09017232刘晓臻\_课程研学报告（一）

**学号：09017232 姓名：刘晓臻 日期：2020.04.18**

# 一、建立文档库

按照要求，至少要有30篇以上的文档。在本人的《关于雅虎搜索(Yahoo! Search)搜索引擎的学习报告》中第三部分，我所选择的论文为《Effective Online Knowledge Graph Fusion》，其参考文献只有20篇，故额外在其参考文献中最新的文章《Biperpedia: An ontology for search applications》的参考文献列表中选择了前15篇文献添加至文档库。

文档库参见文件<docs.txt>，每行为一个文档。

# 二、建立索引

本报告的所有结果都由<simple_retrieval.py>完成。对文档作预处理（包括排除停用词与字母统一为小写）后，使用哈希表存储term，得到倒排索引，其中每个posting中还包含了该term在文档中出现的位置（第几个token）。在倒排索引实现时，为了能够构建向量查询中的term-document矩阵，对每个term编号（顺序为在字典中的存储顺序）。为了显示结果，倒排索引打印到了<inverted_index.txt>中。如图1，输出格式为：

term： term编号 {文档号: [该文档中第1次出现位置，第2次出现位置...], ...}

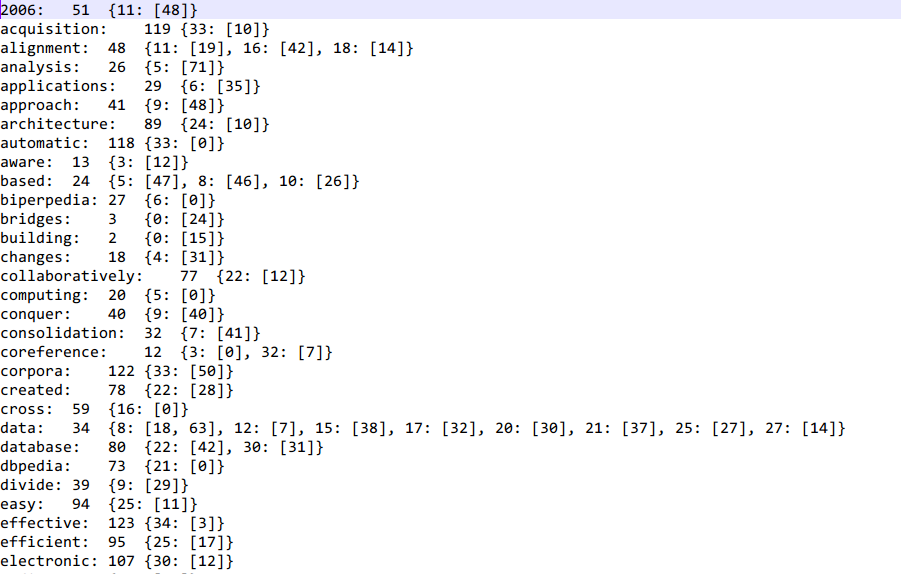


图 1

由于使用哈希表时借用了Python中内置的字典数据结构（其内部实现就是使用的哈希表，且是开散列），其实现查询term的算法的伪代码如下：

*# string 是待查找的term, dicti为待查找词典(hash表)*

*def* lookup(*string*, *dicti*):

*# hash(...)为选用的的哈希函数，将字符串映射到唯一的一个数。*

    h = hash(string)

*# dicti.slots[]为该hash表的hash槽，若其非空则在其中存放了*

*# 实际存放的term的hash值，及对应的postings的指针(value)*

    if (dicti.slots[h].is\_empty()):

        throw(key\_not\_found\_error)

*# 若匹配上，则找到，返回对应的值。*

    elif (dicti.slots[h].actual\_hash\_value == h):

        return dicti.slots[h].value

*# 否则，顺序往后查找，找到底则回到最初，直到再次回到该hash值，说明没有找到，报错*

    else:

        current\_slot\_number = h + 1

        while(True):

            if (dicti.slots[h].is\_empty()):

                throw(key\_not\_found\_error)

            elif (dicts.slots[current\_slot\_number].actual\_hash\_value == h):

                return dicti.slots[current\_slot\_number].value

            else:

                if (current\_slot\_number + 1 >= dicti.capacity):

                    current\_slot\_number = 0

                elif (current\_slot\_number + 1 == h):

                    throw(key\_not\_found\_error)

                else:

                    current\_slot\_number += 1

# 三、建立文档向量

文档是按照[doc.txt](docs.txt)中的顺序编号的，term编号见<inverted_index.txt>。得到的文档向量输出在了<term_document_matrix.txt>，如图2。每一行为一个文档向量，每个term的权重为其在文档中的出现次数。

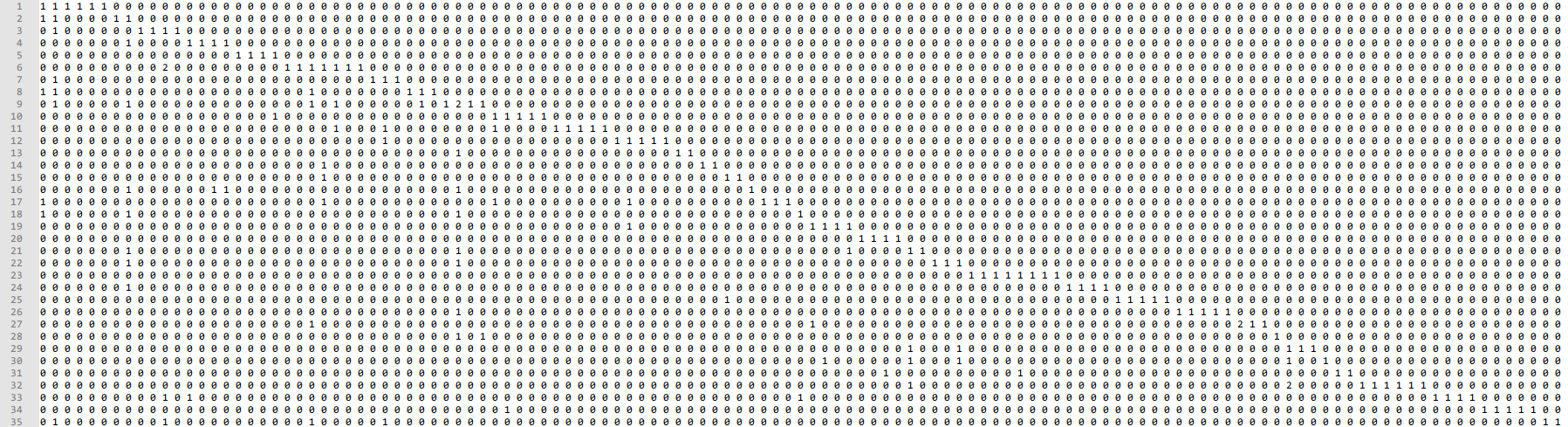


图 2

# 四、建立查询和查询向量

查询为“Effective Online Knowledge Graph Fusion”，查询向量输出在<query_vector.txt>中。同样，每个term的权重为其在文档中出现的次数。需要注意的是，query中存在term词典中没有的词，会被自动过滤掉。

# 五、利用向量模型，计算查询q 与所有文档的相关度

为了加速计算，不直接使用向量相乘，且仅对每个query中的term逐一计算各项分数（因为不在query中的term不可能影响到排序，故可以省略对其的计算。即，使用类似于图3中的算法（除了相乘的不是wf之外），通过计算图4中的各项值，得到每个文档的相关度。

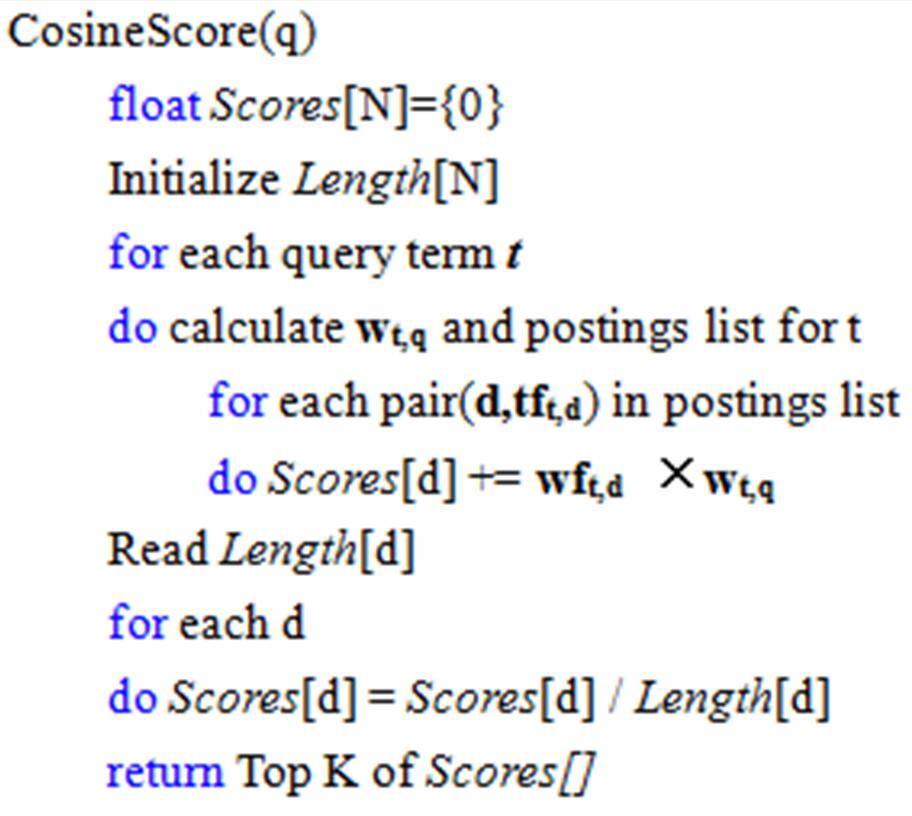


图 3



图 4

对于query的term权重的计算，采用教材（英文版）第六章P128. 的增广权重：

对于document的term权重计算，使用PPT中的计算方法：

另外，对于query，采用wf-idf作为最终的*qi*，对于文档，作余弦归一化（对wf向量）。

query一栏的计算和document一栏的计算过程分别输出为<query_calculation.txt>和<docs_calculation.txt>，文档中每行的数字顺序分别和图4中的顺序对应。

最后得到所有文档和query的相似度得分，排名后输出到<vector_space_rank_and_scores.txt>中，如下图5：

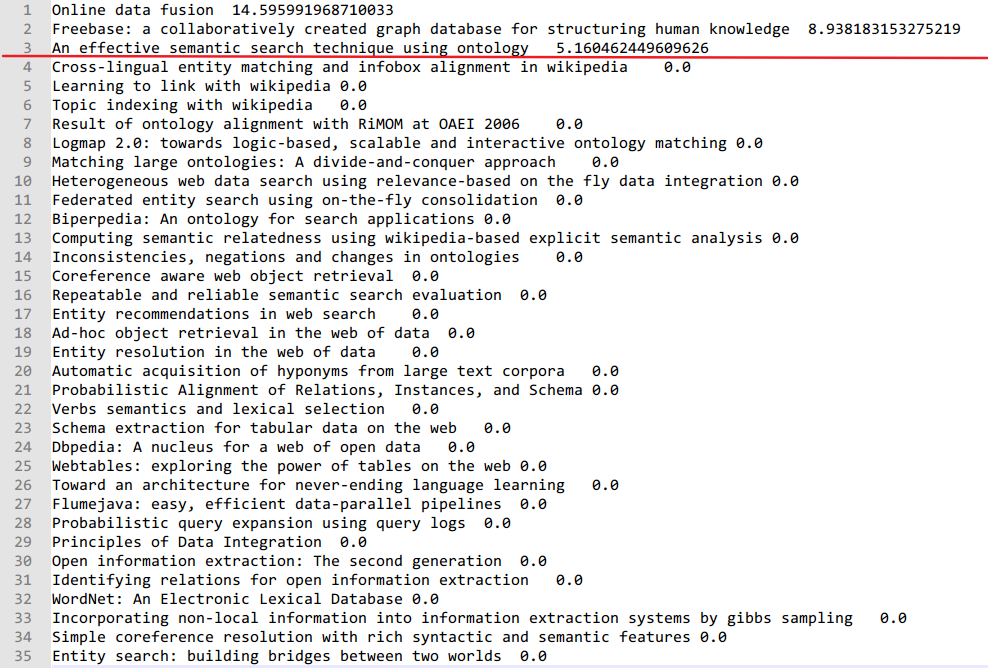


图 5

# 六、基于概率模型的语言模型计算查询q 与所有文档的相关度

1. 计算并建立所有文档向量和查询向量的语言模型。

对于文档向量、查询向量（以及第二问中文档集）的先验概率的计算，使用的是第11 章11.3.3节中的第3种计算方法：

其中*c*, *d*, *q*分别代表某个文档集、某个文档和某个查询，*N*为文档集中的文档数量，*L*为文档中token的数量（去掉停用词之后）。

通过以上方法得到所有文档的语言模型输出到[docs\_lan\_models.txt](doc_lan_models.txt)，查询的语言模型为<query_lan_model.txt>，文档集的语言模型为<col_lan_model.txt>，每个文件中每一行都代表dictionary里每个term的概率（按照term编号的顺序）。

2. 计算查询q 与所有文档的相关度，按相关度从高到低排序后输出。

根据以下计算公式，取=1/2，对每个文档计算概率（得分），排序后输出到[lm\_ rank\_and\_scores.txt](lm_rank_and_scores.txt)，格式同图5。计算过程输出为<lm_calculation.txt>（如图6），其格式为：每个文档对于每个query中的term的计算占一行，这一行的格式为：*P*(*t*|*M*c) *P*(*t*|*M*d) 1/2(*P*(*t*|*M*c) + *P*(*t*|*M*d))。



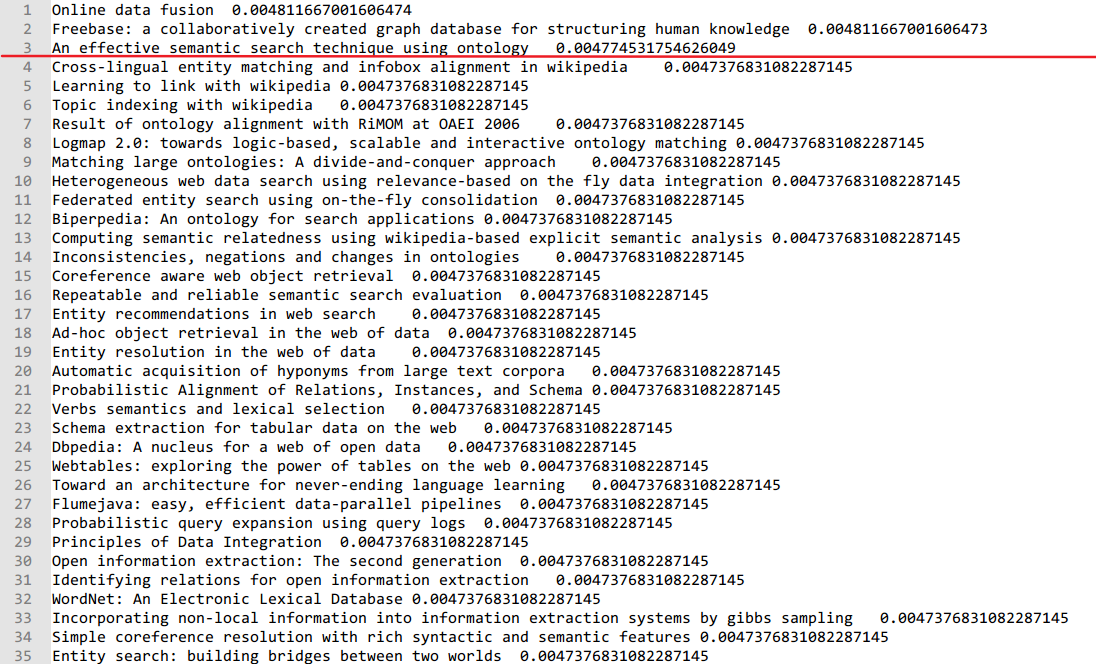


图 6

# 七、对比和分析两种相关度计算的结果，谈谈个人对信息检索相关度计算方面的看法

对比两种相关度计算方法，可以发现计算的结果排名完全一致。这并不是说明它们是一样的效果，而是因为与查询的terms有交集的文档仅有3篇，故其他的文档完全无关，因此只有3篇文档真正被排序了！所以无法比较效果。这个现象的主要原因是查询太短，标题太过简洁。故本人额外使用文档库中的文档《Computing semantic relatedness using wikipedia-based explicit semantic analysis》构建文档库，来对两种方法进行比较。各种输出文件同上，但放在了<additional_doc>文件夹中。

向量模型的结果如图7，语言模型的结果如图8：

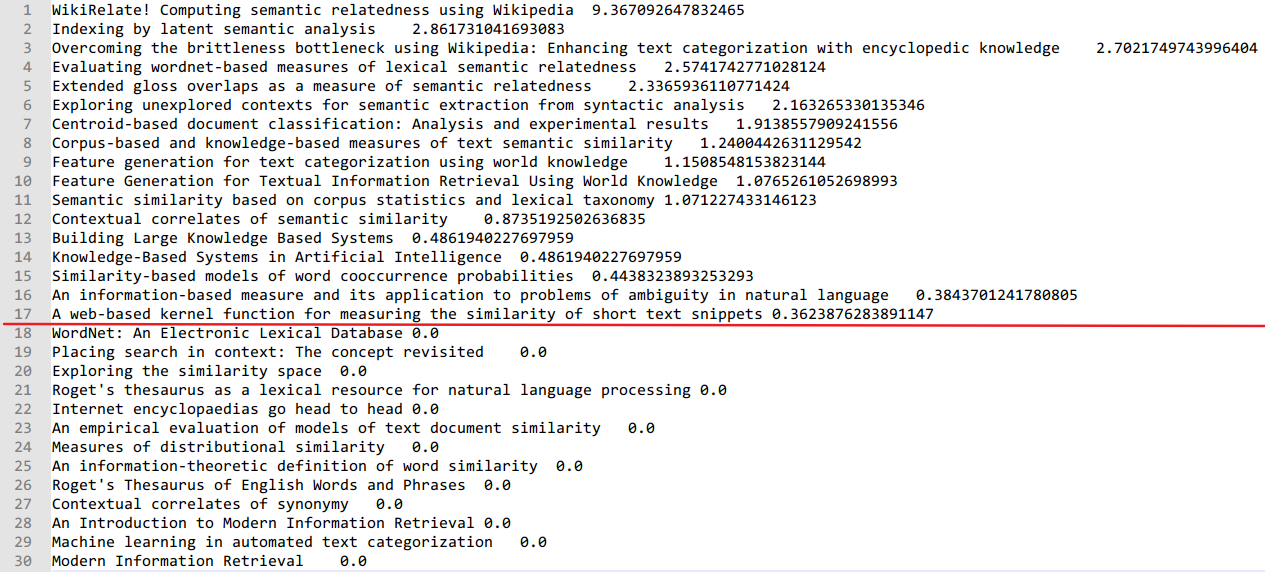


图 7

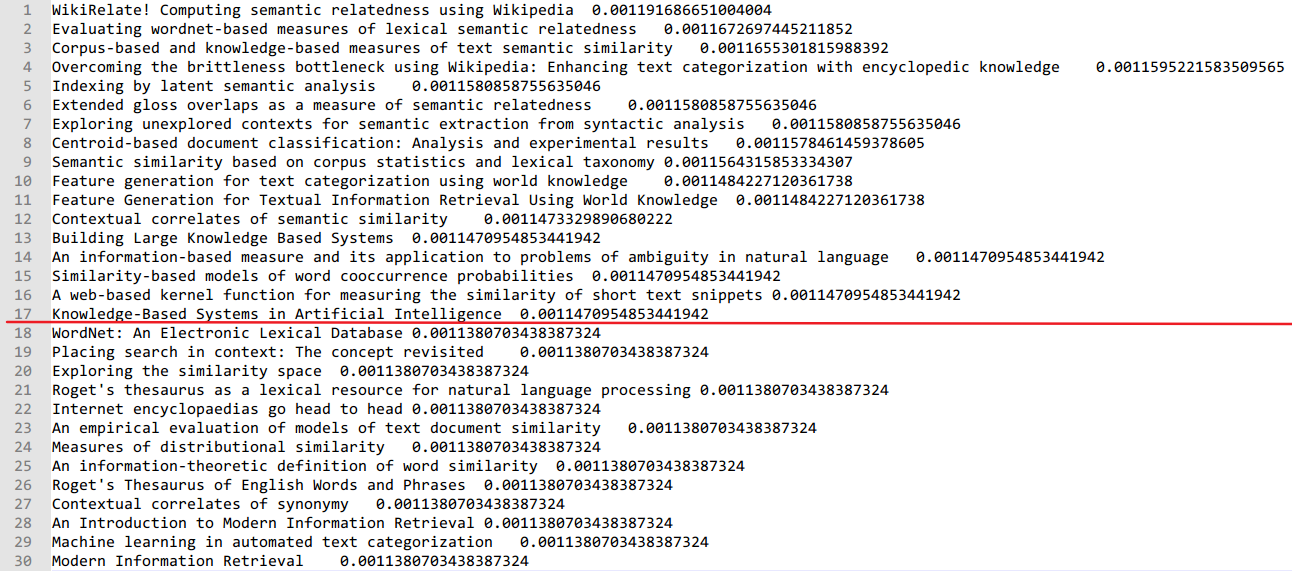


图 8

前17篇文档都是不完全无关的（有与query的terms重叠的词）。这次的结果并不完全相同。这种差异可能是与两种模型对于文档长度在排序中的使用以及文档集中term频率的不同使用方法有关：向量模型使用的是idf，而语言模型中使用的则是cf。而由于语言模型中使用的是unigram模型，故语序并没有起作用，因此，本实验中的语言模型与向量模型所使用的其他信息比较相似，两种模型的结果也差不多：query中出现的词在document中出现得越多，这个document就越靠前，差异仅体现几个document之间具有相同的query词频的时候。

而实际应用中不能忽视语序，故对于语言模型，最好使用bigram及以上的模型，否则效果与向量模型差不多。