html 文件检索系统开发文档

精 82 张翀 2018010625

1. 实现功能列表

a) 基础项：分词算法的实现；倒排索引的建立；带有 skip pointer 的多元 AND queries 实现，任意个数单词的 AND、OR、NOT 语句查询

b) 加分项：短语查询的实现；支持通配符的单词和短语查询

c) 辅助项：搜索引擎 GUI 界面

2. 实验环境与类库

a) 实验环境：python3.5

b) 第三方类库：nltk，用于分词。

c) python 标准库：os，re，itertools，functools，time，tkinter

3. 功能实现原理

a) 分词算法

就分词而言，借助了外部分词工具 nltk。假设已有段落文本 T，标点符号集 P，那么首先将 T 中的换行符换作空格，再用 nltk.word_tokenize()对 T 进行分词，即可获得 T 的含标点的分词结果 T'；去除 T'中也在 P 中的元素（即取 T'-P），即可得到需要的分词结果。

b) 倒排索引

首先需要得到每一个 html 文件中的有效文本部分，即标题和摘要部分。CACM 的每个文件都具有统一的格式，即:

<html>\n<pre>\n\n [标题] [摘要] CACM [月份], [年份] [其他内容] </pre>\n</html>

仅有其中的一篇没有 CACM 四个字母，进行特殊化处理。因此，使用 re.findall 便可得到每一个文件中的标题+摘要内容文本。

之后，对每一个文本进行分词，对所有文本的分词结果求交集(去重)，得到总词集 W。对于 W 中的每一个单词 w，建立哈希映射 H（使用字典即可）使其对应到一个空字典 H[w]，H[w]中的键为文本编号，值为 w 在文本中的出现位置列表。简言之，此处建立了一个 {词：{文档：[位置] }}格式的嵌套字典，以便后续的短语查询。

在此之后，对于文件集 D 中的每一个文件 d，设其分词结果为 t。对于 t 中的每一个词 w，H[w]添加键 d(若有则不重复添加)，而 H[w][d]则添加新元素（w 在 d 中的位置）。这样就生成了带有位置信息的倒排索引。至于普通的倒排索引，不添加位置信息即可；H[w]还可以使用列表的数据结构，且这样的话鉴于遍历顺序，这些列表都是有序的。

c) 多元 AND queries 与 skip pointer

首先解决二元的 AND queries。对于词 a 与 b，利用之前建立的字典（哈希）可以快速得到出现 a 和 b 出现的文本序号集D_a和D_b，并使用第三节课 PPT 中的算法即可求交集。PPT 的第六页为无 skip pointer 的算法，第二十一页为有 skip pointer 的算法。拥有 skip pointer 的文本序号，以D_a为例，可用如下方式得到：

Algorithm 1

$step \leftarrow int(\sqrt{length(D_a)})$
for $i \in range(length(D_a))$ **do**
    if $i\%step - 0$ & $i + step < length(D_a)$ **then**
        $skip(D_a(i)) \leftarrow D_a(i + skip)$
    else
        $D_a(i)$ *has not skip pointer*

至于多元 AND queries，则只需要不断取出其中的两个，用其 AND query 的结果替代他们即可。同时，如果对象中有一个不是单词而是文本序号列表，那么对应的文本序号集就是它本身，无需哈希。

d) 短语查询

本项目的短语查询主要基于位置索引，而之前的带有位置信息的倒排索引正是为此。（参考 PPT）。

假设短语 p 由单词 p[0],p[1],...,p[n-1]组成，那么首先利用多元 AND queries 得到同时包含这些单词的文档 d[0],d[1],...,d[k-1]。对于其中任意一个文档 d[i]，各单词有不同的出现位置列表，将这些列表作笛卡尔积，即可得到所有这些词的出现位置的组合。在这些组合中，只要能找到一个组合是公差为 1 的等差数列，那么短语 p 就在文档 d[i]中出现。由此就实现了短语查询。

e) 通配符查询

本项目支持任意多次将单词中的任意长的一段替换为"*"进行查询，主要基于 bigram_index（参考 PPT）。至于用"?"替换单个字符的查询，由于原理一样，只需要限制每一个位置的长度为 1，因此便略过不表。本部分应首先实现从通配符到可取词的对应。

为了实现 bigram_index，以词汇库为源提取了 bigram 集（即['$o','on','ne',...]形式的无重集），并建立了类似与词-文章的倒排索引（无位置信息）。这样，以 a*bc 为例，就只需要进行对['$a','bc','c$']的 and query 即可，最后的得到的词再进行检查。bigram-word 的 and queries 与词汇-文档的 and queries 实现相似，不赘述。

接下来考虑更复杂的情况，即有多个星号。定义第一个星号左边部分为<left>，最后一个星号右边部分为<right>。以下分四种情况考虑：

1）<left>和<right>的长度都不为 0，则将他们都拆分成 bigram。以 left**right 为例，则左边拆分为['$l','le','ef','ft']，右边拆分为['ri','ig','gh','ht','t$']，对这两个集合的并集作 and query，得到词集 Q。Q 中可能有不满足通配符表达式的词，但是 Q 中词的总数不会太多，因此只要用表达式进行一次筛选即可。之前说到的简单情况是可以并入这种情况的。

2）<left>和<right>长度都为 0。分以下几种情况：

*2a）*字符串总长度大于等于 4

如果去掉左右的星号后，里面没有星号，那么将里面的部分拆分成不带开头和结尾的 bigram 进行 and query。如果里面还有星号，那么可能匹配结果较多，先取所有词集，再进行检查筛选。

*2b）*字符串总长度为 3

那么其必然是"***"或是"*a*"，前者返回所有词集，后者返回所有词集再作筛选即可。

*2c）*字符串长度为 1 或 2

那么必然是"**"或"*"，返回所有词集即可。

3）<left>或<right>中的一个长度为零。取不为零的那个，展开成 bigram，进行 and query 搜索后，对得到的词集进行筛选。

得到可取词之后，可用通过倒排索引的哈希直接得到对应的文档集，这些文档集求并即可。

f) 带通配符的短语查询

对于短语中每一个词，其根据之前的方法得到多个可能对应的词，这些词组成一个列表。将这些列表作笛卡尔积，即可得到可能对应的短语。对每个这样的短语作短语查询，最后的文档序号集求并集即可。

g) 任意个数单词的 AND、OR、NOT 语句查询

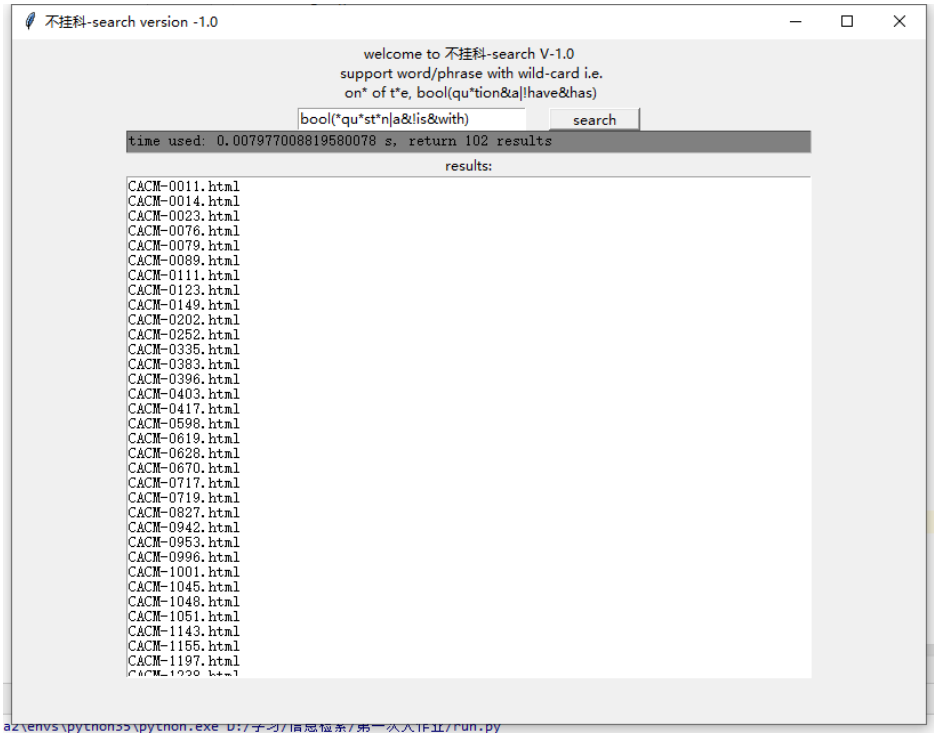
首先，对于两个单词的话，AND 已经在之前提及，OR 即求并集（合并再去重）。NOT 如果是单独使用或是和 OR 连用，取补集即可；如果和 AND 连用，那么 a AND (NOT b)=a-(a AND b)。

对于多个单词，还需要进行人工先转化为析取范式，然后程序识别 &、|、!三个字符串来判断目标，进行运算，先算出&式的结果，再计算|的结果。如 r|(p&!q)就是先计算 p&!q，再计算 r。对于每一个&式，程序自动会把 NOT 表达式放在最后以便和 AND 连用，即自动将 p&!q&r 转化为 p&r&!q（无需人工调整）。因为是析取范式，所以在输入框中不用加括号和空格，识别程序也更加简单。

由于有之前的支持，此处可以支持带有通配符。

h) GUI 界面

利用 tkinter，运行后首先进行数据的初始化，然后可以利用输入框进行带通配符（仅限*）的短语查询（直接输入即可）和 AND、OR、NOT 语句查询（需要以 bool(析取范式)的形式输入）。会显示用时、结果数和文档名，一定程度上模仿了现有的搜索引擎的布局，如下图所示：



4. 实验结果与分析

a) 建立索引的速度、大小、准确率

就速度而言，建立带位置信息倒排索引的平均大约 8.5s，无位置信息也用时几乎同样多，因为其过程是一样的，只不过前者多存储了一些东西。由于算法中获取了两次文本内容，也进行了两次分词，可能导致时间多了很多。但这样避免了内存占用过多。

就大小而言，无位置信息的大小约为 695KB，有位置信息的大小约为 1940KB，都是 csv 存储，前者是序号+词的字符串+文档编号，后者是词的字符串+文档编号+位置序号。

就准确率而言，并不算太高，主要是分词上存有偏误。其中有符号的原因，也有同一词汇的变形视作不同单词的原因，也有一些其他原因。以符号为例，CACM-0001.html 中标题为“Preliminary Report-International Algebraic Language”，导致“report-international”被视为一个词。以变形为例，”problem”和”problems”被视作不同的词。其他原因较为琐碎，比如对数学公式进行拆词等等。

b) skip pointer 的加速作用和新增大小

此处先说明大小。单独记录每个词的字符串和有 skip_points 的文档序号，共花销了 293KB，相当于无位置信息的倒排索引大小的 40%；尽管这并不一定要存储下来。

关于加速作用，和 and query 的对象有关。以运行 5000 次的用时为标准，各结果如下所示：

	“with” and “a”	“have” and “problem	“of” and “question”
无 skip pointer	2.383s	0.702s	2.612s
有 skip pointer	3.031s	0.797s	1.878s
	“is” and “a” and “of”		
无 skip pointer	5.713s		
有 skip pointer	6.432s		

可见，skip pointer 的加速作用并不明显，甚至可能是减速，这与进行 and query 的对象不无关系。由于文档数量较少，每一个倒排索引的 list 并不会很长，而进行 skip 本身便有一定的花销，还不一定能减少多少比较次数。文档数量较少时，skip pointer 可能意义并不大。

c) 短语检索速度与索引大小

关于速度，在没有通配符的情况下，用时并不多——即使是”of the”这种频繁出现的短语（1024 次），也仅仅需要 0.08s。而更长的短语如”one of the”，则只需要 0.03s。这应当还是得益于哈希（字典）的快速访问。

关于大小，在之前已经提及，短语检索用到的 positional_index 带来了约 1MB 的额外内存开销。

d) 通配符检索速度与索引大小

相较于普通单词的查询，尽管通配符单词查询多出一个确定单词的过程，其时间并未多花太多。如搜索”left”用时几乎为 0（直接哈希），搜索”l**t”则平均用时 0.001s，这些时间主要被用于进行 and queries 和表达式检查，且表达式检查时因为词数较少、词长较短，并没有复杂的计算量。尽管有很多情况都是先取全词集再检查，但是考虑到情况较为极端，结果也很可能接近全词集，总体上并不会增加较多用时。

但带通配符的短语查询用时极不确定，可能很多，也可能很少，主要取决于笛卡尔积结果的数量，更根本地则取决于通配符的数量。以“*****”为例，其结果必然极其多（得到文档个数^5 个结果后再求并集）。针对此情况，可以考虑用遍历文本作检查等方式取代之，以约束计算量上限。

通配符检索依赖 bigram_index，其存储后的大小约 1041KB，存储格式为 csv，存储内容是 bigrams 和 words 的字符形式。

5. 代码函数与重要变量说明

源代码包括 ini_and_func.py 和 run.py 两个文件，前者为项目的实现，后者则是针对本项目制作的 GUI，直接在命令行中运行即可，需等待约 10s 以完成初始化，文件同目录内需有名为"cacm"、内含 html 文件的文件夹。因此，以下仅仅是 ini_and_func.py 中的说明

a) 函数

and_query_skip(a,b):即对两个词或列表的使用 skip pointer 的 and query

and_query(a,b):即对两个词或列表的不使用 skip pointer 的 and query

multi_and_query_skip(t):对多个词的使用 skip pointer 的 and query（词为列表格式）

multi_and_query(t):同上，不使用 skip pointer

run_time(func,epoch):测试某语句 func 运行 epoch 次的时间

inverted_untuple_desartes(c):将带括号的笛卡尔积转化为不带括号的列表形式，但是顺序会颠倒。

strict_desc(list):判断一个 list 中的值是否严格递减

judge_adaj(positions):有一系列的词，每个词在文档中有一系列的位置，从而形成了二维列表 positions。用该函数判断是否存在一种位置组合，使得这些词按照顺序出现在文档里（即用于短语查询）。

phrase_search(s):对短语字符串 s 进行查询

generate_bigram(w):将一个词转化为 bigram 列表的形式，包括开头的$?和结尾的?$

search_two_bigram(b1,b2):即相当于两个 bigram（如'ab'和'cd'）或是词列表的查询词的 and query

search_list_bigram(l):即对一个列表的 bigram 进行 and query

check(pattern,w):检查 w 是否符合某模式串

search_star(s):返回通配符单词 s 出现的文档

single_word_wild_card_search(s):返回单词 s 出现的文档，无论是否有通配符

and_query_wild(a,b):类似于之前的 and_query，但是允许对象带有通配符

whole_wild_search(s):进行可带通配符的短语查询，并执行最后的去重，返回文档编号集

num2doc(ans):将文档编号集转化为文档名集合

final_search(s):对大小写进行兼容的可带通配符的短语查询，返回文档名集合

OR_2(a,b):对两个可能含有通配符的单词/列表作 OR 运算

NOT_1(a):对列表/可能含有通配符的单词作 NOT 运算

AND_NOT(a,b):进行 a and(not b)的运算，a 可以是可能含有通配符的单词，也可以是列表，b 只能是可能含有通配符的单词

split_or(s):提取析取范式中的每一个&表达式

dealwithand(s):计算一个&表达式(如 a&!b&c，即 a AND(NOT b) AND c)的检索结果

multi_OR(l):对析取范式中每一个&表达式的结果进行合并

bool_calculate(s):对析取范式字符串进行检索

b) 变量

path:即 cacm 文件夹的名字

punctuations:即需从分词结果中去掉的英文符号

word_doc_dict:即无位置信息的倒排索引

positional_index:即有位置信息的倒排索引，形式为{词:{文档：[位置]}

bigram_word:即 bigram 的倒排索引