Web lab1 实验报告

> *PB18000037 肖桐 PB18071521 高路尧*

实验说明

我们进行实验对数据进行了预处理操作方便存储以及数据的读写.

具体在于将5个文件夹的内容进行**综合到了5个json文件中**, 并命名为`2018_01.json 2018_02.json 2018_03.json 2018_04.json 2018_05.json`

对每个文章选择顺序的方式赋予新的ID, 方便进行数据的存储和选择, 记录在`output/id2uuid.pkl`，可用 `Python pickle` 包进行读取

在对tfidf向量的矩阵存储方面, 我们选择使用**稀疏矩阵的方法**来降低对空间的占用. 因为对数据的分析后发现提取的文章中的词汇在的其他文章中出现的概率很小, 即对应的矩阵值为0. 矩阵的行对应的是提取的`word`单词, 列对应的是文章的id, 记录的内容是的文章中这个单词的`tfidf`值.

数据处理 build_indices.py

我们保存的倒排表是文章和词语之间的稀疏矩阵，同时该倒排表保存在一个文件 `output/invert_indices.dict` 中，可用 `Python pickle` 包进行读取。文本取文章的标题和文章的主体。

在数据处理部分使用`nltk`, `gensim`等python库, 来进行文章语言的处理.

`word_list = list(gensim.utils.tokenize(text, lowercase=True, deacc=True))` `# tokenize` 来进行分词操作

当单词在停用词表时将其删除

使用 `porter_stemmer = nltk.stem.PorterStemmer()`, `porter_stemmer.stem(word)` 对单词进行标准化处理.

对于每一篇文章，在经过上述的标准化等操作之后，将文章的 `id` 附加到所有在该文章内出现过的单词的倒排表中，同时记录文章的长度和每个词出现的频数，用来计算 `tf` 值。要计算每个单词的 `idf` 值，就只需要用对应单词的倒排表的长度与文本总数相除即可。这样就得到了稀疏的 `tf-idf` 矩阵。

Bool检索 bool_search.py

输入词格式限制：`AND`、`OR` 视为二元运算符，`NOT` 视为一元运算符，表达式中可以带括号，所有不属于 `AND`、`OR`、`NOT`、括号的字符串都将视为搜索词。运算符的优先级为：括号 > `NOT` > `AND` = `OR`。我们实现的 `bool_search` 对表达式的大小写不敏感（但是对搜索词大小写敏感）。

实现方式：受编译原理课程启发，可以对输入的表达式分别作词法分析、语法分析，然后构建出一棵语法树。词法分析器负责返回 `Token`，`Token` 种类有 `AND`、`OR`、`NOT`、`LB`（左括号）、`RB` 右括号、`WORD`（搜索词）。语法分析负责根据预测分析器返回的结果构建语法树，具体步骤如下：

1. 首先去除多余的最外围括号，比如 `("xt" AND "gly")`（对应词法分析器返回的结果为 `LB WORD AND WORD RB`）中就可以将最外围括号去除。
2. 找出最顶层的运算符，比如 `("1" OR "2") AND "3"`，对应的最顶层运算符为 `AND`。若不存在最顶层运算符，则该表达式视为搜索词；否则以该最顶层运算符为分割点，左右分别进行递归构建语法树（如果最顶层运算符为 `NOT`，则只需要对右边的表达式进行递归构建）.

得到对输入`Bool`数据的语法树后, 进行Bool检索操作. 对语法树进行`def tree_to_stack(root: Tree)`将树通过后序遍历转化成栈存储起来.

在`def bool_search(indicesfile:str)`中对栈中的数据进行分析. 使用python中的`set`的数据结构对结果进行处理, `And`操作是对两个set集合进行`&`操作, `OR`操作对两个set集合进行`|`操作, NOT操作是对目标set和全集进行`fullset.difference(setstack[-1])`操作. 时间复杂度在实验数据中为`O(n+m)`, `n,m`为两个`set`的元素个数

最后的结果得到唯一的set集合, 即为得到的文章id的集合. 在最后使用`id2uuid.dict`字典, 在最后将实验的文档id转成uuid输出.

测试样例如下:

Tf-idf Semantic_Search.py

数据记录格式稀疏矩阵见上

在得到输入的查询词后, 经过标准化操作在矩阵中找到对应单词和文章的tfidf向量`d`, 在对查询词建立`tfidf向量q`后, 进行cosine的相似度计算. 计算公式如下:

$$cos(\vec{q},\vec{d}) = \frac{\sqrt{\sum_V^{i=1} q_i d_i}}{\sqrt{\sum_V^{i=1} d_i^2}\sqrt{\sum_V^{i=1} q_i^2}}$$

因此, 在计算过程中, 对于在稀疏矩阵上使用的优化方式是对查询词所对应的tfidf进行累加后得到结果. 引入了`TextInfo`数据类型, 对每个文章`text`都有对应的记录. 第一项是`self.dot`是累加查询词tfidf和文章对应词的tfidf的结果. 因为在文章中没有出现的查询词的tfidf都记录为0, 最后得到的结果是 $\sum_V^{i=1} d_i^2$.

同理, 第二项记录的是文章对应词的tfidf的乘积和, 通过遍历即可得到最后的结果是 $\sum_V^{i=1} q_i d_i$. 最后根据公式$cos(\vec{q},\vec{d}) = \frac{\sqrt{\sum_V^{i=1} q_i d_i}}{\sqrt{\sum_V^{i=1} d_i^2}\sqrt{\sum_V^{i=1} q_i^2}}$ 可以得到每个文章tfidf向量相对于查询词tfidf向量的`cos`值, 选择最大的`cos`值对应的文章id和uuid即可得到结果.

测试样例如下:

对实验给出的`searchwords.txt`的内容进行检测后得到的结果是 . 并且根据文本分析, 文章中出现的查询词数量有70个, 并且出现次数为300次, 足以说明文章内容和查询词极为相关. 如图所示:

实验进行的优化

1. 我们使用 `Python multiprocessing` 使用多进程对多个文件进行处理，因为预先将5个文件夹中数据都综合到了5个 `json` 文件中，因此多进程就很好操作了。实现方式为，将在数据处理阶段计算 `tf-idf` 矩阵的阶段写在单个函数内，返回值为一个倒排索引表。我们要做的是，对于5个文件用多进程的方式调用都一次这个函数，然后在对最后返回的结果做一个综合就可以了。实测单进程建立倒排索引表需要 20-30min，多进程加速之后只需要 5-6min。
   结果图:

2. 倒排索引的空间复杂度的优化, 改记录 `uuid` 为 `id` 。考虑到我们的实验数据是静态不变、相对独立的，所以完全没必要使用 `uuid` 这么长的字符串作为区分。可以使用一个简单易分别的整数来代替 `uuid` 。因此我们自己为所有的文本都进行了一个编号，编号范围为 `0 ~ 306241` 。

   这样做的好处有两个：

   - 在建立倒排索引表的时候可以节省很大的内存和硬盘空间
   - 在进行语义查询时，可以为计算 `tdidf` 向量的 `cosine` 相似度带来便捷。

   对于第一个好处而言，使用 `Python sys.getsizeof()` 方法可以看到，每个 `id` 所占空间大小为 `28 Bytes`，而每个 `uuid` 所占空间大小为 `89 Bytes`。粗略计算，我们统计出来的单词个数为 `265922` 个，若倒排表稀疏因子为 `0.02`，则节省的内存空间为 $265922 \times 306242 \times 0.02 \times (89 - 28) = 92.53GB$