

Web信息处理与应用实验报告

lab1-邮件搜索引擎实践

曾勇程 PB18000268

张弛 PB18000269

Web信息处理与应用实验报告

lab1-邮件搜索引擎实践

实验目的

实验内容(算法)

实验条件

使用算法及优化方式

预处理部分

获得所有文档路径

提取邮件有用内容

分词

去除停用词

词根化

生成邮件预处理文件

统计top1000词项

建立倒排表文件和词项、文档频率文件

建立tf_idf矩阵文件

清除无用文件

布尔检索

语义检索

建立tf_idf矩阵

预处理查询query

语义查询实现

整合的邮件检索系统

导入布尔检索和语义检索源代码

邮件检索系统

优化

调用库进行词根化处理

使用压缩矩阵

使用多线程方法

实验结果

布尔检索

语义检索

实验目的

以给定的邮件数据集为基础，实现一个邮件搜索引擎。对于给定的查询，能够以精确查询或模糊语义匹配的方式返回最相关的一系列邮件文档。

实验内容(算法)

按照老师课上所讲的搜索引擎建立流程，我们的工作可以分为以下几个步骤：

1. 对邮件数据集进行**预处理**。这一步骤包括搜索数据文件中每封邮件的路径、提取邮件标题内容并依次进行分词、去除停用词、词根化处理，打印至预处理文档中。
2. 统计每个单词的文档频率、词项频率等数据并依此建立总频率最高的1000个单词的**倒排索引表**和**tf-idf表**。
3. 根据倒排索引表和tf-idf表完成**布尔查询**和**语义查询**，实现搜索引擎功能。
 - 布尔检索：使用了两个栈分别存放**操作码**（and, not, or, 左右括号）和**操作数**（即查询单词），通过栈来实现布尔表达式的优先级计算(默认优先级：not > and > or)。
 - 语义检索：根据课上所学知识，利用 tf-idf 相关知识实现。
4. 为降低邮件搜索引擎所依赖的文件所消耗的空间，对邮件的单词进行**词根化处理**，极大地减少了单词数量；另一方面，对**词项频率**和**tf-idf矩阵**采用压缩矩阵的存储方法：以**二维压缩矩阵**形式存储，即每个子列表表示一个文档(一封邮件)，子列表内部，以下标0开始，下标为 2k 的列表元素为词项编号，而下标为 2k+1 的列表元素为该词项对应的词项频率或 tf_idf 值，极大节约了**存储**和**搜索代价**。
5. 为提高查询的加载效率(在整合的邮件检索系统中)，使用**多线程**方法，减少加载时间，对代码进一步优化。

实验条件

1. 硬件条件：一台PC机。
2. 软件条件：
 - 系统Windows10
 - 语言python3.8.5
 - 编辑器pycharm2018.3.4

使用算法及优化方式

预处理部分

获得所有文档路径

代码如下：

```
1 def find_file(path):      # Path represents the relative path
2     index = 0             # Index represents the subscript of a list element
3     file_list = os.listdir(path)    # 获得当前目录下的所有文件(文件夹)
4     for i in range(len(file_list)):
5         file = file_list[index]
6         relative_path = path + "/" + file # relative path
7         if os.path.isdir(relative_path):
8             subfile = find_file(relative_path) # deep search
9             for sub_file in subfile:
10                 file_list.append(sub_file)
11             del file_list[index]    # 删除非文件的列表元素，索引 index 不变
12         elif os.path.isfile(relative_path):
13             file_list[index] = relative_path
14             index = index + 1
15     return file_list
```

采用深度遍历算法，先查询当前目录下的所有文件，然后对文件进行判断，若为目录，则对该目录采用深度遍历，若为一封邮件，则将它的路径添加到列表中，本函数采用相对路径，从 path 开始深度遍历。

将结果(file_list)保存到 filelist_path 所指代的文件中(在 pretreatment() 函数中执行该操作):

```
1 # 打印所有文档路径
2 document_map = dict(zip(range(len(filelist)), filelist)) # 转化为字典
3 f = open(filelist_path, 'w')
4 handler = json.dumps(document_map) # 调用 json 库
5 f.write(handler)
6 f.close()
```

pretreatment.py 中的 pretreatment(filelist, target_path, filelist_path) 主要的任务是**提取邮件有用内容、分词和词根化**3部分，因此下面分3部分讲解该函数。

提取邮件有用内容

使用 re 库，通过正则表达式获取每封邮件中的标题和内容正文，由于有些邮件中不是以 utf-8 格式编码（即可能有中文），故在打开方式中，增加参数 errors='ignore'，忽略打开错误。

```
1 find_content = "\n\n(.*)" # 匹配内容正文
2 find_subject = "subject: (.*)\n" # 匹配标题
3 with open(target_path, 'w') as wfp:
4     for file in filelist:
5         fp = open(file, 'r', errors='ignore')
6         txt = fp.read().lower() # 将文本全转化为小写
7         fp.close()
8         subject = re.findall(find_subject, txt, re.M | re.I)[0] # 得
到主题
9         content = re.findall(find_content, txt, re.S)[0] # 得到邮件正
文内容
```

分词

分词前我们会对读取的正文内容做初步处理，即把**标点符号**和**数字**去除，为防止影响到之后的分词(数字的存在会干扰 top1000 词项的选取)，我们将其替换为空格。

```
1 string_replace = string.punctuation + '0123456789'
2 replacement = '' # 用 ' ' 替换 string_repalce 中的每个符号
3 for i in range(len(string_replace)):
4     replacement = replacement + ' '
```

```
1 remove = str.maketrans(string_replace, replacement) # 构造标点
符号和数字到空格的映射关系
2 content = content.translate(remove) # 去除标点
符号和数字
3 subject = subject.translate(remove)
```

原本我们使用 nltk 库中的 nltk.word_tokenize() 函数进行分词，但考虑到文档数目 51 万过大，且内容均为英文，为了提升预处理的效率，我们改为使用 re 库中的分割函数 split() 直接对空白字符进行切分。

```

1      # 由于使用nltk库分词会大大降低对51w封邮件的处理效率，改为使用split分词
2      # tokens = nltk.word_tokenize(content)      # 分词 spilt
3      # sub_tokens = nltk.word_tokenize(subject)
4      tokens = content.split()
5      sub_tokens = subject.split()

```

去除停用词

第一次尝试中原本想利用 `nltk` 库中的停用词表，如下：

```

1  import nltk
2  from nltk.corpus import stopwords

```

```

1      # 去除停用词
2      tokens_nosw = [word for word in tokens if word not in
stopwords.words('english')]
3      sub_tokens_nosw = [word for word in sub_tokens if word not in
stopwords.words('english')]

```

但是经过试验发现，采用停用词库会大大降低对51w封邮件的预处理效率（导致可能会运行一整天）。并且在语义查询时，如果只输入一句话，去除停用词后会有很大的误导性，导致语义查询出来的邮件中只有几个词和查询 `query` 相同，但语义完全不同，在权衡以后，**我们决定不采用停用词表**，这样的话，我们能将预处理的速度能缩减为2小时。

词根化

调用 `nltk.stem` 库中的 `SnowballStemmer('english')` 类，参数为选择的语言

```

1  import nltk.stem

```

```

1      stem = nltk.stem.SnowballStemmer('english') # 参数是选择的语言，用于词根化

```

```

1      tokens_stem = [stem.stem(ws) for ws in tokens]      # 词根化
2      sub_tokens_stem = [stem.stem(ws) for ws in sub_tokens]

```

生成邮件预处理文件

将预处理完的邮件内容保存到 `target_path` 所指的文件中：

```

1      content_fin = ' '.join(tokens_stem)      # 用空格拼接内容
2      subject_fin = ' '.join(sub_tokens_stem)
3      mail = subject_fin + ' ' + content_fin + '\n\n' + '-----Document
demarcation-----' + '\n\n'
# 以 '-----Document demarcation-----' 划分每个文档
4      wfp.write(mail)      # 写入预处理文档

```

利用邮件预处理文件，记录出现的词项和频率，取出 top1000 的词项，输出到 txt_2 文件中。

```

1  def get_top_1k_words(txt_1, txt_2):
2      global terms
3      wordlist = {}          # 记录出现过的词项和频率
4      fd = open(txt_1, 'r')
5      line = fd.readline()
6      while line:
7          if line != "-----Document demarcation-----\n": # 判断一封邮件是否结束
8              words = line.split()    # 分词
9              for word in words:
10                 if word in wordlist:
11                     wordlist[word] += 1    # 频率加1
12                 else:
13                     wordlist[word] = 1    # 记录出现的新词
14             line = fd.readline()
15         fd.close()
16         f2 = open(txt_2, 'w')          # 写入 txt_2 文档
17         for i in range(1000):
18             word_top = max(wordlist, key=wordlist.get)    # 用 max 取出字典中的
top1000 的键
19             terms.append(word_top)
20             f2.write(word_top + '\n')
21             wordlist[word_top] = -1    # 将已经取出的词项的频率置为 -1
22         f2.close()

```

建立倒排表文件和词项、文档频率文件

扫描邮件预处理文件，用列表 `count` 记录每篇文档中的词项频率，用列表 `flag` 标记每篇文档中出现的 top1000 词项是否已经添加到倒排表中。将词项频率写入 `out_doc_count_txt` 文件中，由于词项频率为稀疏矩阵，故只记录每篇文档出现过的 top1000 的词项的频率，存储形式为：“词项 对应的词项频率”。将倒排表写入 `path1` 指定的文件中。用 `doc_count` 统计文档频率，并写入 `path2` 指定的文件中。

[illegible]

```

18         else:
19             inve_table[word] = [doc_id]          # 若第一次出现该词，
则在字典中添加一个新的键值对
20             flag[terms.index(word)] = 1
21         else:
22             str_term_count = ''
23             for i in range(1000):
24                 if count[i] > 0:
25                     str_term_count = str_term_count + str(i + 1) + ' ' +
str(count[i]) + ' '
26             str_term_count = str_term_count + '\n'
27
28             wfd.write(str_term_count)          # 将每篇文档的词项频率写入
29
30             count = np.zeros(1000)
31             doc_id += 1
32             flag = np.zeros(1000)          # 重置 flag 列表，重新标记一封新的邮
件中是否出现top1000词项
33             line = fd.readline()
34             wfd.close()
35             fd.close()
36
37
38 # 将倒排表写入指定文件中，同时生成文档频率 doc_count，并同样写入指定文件中
39 def write_index_file(path1, path2):
40     global doc_count, inve_table
41     f = open(path1, 'w')
42     fd = open(path2, 'w')
43     for word, docIDS in inve_table.items():
44         doc_count[terms.index(word)] = len(docIDS)
45         indexing = word + '\t' + str(len(docIDS)) + '\t'
46         for docID in docIDS:
47             indexing += (' ' + str(docID))
48         f.write(indexing + '\n')
49     f.close()
50     map_doc_count = map(str, doc_count)
51     str_doc_count = ' '.join(map_doc_count)
52     fd.write(str_doc_count)
53     fd.close()

```

建立tf_idf矩阵文件

利用所学公式，计算每篇文档中出现过的 top1000 词项的tf_idf值，注意：这里词项频率我们没有建立二维全矩阵，而是采用了压缩矩阵的存储方法，以降低存储和搜索开销。在该文件中，存储形式为每一行对应一个文档(一封邮件)，并且按 "词项 词项频率" 形式存储，为了提高速率，我们计算tf_idf值时要计算词项的下标以及该词项对应的词项频率的下标。tf_idf 矩阵文件也采用压缩矩阵存储形式 (tf_idf 矩阵也是个稀疏矩阵)，即每一行对应一个文档(一封邮件)，并且按 "词项 对应的tf_idf值" 形式存储。

```

1 def doc_tf_idf(n, tc_path, output_txt): # n 表示文档总数
2     global doc_count
3     fd = open(tc_path, 'r')          # tc_path 为记录词项频率的文件的途径
4     line = fd.readline()
5     wfd = open(output_txt, 'w')
6     while line:

```

```

7         doc_term = line.split()
8         nums = int(len(doc_term)/2) # 计算词项数量(存储形式: "词项 词项频率 ",即从
下标0开始,偶数项为词项,奇数项为词项频率)
9         element = ''
10        for term_id in range(nums):          # 只记录了词项频率非0的词项, 因此
tf 不可能为 0
11            if doc_count[int(doc_term[2*term_id]) - 1] != n: # 等于 n 的话,
tf_idf 的值就为 0, 不需要存储
12                tf_idf = round((1 + math.log(float(doc_term[2*term_id+1]),
10)) * math.log(n/doc_count[int(doc_term[2*term_id]) - 1], 10), 6) #
计算该文档中出现过的top1000词项的tf_idf值
13                element = element + str(doc_term[2*term_id]) + ' ' +
str(tf_idf) + ' '
14            element = element + '\n'
15            wfd.write(element)
16            line = fd.readline()
17        wfd.close()
18        fd.close()

```

清除无用文件

上面一共建立了7个文件, 当所有文件建立完以后, **邮件预处理文件**和**词项频率文件**就没用了, 可以手动删除以节省空间。

布尔检索

布尔搜索需要先读取倒排索引表。

```

1 # 从文件中读取索引创建索引字典
2 def create_index(file_name):
3     global inverse_index
4     f = open(file_name, 'r')
5     line = f.readline().rstrip()
6     while line:
7         line = re.split(r'\s+', line)
8         inverse_index[line[0]] = set() # 用集合表示含有该词项的文档集合
9         for id in line[2:]: # 0 为 词项, 1 为文档频率, 2 之后为文档编号
10             inverse_index[line[0]].add(id)
11         line = f.readline().rstrip()
12     return inverse_index

```

随后对于每次输入的搜索语句, 计算其布尔逻辑运算结果, 得到返回文档序列(在 `search_index(keywords, doc_map, n)` 函数中)。我们使用了两个栈分别存放**操作码** (and, not, or, 左右括号) 和**操作数** (即查询单词), 通过栈来实现布尔表达式的优先级计算(默认优先级: not > and > or)。

```

1     global inverse_index
2     opcode = [] # 用opcode列表充当操作码栈
3     operand = [] # 用operand列表充当操作数栈
4     full = set()
5     for id in range(n): #构建全集

```

```

6         full.add(str(id))
7     for token in keywords:
8         # 如果读取左括号或not, 直接入栈
9         if token == '(' or token == 'not':
10             opcode.append(token)
11         # 如果读取运算符and, 判断栈顶运算符是否为and, 若是则计算, 若不是则直接入栈
12         elif token == 'and':
13             if len(opcode) and opcode[len(opcode) - 1] == 'and':
14                 preval = operand.pop()
15                 pre2val = operand.pop()
16                 operand.append(preval & pre2val)
17             else:
18                 opcode.append(token)
19         # 如果读取or, 因为其优先级较低, 在栈顶为and或or时都先计算栈顶运算再入栈
20         elif token == 'or':
21             if len(opcode) and opcode[len(opcode) - 1] == 'and':
22                 opcode.pop()
23                 preval = operand.pop()
24                 pre2val = operand.pop()
25                 operand.append(preval & pre2val)
26                 opcode.append(token)
27             elif len(opcode) and opcode[len(opcode) - 1] == 'or':
28                 preval = operand.pop()
29                 pre2val = operand.pop()
30                 operand.append(preval | pre2val)
31             else:
32                 opcode.append(token)
33         # 若读取到右括号, 则不断循环计算栈顶运算, 直到遇到匹配的左括号为止
34         elif token == ')':
35             preop = opcode.pop()
36             while preop != '(':
37                 preval = operand.pop()
38                 pre2val = operand.pop()
39                 if preop == 'and':
40                     operand.append(preval & pre2val)
41                 else:
42                     operand.append(preval | pre2val)
43                 preop = opcode.pop()
44         # 计算完括号后, 若栈顶为not, 直接计算
45         if len(opcode) and opcode[len(opcode) - 1] == 'not':
46             opcode.pop()
47             preval = operand.pop()
48             operand.append(full - preval)
49         # 若读取的为单词, 仅在栈顶为not时直接计算, 否则直接入栈
50         else:
51             if len(opcode) and opcode[len(opcode) - 1] == 'not':
52                 opcode.pop()
53                 operand.append(full - inverse_index.get(token, set()))
54             else:
55                 operand.append(inverse_index.get(token, set()))
56     # 计算最后可能剩下的二元表达式
57     while len(opcode):
58         preop = opcode.pop()
59         preval = operand.pop()
60         pre2val = operand.pop()
61         if preop == 'and':
62             operand.append(preval & pre2val)
63         else:

```



```

64         operand.append(preval | pre2val)
65     ids = operand.pop()
66     print_result(ids, doc_map)

```

需要注意的是，对于输入的查询语句，我们同样会先对其进行词根化处理，来和倒排表中单词匹配。

```

1     stem = nltk.stem.SnowballStemmer('english')
2     keywords = [stem.stem(ws) for ws in keywords]

```

布尔检索主函数：

```

1 def bool_search():
2     inve_txt = '../output/inverse_index.txt'
3     filelist_path = '../output/doc_map.txt'
4     create_index(inve_txt) # 创建索引表
5     doc_map = create_map(filelist_path) # 创建文档路径字典
6     total = len(doc_map) # 文档(邮件)总数
7     while True:
8         query = input('请输入布尔查询的内容: ')
9         query = query.lower()
10        # 识别替换括号，去除两侧空白字符
11        query = re.sub('\(', ' ( ', query)
12        query = re.sub('\)', ' ) ', query).strip()
13        # 切词
14        keywords = re.split(r'\s+', query)
15        if keywords:
16            search_index(keywords, doc_map, total)
17        else:
18            print("Input does not contain keywords!")

```

语义检索

建立tf_idf矩阵

以**二维矩阵**形式存储，每个子列表表示一个文档(一封邮件)，子列表内部，以下标0开始，下标为 2k 的列表元素为词项编号，而下标为 2k+1 的列表元素为该词项对应的 tf_idf 值，大大缩小矩阵存储开销，也减少了无用计算。结果存储在全局列表 dtifs 中。

```

1 def get_tf_idf(path):
2     global dtifs
3     fd = open(path, 'r')
4     line = fd.readline()
5     while line:
6         if line != '\n':
7             doc_tf = line.split()
8             dtifs.append(doc_tf)
9         else: # 为 '\n' 表示该邮件中，top1000词项都没有出现
10            doc_tf = []
11            dtifs.append(doc_tf)
12        line = fd.readline()
13    fd.close()

```

预处理查询query

我们所支持的语义查询是任意输入一行语句，可以包括**任意的标点符号**，搜索相关度前10的文档(邮件)，因此要对查询 query 做预处理：转化为**全小写**、**替换标点符号和数字**、**词根化**和**分词**。

```
1 def query_process(query):
2     # 对查询做预处理
3     query = query.lower() # 转化为小写
4     string_replace = string.punctuation + '0123456789'
5     replacement = '' # 用 ' ' 替换 string_replace 中的每个符号
6     for i in range(len(string_replace)):
7         replacement = replacement + ' '
8     remove = str.maketrans(string_replace, replacement) # 去除标点符号和数字
9     query = query.translate(remove)
10    words = query.split() # 分词
11    stem = nltk.stem.SnowballStemmer('english') # 参数是选择的语言，用于词根化
12    tokens_stem = [stem.stem(ws) for ws in words] # 词根化
13    return tokens_stem
```

语义查询实现

语义查询主要在 `semantic_search(query, n, terms, doc_counts)` 函数中实现，下面是他的每一个部分。

根据老师课上讲的原理，对查询语句先计算出查询query中top1000词项的tf-idf值，形成tf-idf向量：

```
1 vec_q = np.zeros(1000) # 初始化
2 for term in query:
3     if term in terms:
4         vec_q[terms.index(term)] += 1.0 # 统计每个词在查询query中的出现次数
5 for i in range(len(vec_q)):
6     if vec_q[i] != 0:
7         vec_q[i] = round((1 + math.log(vec_q[i], 10)) *
8 math.log(n/float(doc_counts[i]), 10), 10)
9         # 计算查询query中top1000词项的
tf_idf 值
10 vec_q_np = np.array(vec_q) # 转化为 numpy库的数组
```

得到每篇文档对应的tf-idf向量(`dtifs` 采用压缩矩阵的存储方法：以**二维矩阵**形式存储，每个子列表表示一个文档(一封邮件)，子列表内部，以下标0开始，下标为 $2k$ 的列表元素为词项编号，而下标为 $2k+1$ 的列表元素为该词项对应的 `tf_idf` 值)：

```
1 global dtifs
```

```
1 for tds in dtifs:
2     vec_doc = np.zeros(1000)
3     nums = int(len(tds)/2)
4     for term_id in range(nums):
5         vec_doc[int(tds[2 * term_id]) - 1] = float(tds[2 * term_id + 1])
6     vec_doc_np = np.array(vec_doc)
```

用查询query的tf-idf向量与各个文档tf-idf向量的做内积运算，并做归一化，得到相关度列表relevancy。

```
1 value = np.sqrt((vec_q_np * vec_q_np).sum() * (vec_doc_np *  
vec_doc_np).sum()) # 归一化系数  
2 if value != 0:  
3     cosine = ((vec_q_np * vec_doc_np).sum())/value # 求相关度  
4 else:      # 分母为 0 表示查询query或该文档中午top1000词项出现，相关度自然为  
0  
5     cosine = 0  
6     relevancy.append(cosine) # 求得相关度的列表
```

得到相关度top10的文档的编号及其相关度：

```
1 for i in range(10):  
2     index = relevancy.index(max(relevancy))  
3     doc.append(index)  
4     relevancy_top10.append(relevancy[index])  
5     relevancy[index] = -1 # -1 表示已经用完,不对后面的分析产生影响  
6 return doc, relevancy_top10
```

语义查询主函数：

```
1 def semantic():  
2     filelist_path = '../output/doc_map.txt'  
3     doc_count_txt = '../output/doc_count.txt'  
4     tf_idf_txt = '../output/tf_idf.txt'  
5     terms_txt = '../output/top1kwords.txt'  
6  
7     print("Please wait a few minutes for loading...")  
8  
9     doc_map = create_map(filelist_path) # 创建映射  
10    doc_count = get_doc_count(doc_count_txt) # 创建文档频率 doc_counts  
11    terms = get_terms(terms_txt) # 创建 top1000 的词项  
12    get_tf_idf(tf_idf_txt) # 读取 tf_idf 文档  
13    n = len(doc_map) # 文档总数  
14  
15    while True:  
16        query = input('请输入语义查询的内容:')  
17        tokens_stem = query_process(query)  
18        doc, relevancy_top10 = semantic_search(tokens_stem, n, terms,  
doc_count)  
19        print('The top 10 most relevant documents:')  
20        for i in range(len(doc)):  
21            print(i+1, ':', doc_map[str(doc[i])], "\t\t relevancy: ",  
relevancy_top10[i])
```

整合的邮件检索系统

导入布尔检索和语义检索源代码

```
1 import semantic_search as semantic
2 import bool_search
```

邮件检索系统

邮件检索系统采用了多线程优化加载 loading 速率。

```
1 import re
2 import threading
3
4 def main():
5     filelist_path = '../output/doc_map.txt'
6     doc_count_txt = '../output/doc_count.txt'
7     tf_idf_txt = '../output/tf_idf.txt'
8     inve_txt = '../output/inverse_index.txt'
9     terms_txt = '../output/top1kwords.txt'
10
11     print("Please wait a few minutes for loading...")
12
13     t1 = threading.Thread(target=semantic.get_tf_idf, args=(tf_idf_txt,)) #
14     # 创建线程 1
15     t1.start() # 开始执行线程 1
16     t2 = threading.Thread(target=bool_search.create_index, args=(inve_txt,))
17     # 创建线程 2
18     t2.start() # 开始执行线程 2
19
20     doc_map = semantic.create_map(filelist_path) # 创建映射
21     doc_count = semantic.get_doc_count(doc_count_txt) # 创建文档频率
22     # doc_counts
23     terms = semantic.get_terms(terms_txt) # 创建 top1000 的词汇
24
25     t1.join() # 等待线程 1 和 2 执行完
26     t2.join()
27
28     n = len(doc_map) # 文档总数
29
30     print("The retrieval method is as follows:")
31     print("0) quit 1) bool search 2) semantic search")
32     choose = input("Please enter the retrieval method:")
33     while choose != '0':
34         if choose == '1':
35             query = input('请输入布尔查询的内容:')
36             query = query.lower()
37             # 识别替换括号, 去除两侧空白字符
38             query = re.sub('\(', ' ( ', query)
39             query = re.sub('\)', ' ) ', query).strip()
40             # 切词
41             keywords = re.split(r'\s+', query)
42             if keywords:
43                 bool_search.search_index(keywords, doc_map, n)
44             else:
```

```

42         print("Input does not contain keywords!")
43
44     elif choose == '2':
45         query = input('请输入语义查询的内容:')
46         tokens_stem = semantic.query_process(query)      # 预处理查询query
47         doc, relevancy_top10 = semantic.semantic_search(tokens_stem, n,
48 terms, doc_count)  # 得到相关度top10的文档编号及其相关度
49         print('\nThe top 10 most relevant documents:')
50         for i in range(len(doc)):
51             print(i+1, ':', doc_map[str(doc[i])], "\t\t relevancy: ",
52 relevancy_top10[i])
53             print('\n')
54
55     else:
56         print("\nInput ERROR!!!\n")
57
58     print("The retrieval method is as follows:")
59     print("0) quit    1) bool search    2) semantic search")
60     choose = input("Please enter the retrieval method again:")

```

优化

调用库进行词根化处理

我们在词根化的步骤中，调用了 `nltk` 库中的数据和函数(由于用上停用词表的效果并不好，因此不用停用词表)，词根化大大减少了单词数量，减少存储数据。

```

1 import nltk
2 # from nltk.corpus import stopwords
3 import nltk.stem

```

```

1 # 词根化
2 tokens_stem = [stem.stem(ws) for ws in tokens]
3 sub_tokens_stem = [stem.stem(ws) for ws in sub_tokens]

```

使用压缩矩阵

在计算 `tf-idf` 矩阵时，我们没有用二维矩阵存储各文档中1000个单词的频率(即频率为0也存储)，而是仅统计了出现在文档中单词的频率，同时对于词项编号和对应`tf-idf`值，也没有建立三级列表或二元组来存储来绑定两者(这会增加极大的开销)，而是直接以二维压缩矩阵形式存储，即每个子列表表示一个文档(一封邮件)，子列表内部，以下标0开始，下标为 $2k$ 的列表元素为词项编号，而下标为 $2k+1$ 的列表元素为该词项对应的 `tf-idf` 值，极大节约了存储和搜索代价。

```

1 str_term_count = ''
2 for i in range(1000):
3     if count[i] > 0:
4         str_term_count = str_term_count + str(i + 1) + ' ' +
5 str(count[i]) + ' '
6 str_term_count = str_term_count + '\n'

```

```

1     nums = int(len(doc_term)/2)
2     element = ''
3     for term_id in range(nums):          # 只存储词项频率非0的top1000词项，因此
tf 不可能为 0
4         if doc_count[int(doc_term[2*term_id]) - 1] != n:
5             # 等于 n 的话，tf_idf 的值就为 0，不需要存储
6             tf_idf = round((1 + math.log(float(doc_term[2*term_id+1]), 10)) *
math.log(n/float(doc_count[int(doc_term[2*term_id]) - 1]), 10), 6)
7             element = element + str(doc_term[2*term_id]) + ' ' + str(tf_idf)
+ ' '

```

```

1     for tds in dtifs:
2         vec_doc = np.zeros(1000)
3         nums = int(len(tds)/2)
4         for term_id in range(nums):
5             vec_doc[int(tds[2 * term_id]) - 1] = float(tds[2 * term_id + 1])

```

经过测试，不用压缩矩阵存储时，搜索时间大约为2分钟左右，而采用压缩矩阵存储时，搜索时间仅需30s左右。

源代码：

```

1 import datetime

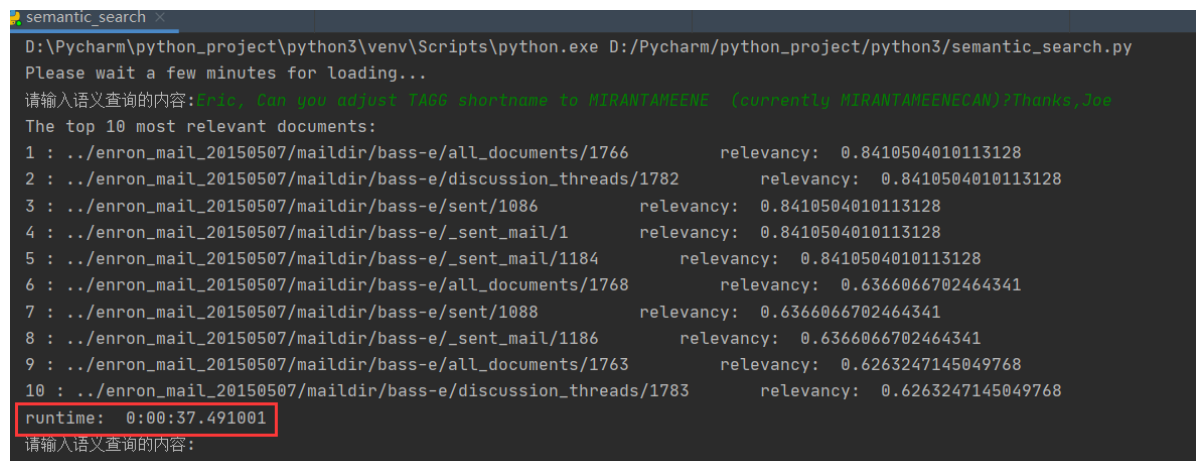
```

```

1     while True:
2         query = input('请输入语义查询的内容:')
3         start = datetime.datetime.now()
4         tokens_stem = query_process(query)
5         doc, relevancy_top10 = semantic_search(tokens_stem, n, terms,
doc_count)
6         print('The top 10 most relevant documents:')
7         for i in range(len(doc)):
8             print(i+1, ': ', doc_map[str(doc[i])], "\t\t relevancy: ",
relevancy_top10[i])
9         end = datetime.datetime.now()
10        print("runtime: ", end - start)

```

测试结果如下：



```

semantic_search x
D:\Pycharm\python_project\python3\venv\Scripts\python.exe D:/Pycharm/python_project/python3/semantic_search.py
Please wait a few minutes for loading...
请输入语义查询的内容: Eric, Can you adjust TAGS shorthand to MIRANTAREERE (currently MIRANTAREEELAN)? Thanks, Joe
The top 10 most relevant documents:
1 : ../enron_mail_20150507/maildir/bass-e/all_documents/1766      relevancy: 0.8410504010113128
2 : ../enron_mail_20150507/maildir/bass-e/discussion_threads/1782  relevancy: 0.8410504010113128
3 : ../enron_mail_20150507/maildir/bass-e/sent/1086      relevancy: 0.8410504010113128
4 : ../enron_mail_20150507/maildir/bass-e/_sent_mail/1      relevancy: 0.8410504010113128
5 : ../enron_mail_20150507/maildir/bass-e/_sent_mail/1184      relevancy: 0.8410504010113128
6 : ../enron_mail_20150507/maildir/bass-e/all_documents/1768      relevancy: 0.6366066702464341
7 : ../enron_mail_20150507/maildir/bass-e/sent/1088      relevancy: 0.6366066702464341
8 : ../enron_mail_20150507/maildir/bass-e/_sent_mail/1186      relevancy: 0.6366066702464341
9 : ../enron_mail_20150507/maildir/bass-e/all_documents/1763      relevancy: 0.6263247145049768
10 : ../enron_mail_20150507/maildir/bass-e/discussion_threads/1783  relevancy: 0.6263247145049768
runtime: 0:00:37.491001
请输入语义查询的内容:

```

可见，利用压缩矩阵的效果很好。

使用多线程方法

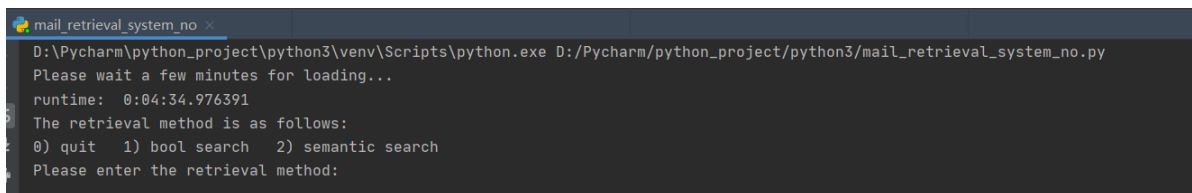
由于整合的邮件检索系统需要加载**倒排表文件**和**tf-idf矩阵文件**，这两个文件的串行加载将会花费很多时间(大约为4分半左右)，故采用多线程优化加载速度，优化后的加载速度大约为4分钟，优化了约等于10%。

```
1      t1 = threading.Thread(target=semantic.get_tf_idf, args=(tf_idf_txt,))
2      # 创建线程 1
3      t1.start() # 开始执行线程 1
4      t2 = threading.Thread(target=bool_search.create_index, args=(inve_txt,))
5      # 创建线程 2
6      t2.start()      # 开始执行线程 2
7
8      doc_map = semantic.create_map(filelist_path) # 创建映射
9      doc_count = semantic.get_doc_count(doc_count_txt) # 创建文档频率
10     doc_counts
11
12     terms = semantic.get_terms(terms_txt) # 创建 top1000 的词汇
13
14     t1.join()      # 等待线程 1 和 2 执行完
15     t2.join()
```

未使用多线程的源代码：

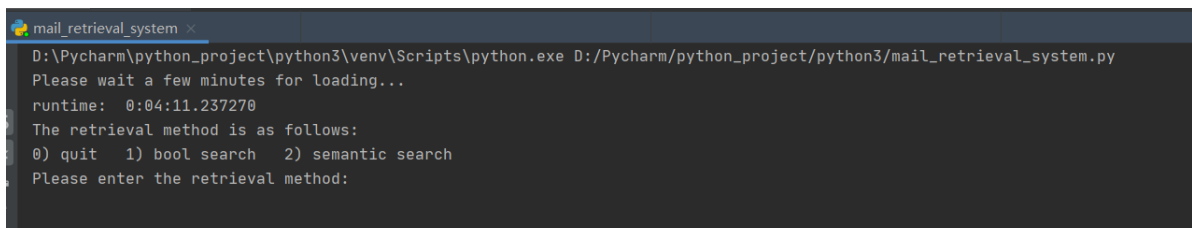
```
1      start = datetime.datetime.now()
2
3      doc_map = semantic.create_map(filelist_path) # 创建映射
4      doc_count = semantic.get_doc_count(doc_count_txt) # 创建文档频率
5      doc_counts
6
7      terms = semantic.get_terms(terms_txt) # 创建 top1000 的词汇
8      index = bool_search_no.create_index(inve_txt)
9      dtifs = semantic.get_tf_idf(tf_idf_txt) # 读取 tf_idf 文档
10
11     end = datetime.datetime.now()
12     print("runtime: ", end - start)
```

未使用多线程的加载的运行时间为：



The screenshot shows a terminal window titled 'mail_retrieval_system_no'. It displays the execution of a Python script. The output includes the path 'D:\Pycharm\python_project\python3\venv\Scripts\python.exe D:\Pycharm\python_project\python3\mail_retrieval_system_no.py', a loading message, and the runtime '0:04:34.976391'. It also shows the retrieval method options: '0) quit 1) bool search 2) semantic search'.

使用多线程的加载的运行时间为：



The screenshot shows a terminal window titled 'mail_retrieval_system'. It displays the execution of a Python script. The output includes the path 'D:\Pycharm\python_project\python3\venv\Scripts\python.exe D:\Pycharm\python_project\python3\mail_retrieval_system.py', a loading message, and the runtime '0:04:11.237270'. It also shows the retrieval method options: '0) quit 1) bool search 2) semantic search'.

实验结果

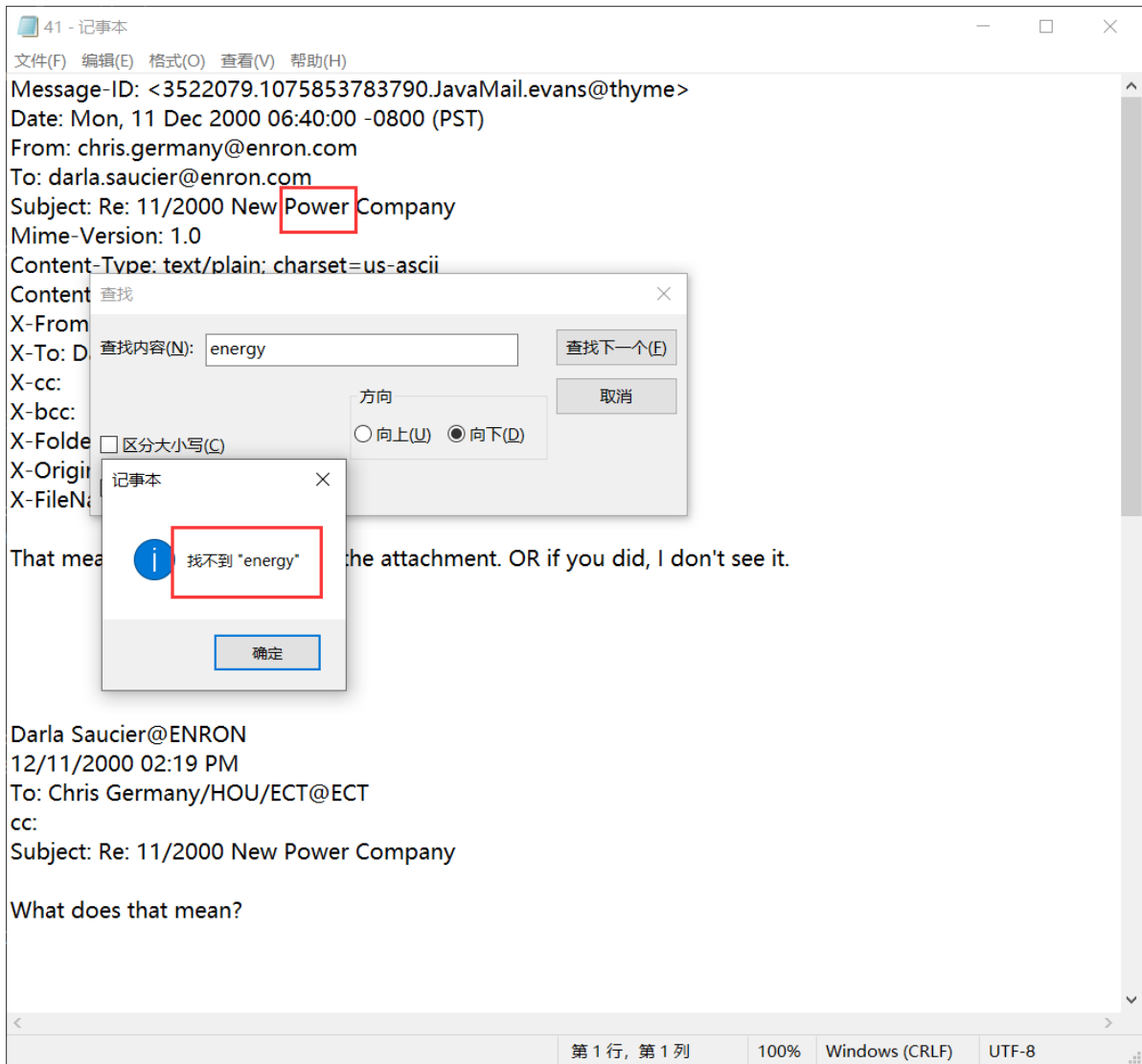
布尔检索

查询: (power or business) and not energy

结果:

```
semantic_search x bool_search x
73964: ../enron_mail_20150507/maildir/delahey-d/discussion_threads/1266
73965: ../enron_mail_20150507/maildir/stokley-c/chris_stokley/sent/120
73966: ../enron_mail_20150507/maildir/buy-r/inbox/689
73967: ../enron_mail_20150507/maildir/taylor-m/archive/8_00/146
73968: ../enron_mail_20150507/maildir/hain-m/all_documents/1090
73969: ../enron_mail_20150507/maildir/lay-k/all_documents/910
73970: ../enron_mail_20150507/maildir/whalley-g/sent_items/50
73971: ../enron_mail_20150507/maildir/carson-m/inbox/5
73972: ../enron_mail_20150507/maildir/kitchen-l/_americas/portland/141
73973: ../enron_mail_20150507/maildir/dasovich-j/all_documents/8287
73974: ../enron_mail_20150507/maildir/shackleton-s/sent/5621
73975: ../enron_mail_20150507/maildir/arnold-j/inbox/86
73976: ../enron_mail_20150507/maildir/jones-t/all_documents/10710
73977: ../enron_mail_20150507/maildir/mann-k/sent/3799
73978: ../enron_mail_20150507/maildir/lewis-a/deleted_items/800
73979: ../enron_mail_20150507/maildir/beck-s/discussion_threads/1778
73980: ../enron_mail_20150507/maildir/germany-c/sent/41
```

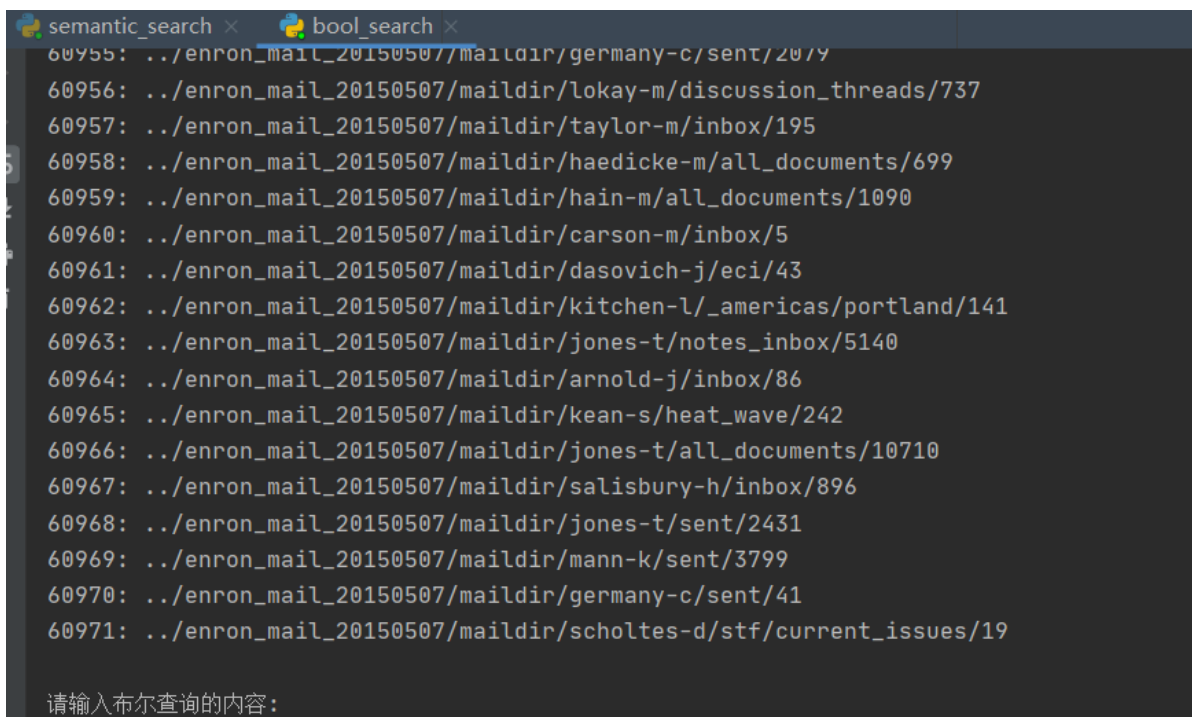
共73980篇符合条件, 检索最后一封观察是否真实符合条件, 结果如下:



可见确实符合条件。

查询: `powerful`

结果:



查询: `not powerful`

结果：

```
semantic_search ×  bool_search ×
456414: ../enron_mail_20150507/maildir/kitchen-l/_americas/regulatory/51
456415: ../enron_mail_20150507/maildir/forney-j/sent_items/171
456416: ../enron_mail_20150507/maildir/taylor-m/all_documents/2623
456417: ../enron_mail_20150507/maildir/scott-s/_sent_mail/568
456418: ../enron_mail_20150507/maildir/kean-s/archiving/untitled/6183
456419: ../enron_mail_20150507/maildir/skilling-j/all_documents/86
456420: ../enron_mail_20150507/maildir/jones-t/sent/5595
456421: ../enron_mail_20150507/maildir/rogers-b/deleted_items/378
456422: ../enron_mail_20150507/maildir/blair-l/sent_items/906
456423: ../enron_mail_20150507/maildir/stokley-c/chris_stokley/iso/mrkt_info/128
456424: ../enron_mail_20150507/maildir/dasovich-j/all_documents/2128
456425: ../enron_mail_20150507/maildir/shapiro-r/discussion_threads/490
456426: ../enron_mail_20150507/maildir/harris-s/deleted_items/131
456427: ../enron_mail_20150507/maildir/kaminski-v/discussion_threads/1413
456428: ../enron_mail_20150507/maildir/jones-t/all_documents/9841
456429: ../enron_mail_20150507/maildir/jones-t/notes_inbox/3337
456430: ../enron_mail_20150507/maildir/beck-s/discussion_threads/1778

请输入布尔查询的内容：
```

`powerful` (60971封)和 `not powerful` (456430封)一共 517401 封刚好是邮件总数，符合预期

布尔查询还有很多例子，经测验均圆满完成任务。

语义检索

第一个例子：

```

1  Message-ID: <17027752.1075840325838.JavaMail.evans@thyme>
2  Date: Fri, 9 Mar 2001 11:24:00 -0800 (PST)
3  From: eric.bass@enron.com
4  To: chance.rabon@enron.com
5  Subject: Rebook - QU0663 Mirant
6  Mime-Version: 1.0
7  Content-Type: text/plain; charset=us-ascii
8  Content-Transfer-Encoding: 7bit
9  X-From: Eric Bass
10 X-To: Chance Rabon <Chance.Rabon@enron.com>
11 X-cc:
12 X-bcc:
13 X-Folder: \ExMerge - Bass, Eric\'Sent Mail
14 X-Origin: BASS-E
15 X-FileName: eric bass 6-25-02.PST
16
17
18 ----- Forwarded by Eric Bass/HOU/ECT on 03/09/2001 08:24 AM -----
19
20 | From: Larry Joe Hunter                                03/08/2001 05:53 PM
21 |
22 |
23 |
24 | To: Eric Bass/HOU/ECT@ECT
25 | cc: Janie Aguayo/HOU/ECT@ECT
26 | Subject: Rebook - QU0663 Mirant
27 |
28 | Eric,
29 |
30 | Can you adjust TAGG shortname to MIRANTAMEENE (currently MIRANTAMEENECAN)?
31 |
32 | Thanks,
33 | Joe
34 |
35 |

```

以红框中的内容为检索对象，进行检索，检索结果如下如图所示：

```

D:\Pycharm\python_project\python3\venv\Scripts\python.exe D:\Pycharm\python_project\python3\semantic_search.py
Please wait a few minutes for loading...
请输入语义查询的内容:Eric, Can you adjust TAGG shortname to MIRANTAMEENE (currently MIRANTAMEENECAN)?Thanks, Joe
The top 10 most relevant documents:
1 : ../enron_mail_20150507/maildir/bass-e/all_documents/1766      relevancy: 0.8410504010113128
2 : ../enron_mail_20150507/maildir/bass-e/discussion_threads/1782  relevancy: 0.8410504010113128
3 : ../enron_mail_20150507/maildir/bass-e/sent/1086      relevancy: 0.8410504010113128
4 : ../enron_mail_20150507/maildir/bass-e/_sent_mail/1      relevancy: 0.8410504010113128
5 : ../enron_mail_20150507/maildir/bass-e/_sent_mail/1184      relevancy: 0.8410504010113128
6 : ../enron_mail_20150507/maildir/bass-e/all_documents/1768      relevancy: 0.6366066702464341
7 : ../enron_mail_20150507/maildir/bass-e/sent/1088      relevancy: 0.6366066702464341
8 : ../enron_mail_20150507/maildir/bass-e/_sent_mail/1186      relevancy: 0.6366066702464341
9 : ../enron_mail_20150507/maildir/bass-e/all_documents/1763      relevancy: 0.6263247145049768
10 : ../enron_mail_20150507/maildir/bass-e/discussion_threads/1783  relevancy: 0.6263247145049768

```

可以看到**第4封邮件**就是检索的对象，相关度为 0.84，另一方面，可以看到前5篇相关度一样，可以合理推测两封邮件内容大致一致，用第一遍邮件验证该想法：

```
1  Message-ID: <4708328.1075854734591.JavaMail.evans@thyme>
2  Date: Fri, 9 Mar 2001 00:24:00 -0800 (PST)
3  From: eric.bass@enron.com
4  To: chance.rabon@enron.com
5  Subject: Rebook - QU0663 Mirant
6  Mime-Version: 1.0
7  Content-Type: text/plain; charset=us-ascii
8  Content-Transfer-Encoding: 7bit
9  X-From: Eric Bass
10 X-To: Chance Rabon
11 X-cc:
12 X-bcc:
13 X-Folder: \Eric_Bass_Jun2001\Notes Folders\All documents
14 X-Origin: Bass-E
15 X-FileName: ebass.nsf
16
17 ----- Forwarded by Eric Bass/HOU/ECT on 03/09/2001 08:24 AM
18 -----
19
20
21
22 From: Larry Joe Hunter                                03/08/2001 05:53 PM
23
24
25 To: Eric Bass/HOU/ECT@ECT
26 cc: Janie Aguayo/HOU/ECT@ECT
27 Subject: Rebook - QU0663 Mirant
28
29 Eric,
30
31 Can you adjust TAGG shortname to MIRANTAMEENE (currently MIRANTAMEENECAN)?
32
33 Thanks,
34 Joe
35
36
```

说明邮件中有许多内容相同的重复文档。

第二个例子:

仅查询一封大邮件里的一小段，降低相关度，观察效果如何。

查询的内容如下(仅为这封大邮件里的一小段):

```

84
85 East Coast
86 NEPOOL                90.50      +0.50      90.50      90.50
87 New York Zone J       91.00      +7.50      90.00      92.00
88 New York Zone G       74.50      +1.00      73.50      75.50
89 New York Zone A       68.83      +6.13      65.50      73.50
90 PJM                    67.08      -12.75     62.00      76.00
91 East                   67.08      -12.75     62.00      76.00
92 West                   67.08      -12.75     62.00      76.00
93 Seller's Choice       66.58      -12.75     61.50      75.50
94 End Table
95
96
97 Western Power Prices Fall With Warmer Weather, Natural Gas Loss
98
99 Los Angeles, Dec. 13 (Bloomberg Energy) -- U.S. Western spot
100 power prices declined today from a combination of warmer weather
101 across the region and declining natural gas prices.
102 According to Belton, Missouri-based Weather Derivatives Inc.,
103 temperatures in the Pacific Northwest will average about 2 degrees
104 Fahrenheit above normal for the next seven days. In the Southwest
105 temperatures will be about 3.5 degree above normal.
106 At the California-Oregon Border, heavy load power fell
107 $172.00 from yesterday to $430.00-$475.00.
108 "What happened to all of this bitter cold weather we were
109 supposed to have," said one Northwest power marketer. Since the
110 weather is not as cold as expected prices are drastically lower."
111 Temperatures in Los Angeles today will peak at 66 degrees,
112 and are expected to rise to 74 degrees this weekend.
113 Natural gas to be delivered to the California Oregon from the
114 El Paso Pipeline traded between $20-$21, down $3 from yesterday.
115 "Gas prices are declining causing western daily power prices
116 to fall," said one Northwest power trader.
117 At the NP-15 delivery point heavy load power decreased
118 $206.25 from yesterday to $325.00-$450.00. Light load energy fell
119 to $200.00-$350.00, falling $175.00 from yesterday.
120 PSC of New Mexico's 498-megawatt San Juan Unit 4 coal plant

```

查询结果为:

```

请输入语义查询的内容: According to Belton, Missouri-based Weather Derivatives Inc., temperatures in the Pacific Northwest will average about 2 degrees
next seven days. In the Southwest temperatures will be about 3.5 degrees above normal. At the California-Oregon Border, heavy load power fell $
$206.25. "What happened to all of this bitter cold weather we were supposed to have," said one Northwest power marketer. Since the
The top 10 most relevant documents:
1 : ../enron_mail_20150507/maildir/allen-p/all_documents/10      relevancy: 0.3704641762640625
2 : ../enron_mail_20150507/maildir/allen-p/discussion_threads/224      relevancy: 0.3704641762640625
3 : ../enron_mail_20150507/maildir/allen-p/notes_inbox/10      relevancy: 0.3704641762640625
4 : ../enron_mail_20150507/maildir/neal-s/deleted_items/284      relevancy: 0.3539068883743298
5 : ../enron_mail_20150507/maildir/neal-s/deleted_items/375      relevancy: 0.3521582418297442
6 : ../enron_mail_20150507/maildir/neal-s/deleted_items/362      relevancy: 0.34925247348032096
7 : ../enron_mail_20150507/maildir/jones-t/all_documents/280      relevancy: 0.3425605522911785
8 : ../enron_mail_20150507/maildir/jones-t/sent/279      relevancy: 0.3425605522911785
9 : ../enron_mail_20150507/maildir/neal-s/deleted_items/394      relevancy: 0.33847975277669673
10 : ../enron_mail_20150507/maildir/neal-s/inbox/172      relevancy: 0.33734797877471423
请输入语义查询的内容:

```

虽然相关度很低(仅为0.37), 但该封邮件的排名很高, 可见, 语义查询实现的效果较好。