第三章作业实验报告

项目准备

本信息检索作业采用基于Python3的Jupyter Notebook进行编程，需要第三方库`pandas`，以及Python内置库`math`，`tkinter`，`functools`。原始工程代码为`.ipynb`格式，另有导出的`.py`源代码及`.html`格式的代码及运行结果。

文档预处理

详细内容请查看preprocessing.ipynb(/.py/.html)。

词汇表

为了建立词汇表，首先要对文档文本按词进行划分。该工作采用以下代码完成：

```
doc = get_doc()
doc_words_1 = []
for d in doc[1]:
    w = d.lower().replace("."," ").replace(","," ").replace(", "," ").split(" ")
    w = [word for word in w if word!=""]
    doc_words_1.append(w)
```

考虑到不同的检索模型对词汇表有着不同的要求，这里在逐个单词地对文本进行拆分的基础上，还按照短语对文本进行了划分。代码如下：

```
doc_terms = []
for i in range(len(doc_words_1)):
    d_t = doc_words_1[i] + doc_words_2[i]
    doc_terms.append(d_t)
```

鉴于预设的检索内容最多只有两个词，这里只建立包含两个词的短语。如果需要检索更长的短语，额外拼接即可。

基于文档的划分，可以建立词汇表：

```
def get_vocab(doc_terms):
    vocab = []
    for ts in doc_terms:
        for t in ts:
            if t in vocab:
                pass
            else:
                vocab.append(t)
    return vocab
```

该函数可以处理单个单词组成的词汇表，也可以处理包含任意长度短语的词汇表。

为了通过词汇表表示文档，还需要建立每个文档的“项共现模式”，即c(dj)，通过对文档的拆分表示和词汇表求交集即可得出。

```
def get_c_dj(doc_terms, vocab):
    c_dj = [] # pat of term co-occur / term conjunctive component
    for terms in doc_terms:
        c = []
        for i in range(len(vocab)):
            if vocab[i] in terms:
                c.append(1)
            else:
                c.append(0)
        c_dj.append(c)
    return c_dj
```

在布尔检索模型中，需要用到与c(dj)相似的c(q)来表示检索内容在词汇表中的出现情况。经典的布尔检索可以包含比较复杂的且、或、非逻辑判断，用q_DNF来表示所有符合要求的查询结果（c(q)）的并集，但考虑到预设的查询不含这种逻辑组合，可以进行简化，对所有的c(q)取交集，得到一个最小项出现模式，比如词汇表为[a,b,c]，查询内容为[b]，那么传统上用[0,1,0] or [0,1,0] or [1,1,0] or [1,1,1]来表示，这里可以简化为[0,1,0]，即必须出现的词用1表示，不要求的词用0表示。

该简化的实现代码为：

```
def get_c_q_min(vocab,query):
    c_q_min =[]
    #if c_q = (010)or(011)or(110)or(111), then c_q_min = (010),
    #indicating the must-have terms with 1 and others with 0
    for v in vocab:
        if v == query.lower().rstrip():
            c_q_min.append(1)
        else:
            c_q_min.append(0)
    return c_q_min
```

项权重

这里采用项在文档中的出现频率来表示权重：

```
w.append(doc_terms[i].count(vocab[j]) / get_doc_length()[i])
```

查询内容的权重与之类似：

```
weight.append(qry_terms.count(vocab[j]) / len(qry_terms))
```

布尔检索模型

布尔模型包括两个环节：计算相似度和返回检索结果。

详细内容请查看boolean_model.ipynb(/.py/.html)。

词汇表的运用

布尔检索的特性为：检索结果是一个布尔值，即只有“找到”和“未找到”两个选项。对于短语的检索，这里认为短语必须整体出现才视为“找到”；如果部分出现，或者分别出现，均视为“未找到”。基于以上两点，布尔模型使用包含了所有单词、短语的词汇表，并且对单词、短语不加区分。检索时，只需将检索内容整体代入词汇表进行查找。

相似度的计算

对于不在词汇表中的检索内容，与任何文档的相似度均为0。

对于存在于词汇表中的，检索内容对应的词汇如果出现在文档中则相似度为1，否则为0。

相似度计算的代码实现如下。

```
def bool_sim(qry,doc):
    #requires two [1000100...]s
    if qry.count(1) == 0:
        return 0
    sim = 1
    for i in range(len(qry)):
        if (qry[i] == 1) and (doc[i] == 0):
            sim = 0
            break
    return sim
```

与检索内容相似度为1的文档，作为检索结果返回。

布尔检索样例

```
# In[2]:
print(bool_query("to"))
# Out[2]:
# ['d1', 'd2', 'd5', 'd6']
# In[5]:
print(bool_query("I am"))
# Out[5]:
# ['d2', 'd3']
```

向量检索模型

相比于布尔检索，向量检索支持部分匹配和检索结果排序。对于短语的检索，这里认为只要有一部分出现在文档中即视为检索成功，但出现得越多，相似度越高。

详细内容请查看vector_model.ipynb(/.py/.html)。

词汇表的运用

与布尔模型不同，这里只针对单词建立词汇表，不包含短语。这种做法出于以原因：第一，对于短语，向量模型不需要整体匹配，而是使用部分匹配并计算非布尔的相似度；第二，计算相似度时需要用到词汇的权重，而只用单词建立词汇表，就只对文档按照单词划分，这种划分能反映文档的长度，便于通过出现频率计算权重，而如果使用单词和词汇划分，在尝试中发现频率的计算会有问题，最终导致相似度排名不合理。

相似度的计算

首先需要在预处理中得到的文档权重矩阵d_w，并计算查询内容的权重矩阵q_w，求得两个矩阵的夹角cos<d_w, q_w>，其结果即为相似度。在代码实现上，这里采用了map-reduce来简便计算：

```
def vect_sim(qry,doc):
    #requires two weights [0, 1, 0] and [0.4, 0.4, 0.2]
    upper = reduce(lambda x, y: x+y, list(map(lambda x, y: x*y, qry, doc)))
    l1 = reduce(lambda x, y: x+y, list(map(lambda x, y: x*y, doc, doc)))
    l2 = reduce(lambda x, y: x+y, list(map(lambda x, y: x*y, qry, qry)))
    lower = math.sqrt(l1)*math.sqrt(l2)
    if lower==0:
        return 0
    else:
        sim = upper/lower
    return sim
```

相似度排名

相似度越高的文档，说明查询内容在文档中出现越多、越全。根据相似度的值，对文档进行排名并返回：

```
def vect_query(qry):
    result = []
    doc = get_doc()
    for i in range(doc.shape[0]):
        q = get_qry_weight(qry,get_vocab(get_doc_terms_sing()))
        d = get_doc_weight(get_doc_terms_sing(),get_vocab(get_doc_terms_sing()))
[i]
        sim = vect_sim(q, d)
        result.append(["d%d"%(i+1), sim])
    result.sort(key=take_second,reverse=True)
    return result
```

向量检索样例

```
# In[8]:
print(vect_query("do"))
# Out[8]:
# [['d3', 0.6708203932499368],
#  ['d4', 0.5477225575051661],
#  ['d5', 0.408248290463863],
#  ['d1', 0.3779644730092272],
#  ['d6', 0.3162277660168379],
#  ['d2', 0.0],
#  ['d7', 0.0]]
# In[11]:
print(vect_query("Let it"))
# Out[11]:
# [['d5', 0.5773502691896256],
#  ['d4', 0.5163977794943222],
#  ['d6', 0.22360679774997894],
#  ['d1', 0.0],
#  ['d2', 0.0],
#  ['d3', 0.0],
#  ['d7', 0.0]]
```

倒排索引检索模型

详细内容请查看inverted_indexes.ipynb(/.py/.html)。

索引建立

这里的倒排索引是基于只包含单个单词的词汇表建立的。对于词汇表中的每一个词，在按单词分割好的文档中统计出现频数和位置，以此建立索引，程序如下：

```
def get_indexes(vocab,doc_terms):
    indexes = []
    for v in vocab:
        index = []
        all_docs_count = 0 #appearence count in all docs
        occurence = []
        for i in range(len(doc_terms)): #for each doc
            occur = [] #[doc, count, [pos1, pos2...]]
            doc_count = doc_terms[i].count(v) #appearence count in each doc
            if doc_count != 0: #if appears
                all_docs_count = all_docs_count + 1
                occur.append(i+1)
                occur.append(doc_count)
                pos = []
                for j in range(len(doc_terms[i])):
                    if doc_terms[i][j] == v:
                        pos.append(j+1)
                if len(pos)!=0:
                    occur.append(pos)
            if len(occur)!=0:
                occurence.append(occur)
        index.append(all_docs_count)
        index.append(occurence)
        indexes.append(index)
    return indexes
```

建立好的索引每行代表一个单词，并包括出现的文档数、各文档的序号、出现频数和位置：

```
[[4, [[1, 4, [1, 4, 6, 9]], [2, 2, [1, 5]], [5, 1, [4]], [6, 1, [3]]]],
 [5,
  [[1, 2, [2, 10]],
   [3, 3, [6, 8, 10]],
   [4, 3, [1, 2, 3]],
   [5, 1, [5]],
   [6, 1, [9]]]],
 [1, [[1, 2, [3, 8]]]],
 ...
 [1, [[7, 1, [3]]]],
 [1, [[7, 1, [4]]]]]
```

基于索引检索

对于单词的检索，直接匹配该单词出现的位置，并在返回时将索引中的文档编号、出现位置转换成类似坐标的格式

```
pos = index[1] #[doc, count, [pos1, pos2...]] * n
    result = []
    for p in pos:
        a = p[0] #in which doc
        for b in p[2]: #pos in a doc
            result.append([a,b])
```

如果检索内容是一个短语，则对两个词分别检索，得到各自出现的坐标后判断是否相邻，若都检索成功且相邻则返回该检索结果：

```
positions =[]
for i in range(len(q)): #each word in a querey
    pos = index_query_sing(q[i], get_vocab(get_doc_terms_sing()),
get_indexes(get_vocab(get_doc_terms_sing()), get_doc_terms_sing()))
    positions.append(pos)
candidates = positions[0]
result = []
for c in candidates:
    match = True
    p1 = c[0]
    p2 = c[1]
    for i in range(1, len(positions)):
        if [p1, p2+i] not in positions[i]:
            match = False
    if match:
        result.append(c)
```

索引检索样例

```
# In[14]:
print(index_query("to do"))
# Out[14]:
# [[1, 1], [1, 9], [5, 4]]
```

可视化检索系统

系统基于Python的tkinter库搭建，接受输入的查询数据，调用封装好的检索模型，并显示查询结果。

详细内容请查看gui.ipynb(/.py/.html)。

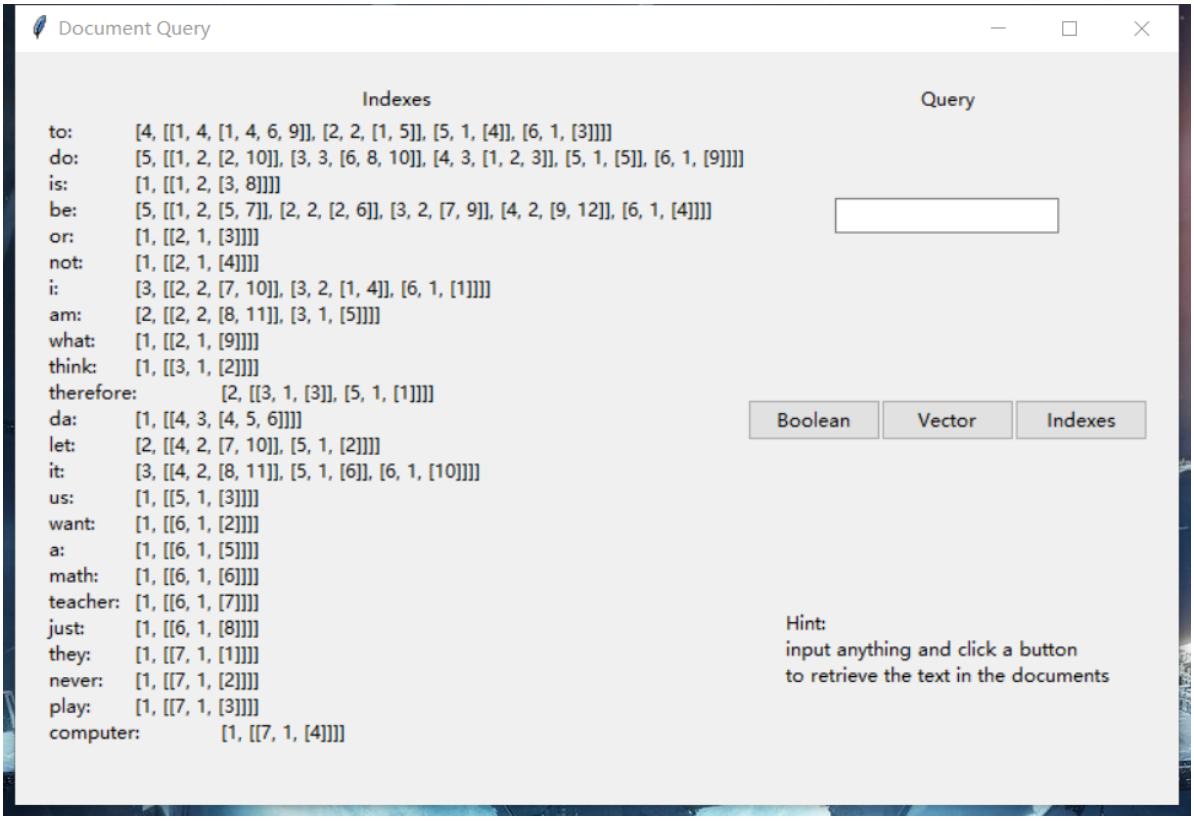
另有非可视化的查询样例，请查看do_query.ipynb(/.py/.html)。

直接运行`gui.py`进入可视化系统，在首页左侧展示了倒排索引，内容是通过调取索引函数获得的。

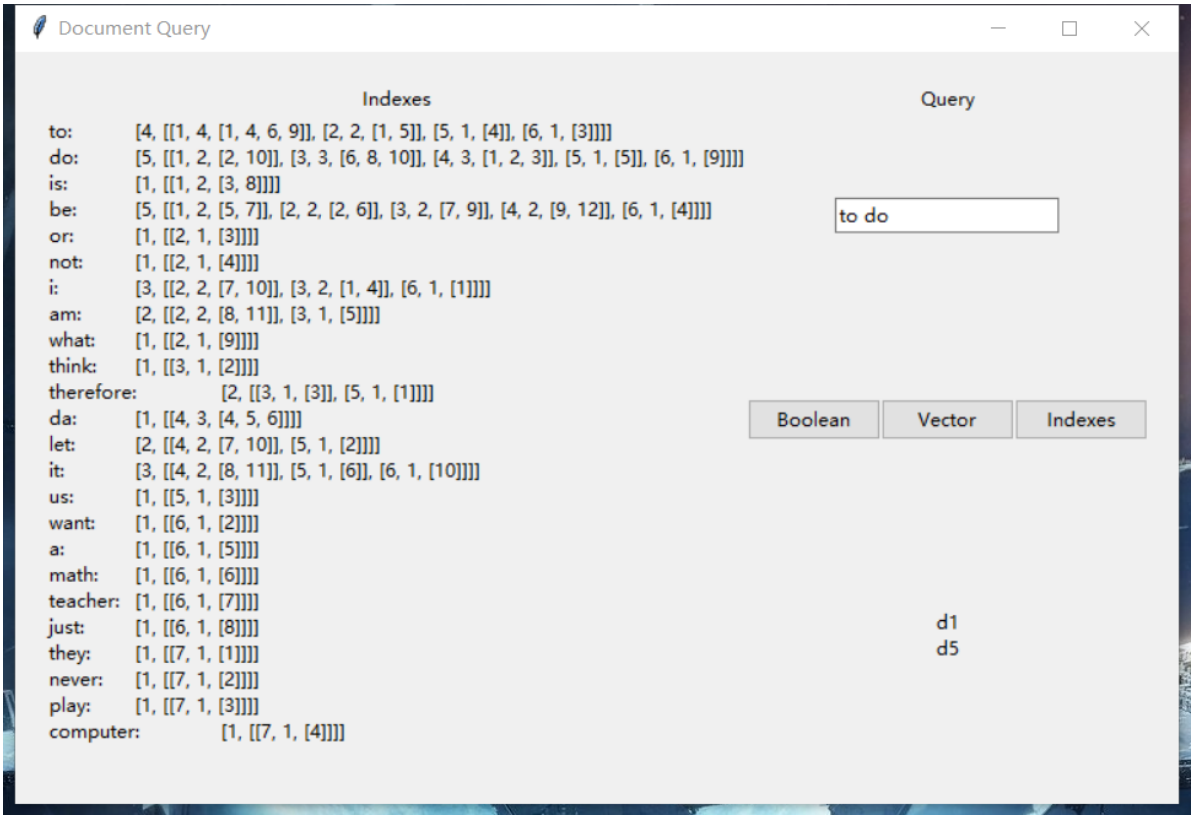
```
vocab = get_vocab(get_doc_terms_sing())
indexes = get_indexes(get_vocab(get_doc_terms_sing()), get_doc_terms_sing())
vocab_indexes = []
for i in range(len(vocab)):
    vocab_indexes.append(vocab[i] + ": \t" + str(indexes[i]))
vocab_indexes
v_i = ""
for index in vocab_indexes:
    v_i = v_i + str(index) + "\n"
```

以单词”am“为例，其索引内容为“[2, [[2, 2, [8, 11]], [3, 1, [5]]]]”，表示一共在两个文档中出现，其中在d2文档中出现两次，分别在第8和第11个单词的位置；在d3文档出现一次，在第5个单词的位置。

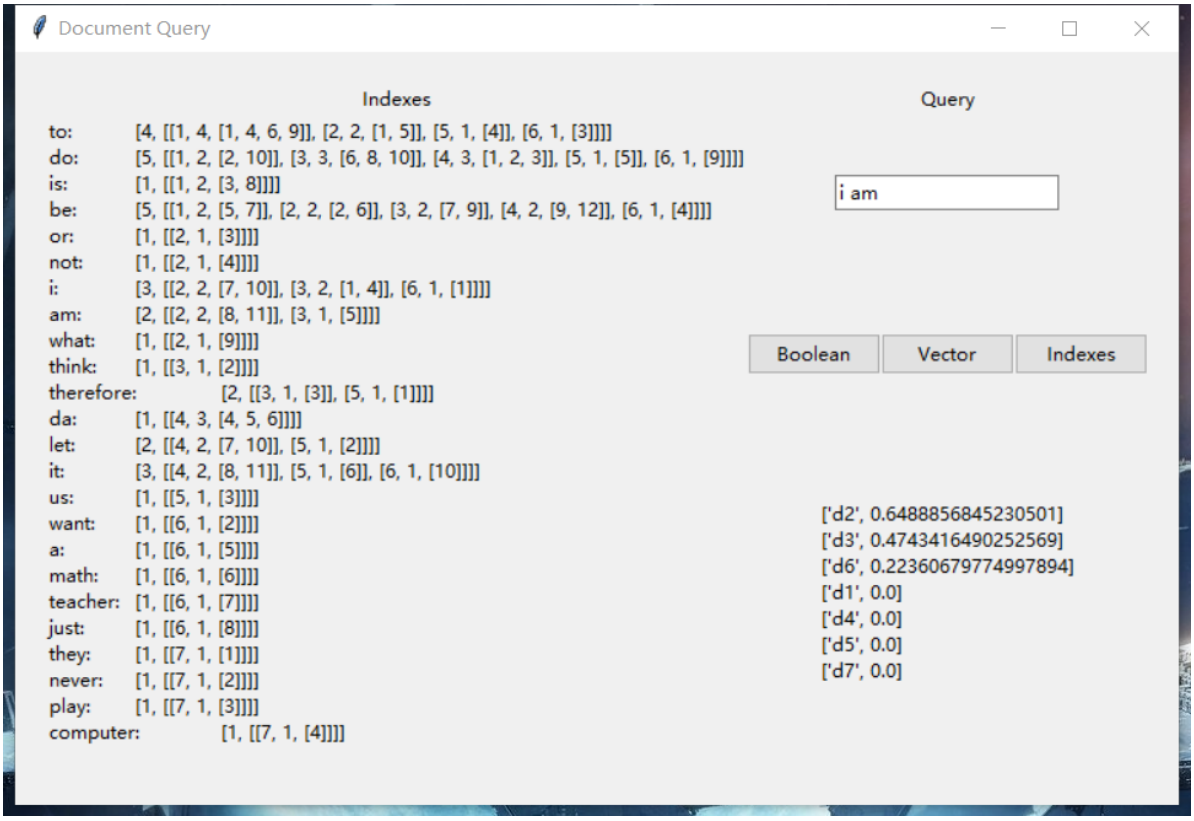
在d首页右侧是检索区，输入任意内容后，点击按钮即可按照三种不同模型检索文档。



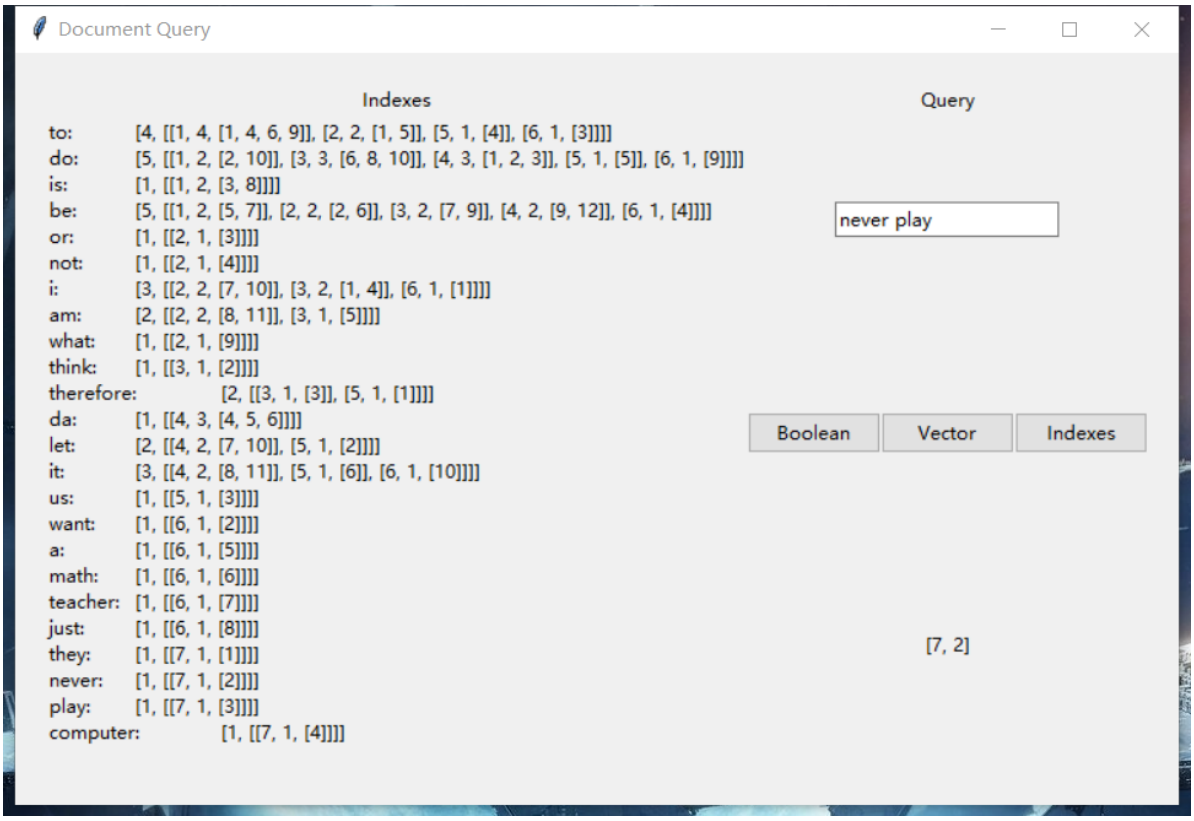
布尔模型检索效果如下，可见在d1和d5两个文档中有"to do"内容。



向量模型检索效果如下，可以看出“i am”内容与d2文档相似度最高，其次是d3、d6，而在其他文档中没有出现任何包含“i”或者“am”的内容。



倒排索引检索效果如下表明“never play”在d7文档的第二个单词位置出现了一次。



对比分析

布尔检索模型的优势在于，可以对多种查询条件进行逻辑组合，尤其是对于“或”和“非”的严格限制是向量检索所不具备的，因此，在查询条件比较复杂、多样时布尔模型具有一定的优势。此外，在代码的实现上，从本项目的开发来看，布尔模型在处理短语检索时有一定的困难，既然模型是基于词汇表进行检索，必须在查询前就建立词汇表，但在短语的最大长度未知的情况下，词汇本对短语的处理就有很大的不确定性，可能会带来大量的冗余和性能浪费。（注：本项目的布尔模型基于检索内容必须全部匹配这一假设。）

向量检索模型的优势则在于其可以现实部分匹配，并针对匹配情况计算一个浮点数的而非布尔的相似度。这样做的一个优点就是相比于布尔模型可以更全面、充分地反映查询与文本的匹配情况，不会出现99%都匹配但是因为一个标点的不同而导致无法识别的情况。此外，利用相似度可以对文档进行排序，这在文档数量很大时非常有用，比如搜索引擎中可能有成千上万条匹配结果，利用向量模型可以选择其中最相似的几十条结果，而如果用布尔模型，就只能把大量的结果全部返回。在实际体验上看，向量模型的性能稍差，大约需要五秒完成一次检索，而另外两个模型均在一秒以内。

倒排索引与前两种模型的一大区别就在于不是基于文档来组织索引，而是预先求出每一个词的出现情况。这样直接用查询的词汇进行匹配，可以得到很高的效率。实际体验中，倒排索引的执行也是很快的。此外，想比如布尔模型和向量模型，倒排索引检索用到的索引有更强的可读性，人可以亲自查看索引，手动检索内容。尽管在基于计算机的信息检索系统中这用处不大，但却有着现实意义，比如人通过纸笔记录的一些事项，如果能够建立这种索引，事后查找就会很方便。倒排索引还有一个非常重要的优势，就是可以通过索引直接定位检索内容，就像前一部分的样例中，直接可以了解到待检索的“never play”是出现在d7的第二个词的位置，不像前两种检索只说在哪些文档出现。