glad  :
     {11390: [116]}
(took 22.4098 ms)
```

我们输入的查询语句是“happy”，检索系统不仅返回了“happy”的结果，还返回了“happy”的同义词“glad”的结果。

（7）其他结果：

主程序的使用方法如下图所示：

```
usage: main.py [-h] [-s] [-c] [-k TOPK] [-i]

SearchSystem

optional arguments:
  -h, --help            show this help message and exit
  -s, --stemming        run stemming or not
  -c, --spelling_correcting
                        correct spelling or not
  -k TOPK, --topk TOPK  return top k results
  -i, --indexing        build a new index or not
```

其包含了“-h”、“-s”、“-c”、“-k”、“-i”这几个参数。在前面的结果展示中我们仅仅利用了“-k”参数，其他的参数尚未涉及，下面将分别展示：

[1]在命令行输入“-s”进行 stemming（即词干提取）操作：

```
E:\大三下\信息检索\大作业\SearchingSystem\SearchSystem>python main.py -s
projectpath: E:\大三下\信息检索\大作业\SearchingSystem\SearchSystem\
Reuters path E:\大三下\信息检索\大作业\SearchingSystem\Reuters\
getting file list...
loading spell words
Namespace(indexing='False', spelling_correcting='False', stemming=True, topk=5)
getting word list...
getting index...
loading the wordnet...
================Searching System================
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
```

查询语句输入“apples”，会自动地提取出词干“apple”：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
2
input the query statement:
apples
stemming...
['apple']
['apple', 'apples']
======================TopK Search Result=====================
1  [ 4.709]  doc ID: 1361
FCOJ SUPPLIES SIGNIFICANTLY ABOVE YEAR AGO-USDA
  Total supply of frozen concentrated
  orange juice (FCOJ) in 1986/87 is expected to be significantly
  above year-earlier levels, even with carry-in stocks well below
  the previous season, the U.S. Agriculture Department said. ...
```

[2]在命令行输入“-c”进行 spelling_correcting（即拼写校正）操作：

```
E:\大三下\信息检索\大作业\SearchingSystem\SearchSystem>python main.py -c
projectpath: E:\大三下\信息检索\大作业\SearchingSystem\SearchSystem\
Reuters path E:\大三下\信息检索\大作业\SearchingSystem\Reuters\
getting file list...
loading spell words
Namespace(indexing='False', spelling_correcting=True, stemming='False', topk=5)
getting word list...
getting index...
loading the wordnet...
================Searching System================
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
```

输入“hapy”，系统会校正为“happy”：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
2
input the query statement:
hapy
spelling correcting...
['happy']
======================TopK Search Result=====================
1  [ 2.635]  doc ID: 342
JAPAN, U.S. SET TO BEGIN HIGH-LEVEL TRADE TALKS
  Japan and the U.S. Kick off top-level
  trade talks tomorrow amid signs officials from both sides are
  growing increasingly irritated with each other.
      The talks, held annually at sub-cabinet level to review the ...

2  [ 2.635]  doc ID: 21565
NEW ZEALAND IMPOSES SANCTIONS AGAINST FIJI
  New Zealand has imposed sanctions
  against Fiji in response to that country's change of status to
  a republic, acting prime minister Geoffrey Palmer said.
      The sanctions will end all military cooperation and cut ...
```

这里需要注意的是，拼写校正的功能是有限的，有时候未必会纠正得到我们想要搜索的

词项，下面是一个例子：

```
searching operation:
[1] Overall [2]TOP K [3]BOOL [4]Phrase [5]wildcard [6]synonyms [7]exit
your choice(int):
2
input the query statement:
chian
spelling correcting...
['chain']
=====================TopK Search Result=====================
1  [ 4.272]  doc ID: 1269
F.W. WOOLWORTH'S &lt;Z> 1986 PROFITS RISE 21 PCT
  The specialty retailing area continues
  to pay off for F.W. Woolworth Co, once known only as a five and
  dime store chain, which said its 1986 income rose 21 pct.
      It was the fourth consecutive year of profit increases for ...

2  [ 4.028]  doc ID: 8441
TALKING POINT/WENDY'S INTERNATIONAL &lt;WEN>
  Takeover speculation buoyed Wendy's
  International Inc's stock, even after Coca Cola Co took the
  fizz out of market rumors by denying it was an interested
  suitor. ...
```

我们预期的查询语句是“china”，但不小心拼成了“chian”，最终校正后的词项是“chain”。可见，拼写校正未必能达到我们的预期（这也是正常现象），因此我们最好在搜索时就输入正确的词项，避免得不到想要的查询结果。

[3]在命令行输入“-i”进行构建索引操作（这一步一般在数据收集完毕、开始查询之前进行）：

```
E:\大三下\信息检索\大作业\SearchingSystem\SearchSystem>python main.py -i
projectpath: E:\大三下\信息检索\大作业\SearchingSystem\SearchSystem\
Reuters path E:\大三下\信息检索\大作业\SearchingSystem\Reuters\
getting file list...
loading spell words
Namespace(indexing=True, spelling_correcting='False', stemming='False', topk=5)
establishing the INDEX...
analyzing file:  1.html
analyzing file:  10.html
analyzing file:  100.html
analyzing file:  1000.html
analyzing file:  10000.html
analyzing file:  10002.html
analyzing file:  10005.html
analyzing file:  10008.html
analyzing file:  10011.html
analyzing file:  10014.html
analyzing file:  10015.html
analyzing file:  10018.html
analyzing file:  10023.html
analyzing file:  10025.html
```

在构建索引时，系统会分析每一篇新闻文档，最终形成一个完整的 word_list 和带位置

信息的倒排索引。通过对本项目中的新闻文档（包含 10788 篇）进行索引构建，发现能在 10 分钟左右构建完毕，可见效率比较高；最终形成的倒排索引大小为 14.2 MB，所占空间不大，可直接放入内存中运行。

（8）结果对比：

[1]不同查询的时间对比

本项目实现了许多不同类型的查询，具体包括全局查询、TOP K 查询、布尔查询、短语查询、通配符查询和同义词查询。其中，全局查询和 TOP K 查询均是基于向量空间模型的查询，它们的不同点在于前者在得到文档集后使用 python 内置的 sorted 函数对分数进行排序，而后者则使用自定义的堆排序进行排序。这里尝试查询了几个词项来比较它们的处理时间：

搜索词项	全局查询用时	TOP K 查询用时
hurricane	0.8753 ms	1.0813 ms
apple	1.1829 ms	1.3138 ms
apple computer	1.5549 ms	2.8503 ms
new york	6.4694 ms	8.2851 ms

可见对于同一个搜索词项，全局查询所花时间往往要略短于 TOP K 查询，可能是因为不同算法实现的效率问题。当然，这种差距并不大，而且也不是绝对的，有时候后者的查询时间可能会短于前者。

而对于布尔查询、短语查询、通配符查询和同义词查询而言，因为它们采用的具体方法不同，所以没有互相对比时间的意义，下面结合几个例子来展示它们各自的运行时间。

- 布尔查询：

搜索词项	查询用时
apple AND computer	1.0348 ms
apple AND NOT computer	4.0567 ms
apple OR computer	1.0746 ms
apple	1.4716 ms
NOT apple	4.2950 ms

发现 AND 查询或者 OR 查询所花的时间与普通查询相近，而 NOT 查询则要明显花费更多时间。

- 短语查询：

搜索词项	查询用时
apple computer	0.5071 ms

US military	2.3487 ms
new york	3.0364 ms

短语查询的用时一般比较短（特别是与 TOP K 查询对比时），这归功于带位置信息的倒排索引的实现，它使得我们可以快捷地定位到短语位置。当然，不同短语的查询时间是不同的，原因之一是包含不同短语的新闻文档数不同。

- 通配符查询：

搜索词项	查询用时
appl?	21.3314 ms
appl*	29.7473 ms
appl? compute?	40.2600 ms
appl? com*	137.0765 ms
app* com*	2546.9119 ms

我们可以明显看到，通配符查询所花时间要远远超过前面几种类型的查询。原因也是显而易见的，因为通配符查询需要利用正则表达式来找到所有匹配项。另外，在三种类型的通配符查询中，包含“？”的单个词项往往用时最短，包含“*”的单个词项用时次之，而如果是包含通配符的短语查询，则用时会远超单个词项，这是因为需要对两个词项进行通配符的匹配，还要查找满足位置关系的短语。再者，使用“*”通配符的查询时间也会超过“？”通配符，这是因为前者需要匹配 0 个、1 个或多个字符，而后者仅需匹配一个字符。

- 同义词查询：

搜索词项	同义词查询用时	TOP K 查询用时
hurricane	2.0303 ms	1.0813 ms
apple	2.8384 ms	1.3138 ms
US	111.6125 ms	9.8122 ms

由以上的结果对比可知，对于同一词项的同义词查询用时会超过 TOP K 查询，这是因为在同义词查询中我们不仅要检索包含输入词项的新闻文本，还要检索包含该词项同义词的新闻文本。另一方面，不同词项的同义词查询所花时间也是相差巨大，这是因为包含不同搜索词项的文本数量以及词项的同义词数量不同。

[2]tf-idf 和 wf-idf 方法的对比评估

本项目中实现了两种评分函数，一种是 tf-idf，另一种是其亚线性变换 wf-idf。在 TOP K 查询中，我们首先会通过查询向量计算出文档的评分，然后根据得分排序并返回结果，因此评分函数对于检索结果起着极其重要的作用，在某种程度上不同的评分函数也代表了不同的检索策略。下面以“US military”为例分别展示使用这两种评分函数的检索结果，其中标红的是搜索无关项。

➢ 使用 tf-idf 评分函数：

返回新闻文档 ID（按得分从高到低）	主要内容
20632	美国对伊朗的军事行动
7135	美国与部分亚洲国家的贸易问题
4665	美日芯片贸易问题
1579	美国的咖啡配额谈判失败
14858	美日农产品贸易

由以上结果可知，使用 tf-idf 的搜索结果不太理想，在本例的 TOP 5 查询中除第一个外都是无关的新闻，准确率仅为 20%。通过观察和分析可知，在返回的后四个新闻报道中包含了大量的“美国”字样，因此被错误地返回。

➢ 使用 wf-idf 评分函数：

返回新闻文档 ID（按得分从高到低）	主要内容
20632	美国对伊朗的军事行动
18231	美国保护其在中东湾的军队
21006	美国对伊朗在海湾的石油平台发动袭击
10395	希腊与土耳其的军事冲突风险
21013	美国在伊朗的军事行动陷入僵局

由以上结果可知，使用 wf-idf 的搜索结果较为准确，在本例的 TOP 5 查询中准确率达到 80%，仅有一个无关新闻。该无关新闻讲的是希腊与土耳其的军事冲突风险，其中出现了较多“军事”（即 military）的字眼，因此被错误地返回。

通过以上的对比，我们发现 wf-idf 的效果比 tf-idf 要好很多，究其原因，是因为文档中某一词项的频率对评分的影响并不呈线性关系，也即一篇文章中单词出现 n 次不代表其权重扩大 n 倍。在 wf-idf 中通过 log 计算削弱了词项频率的影响，即对 tf 做了一种亚线性尺度变换，这就使得文档评分的计算变得更为合理。

【实验总结】：

本项目实现了一个基于带位置信息倒排索引的多功能新闻信息检索系统。检索对象是一系列路透社（Reuters）的新闻报道文本，通过键入查询词项可以返回相关的新闻，在某种意义上也可以将其视为一个关注特定新闻媒体动向的新闻检索系统。在技术层面，我构建了带位置信息的倒排索引来快速定位所需的新闻文档，之所以要加上位置信息是为了支持短语查询；最终实现了 TOP K 查询、布尔查询、短语查询、通配符查询、同义词查询等检索功能，同时也实现了词干还原和拼写校正，以此来提高检索效果。

在设计好项目各部分的方案和代码后，我从多个方面对检索系统进行了测试。首先通过

一些查询词项来测试了 TOP K 查询，并观察 K 变化后的结果变化；接着测试了布尔查询，通过查看返回的文档来检验是否满足布尔表达式的逻辑；然后测试了短语查询，查看返回的文档中是否包含了准确的短语；接着测试了通配符查询和同义词查询，查看检索结果是否能按照预期匹配到正确的词项。除此之外，我还测试了词干提取和拼写校正在该检索系统中的应用，发现在一定程度上改善了检索的效果。另外，我也尝试了对本项目中的新闻文档（包含 10788 篇）构建倒排索引，发现能在 10 分钟左右构建完毕，可见效率较高；最终形成的倒排索引大小为 14.2 MB，所占空间不大，可直接放入内存中运行。

在完成以上测试后，我又进一步测试了不同查询的响应时间，并结合实际和具体代码逻辑给出了合理的解释。最后，我对本项目中实现的两种评分函数（即 tf-idf 和 wf-idf）进行了对比评估，通过“US military”的检索结果，评估了两种方法的检索准确率，得出 wf-idf 效果更好的结论并给出了合理的解释。

最后谈一下我对这个项目的进一步展望。首先，为了更好地对信息检索系统进行评价，往往需要一些标准测试集，测试集中包含三个部分：一个文档集、一组用于测试的信息需求集合（信息需求可以表示成查询）以及一组相关性判定结果（对每个查询-文档对而言，通常会赋予一个二值判断结果，即相关或不相关）。在给定测试集的情况下，我们可以使用精确率（返回的结果中相关文档所占的比例）、召回率（返回的相关文档占所有相关文档的比例）以及 F1 值（精确率和召回率的调和平均值）来评估检索结果，还可以使用 MAP（平均正确率均值）等指标来评价搜索引擎输出的有序检索结果。其次，本系统尚未实现 GUI 界面，只实现了命令行的查询功能，后面可以进一步设计出一个图形交互界面，方便用户操作。

评语及评分（指导教师）

【评语】：

该实验报告设计并实现了基于 TF-IDF 实现的文档检索系统，并在一系列路透社（Reuters）的新闻报道文本数据集上进行了定性评价，基本达到设计目标。报告实现了包括布尔查询在内的多种查询方式且模块设计清晰，虽然有对查询结果的定量评价，但缺少了评价算法的实现与同类算法的比较。

评分：87

日期：2021-6-26