邮件检索系统

计算机科学与技术专业　　1711436　　皮春莹

1. 项目信息

1.1 实验环境

(1) Python 3.6
(2) JetBrains PyCharm Community Edition 2019.1.3
(3) 用到的 python 包：codecs，os，json，email，nltk，re，math，numpy，tkinter。

1.2 功能简介

用 python 编写程序，按照向量空间模型，对安然公司的 517401 封电子邮件建立倒排索引。用户可以输入收件人、发件人、标题、正文位置想搜索的多个单词，系统将返回相似度最高的前 N 封邮件的存储路径（N 的值由用户指定）。

2. 代码部分

2.1 实现思路

本次实验主要分为构建索引和查询两个部分：

2.1.1　构建索引

(1) 为了提高查询速度、减少存储空间，需要将每封邮件的存储路径对应到一个文章 ID，用字典结构存储这个信息（ID.json），key 为文章 ID，value 为文档路径，按照文章 ID 排序。
(2) 对五十多万封邮件建立的索引，要包含的信息有单词内容、文档频率、单词位置及对应词频。我们需要用一个字典结构存储文档频率（DF.json），key 为处理后的单词，value 为该单词的文档频率，按照单词的 ASCII 码排序。另外，用字典结构存储出现过这个单词的文档以及对应词频（TF.json），key 为处理后的单词，value 是一个列表，列表中的每一项是一个元组（p , q），p 是文章 ID，q 是词频，字典按照单词的 ASCII 码排序，由于文章 ID 是递增的，因此列表中的元组也就是按照文章 ID 递增排好序的。
(3) 遍历文件夹下所有邮件，给每封邮件赋予一个独特的文章 ID，用 email 中已

有的函数，提取发件人(From)、收件人(To)、主题(Subject)和邮件正文的单词。在将邮件中的单词存入索引之前，需要先做一些处理：

a) 邮件的发件人和收件人格式固定且简单，不用做过多的处理，用逗号直接分割。邮件主题和内容中存在特殊符号、大小写、不同时态、单复数等，用正则表达式过滤特殊符号，调用 nltk 包，删除停用词，提取单词词干。删除停用词的目的是减少无意义词语消耗空间和时间资源，提取单词词干是为了避免受到大小写、时态、单复数等因素的影响。

b) 为了区分单词的位置，在经过处理的单词后面附加一个表示位置信息的字符，0 表示 From，1 表示 To，2 表示 Subject，3 表示 Content。

c) 将处理后的单词存入索引，修改相应的文档频率及词频信息。

(4) 为了减少查询时的计算量，提高速度，最后需要对 TF 文件的索引进一步处理：对词频进行对数处理，再进行归一化。

2.1.2 查询

计算 tf-idf 有很多种可选的方法，对于查询和文档常常采用不同的权重计算机制。这里我选用了上课时讲的 lnc.ltn 计算方法，即：文档: 对数 tf，无 idf 因子，余弦长度归一化；查询: 对数 tf，idf，无归一化。具体步骤如下：

(1) 读入需要查询的单词，提取单词词干，在单词后面附加位置信息，考虑到在同一项查询中可能会出现相同的单词，这里还需要对处理后的单词记录词频并去重。
(2) 从磁盘加载索引信息（DF.json 和 TF.json）。
(3) 构建查询向量$\vec{q}$，第 j 维数据q_j记录的是$d_q * t_q$，d_q表示文档频率经过对数处理的值，t_q表示查询单词的词频对数处理后的值。
(4) 构建文章向量：在索引中查找所需的单词，每个单词都对应着一个列表，列表中存有文章 ID，对于所有出现过的文章 ID，都需要构建一个向量来表示这篇文章，向量$\overrightarrow{a_i}$中的第 j 维数据a_{ij}记录的是查询中的第 j 个单词在文章 i 中的词频经过对数处理和归一化之后的值。
(5) 对于每一个文章向量，计算它与查询向量的乘积，结果即为相似度得分。用字典结构存储找到的文章 ID 以及相似度得分。
(6) 按照得分排序并返回文章 ID
(7) 在 ID.json 文件中，按照文章 ID 查找文档路径，返回路径的字符串集合。
(8) 按照用户的要求，在界面中展示相似度排名前 N 的邮件路径。

2.1.3 思路调整（分块索引）

在动手实践的过程中，发现了上面思路的一些不足之处：遍历完 517401 封邮件并写完索引，用了二十多个小时，记录词频的索引文件大小超过了 900M，字典越到后面体积越大，插入所需的时间也更多。无论是建索引还是读索引，都是很大的负担，因此在这里稍作完善，分块记录索引。

邮件文件夹是以 a-z 开头，并且每个文件夹下的邮件数目分布较均匀，因此，将 makedir 下的文件夹，按照文件夹的名称分类，以 a/b/c/d 开头的文件夹放入 a_d 文件夹，以 e/f/g/h/j 开头的文件夹放入 e_j 文件夹，以此类推，还有 k_p、

q_s、t_z 共五个文件夹，每个文件夹下的邮件约有 10 万封，对于这五个文件夹，按照上面的思路分别建立索引，最后再进行合并。需要说明的是，对于每个文件夹，文章 ID 有一个基础值，这个基础值是之前已编号的邮件数量，ID 从基础值开始向上自增，而不是每次都从 0 开始。

在合并的过程中，由于 ID 和 DF 的信息量并不大，可以分别存入一个文件；文章 ID 和对应词频的信息较大，应该分开存储，查询时有选择地读取，另外，考虑到局部性原理，每一块的体积也不应过小，适度为宜。比较单词的 ASCII 码，小于“d”的存入 TF_beforeD.json，大于等于“d”而小于“j”的存入 TF_EtoJ.json，以此类推，还有 TF_KtoP.json，TF_QtoS.json，TF_TtoZ.json，共五个文件。

在查询时，根据单词选择需要读入哪几个文件，减少读取与查找的时间。

2.2 文件说明

附件中，build_index.py 的作用是分块构建索引，normalize.py 的作用是对词频进行对数和归一化处理，merge_file.py 是将分块的索引进行合并，inquiry.py 的作用是读取用户的查询要求，构建向量计算余弦相似度，并返回查询结果。

temp 文件夹下，是分块构建索引过程中的中间结果，以 a_d 文件夹产生的中间结果为例：id_a_d.json 存储了文章 ID 与路径的信息；df_a_d.json 存储了单词与文档频率的信息；tf_a_d.json 存储了单词与文章 ID 及对应词频的信息。下图图 1 是 id_a_d.json 的截图，图 2 是 df_a_d.json 的截图，图 3 是 tf_a_d.json 的截图。

图 1 id_a_d.json 截图

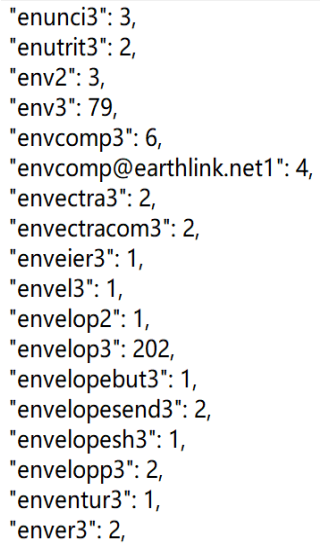

图 2 df_a_d.json 截图

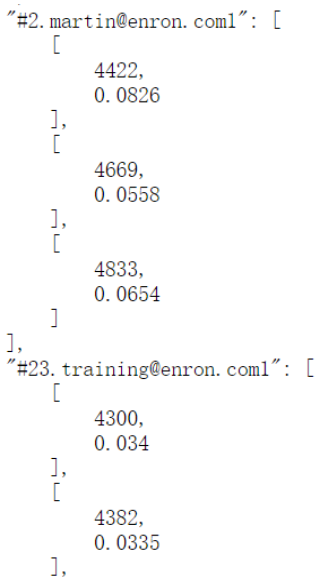

图 3 tf_a_d.json 截图

ID.json 存储所有邮件的 ID 及路径，DF.json 存储所有出现过的单词的文档频率，TF_beforeD.json 存储了 ASCII 码小于“d”的单词-文章 ID 索引，TF_EtoJ.json 存储了 ASCII 码在“d”到“j”之间的单词的单词-文章 ID 索引，TF_KtoP.json、TF_QtoS.json、TF_TtoZ.json 可以此类推。图 4 是 ID.json 的截图，图 5 是 DF.json 的截图，图 6 是 TF_beforeD.json 的截图。

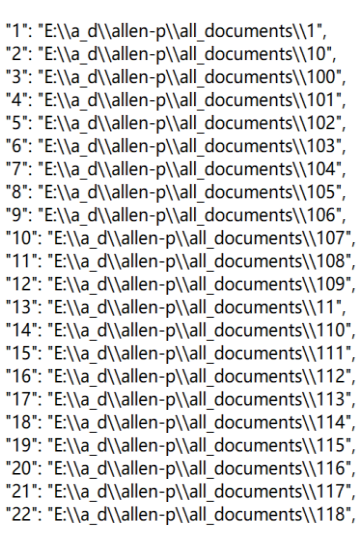

图 4 ID.json 的截图

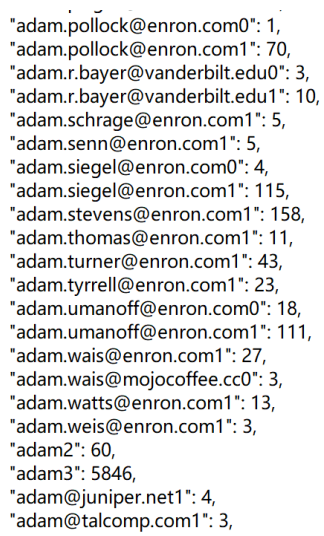

图 5 DF.json 截图

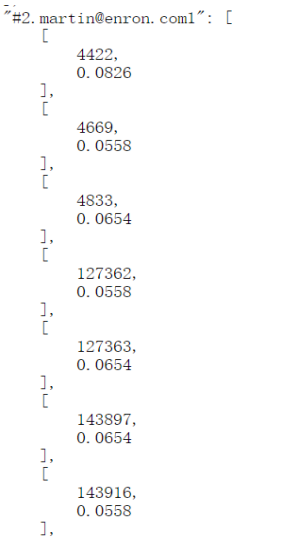

图 6 TF_beforeD.json

2.3 代码解释

对于代码的具体解释，在相应语句处已有标明，这里利用 Understand 代码分析工具，做出流程图，对邮件检索系统的逻辑做细致说明。

(1) build_index.py

build_index.py 的作用是分块构建索引，里面定义了两个重要的类：Email 和 Index。

每一个 Email 对象，记录了一封邮件的文章 ID，并调用 get_words()函数和 proc_text()函数，对其中的单词进行处理，处理后的单词记录在列表中，最后将数据传递给 Index 对象。图 7 是 get_words()的流程图，调用 email 包，提取不同位置的单词，按照四种位置（From、To、Subject、Content），在四个判断条件中，分别对单词进行分词、在尾部附加位置信息、统一大小写。第三和第四种情况是对主题和正文的处理，稍微复杂，需要调用 proc_text()函数去除停用词、提取单词的词干，图 8 是 proc_text()的流程图。

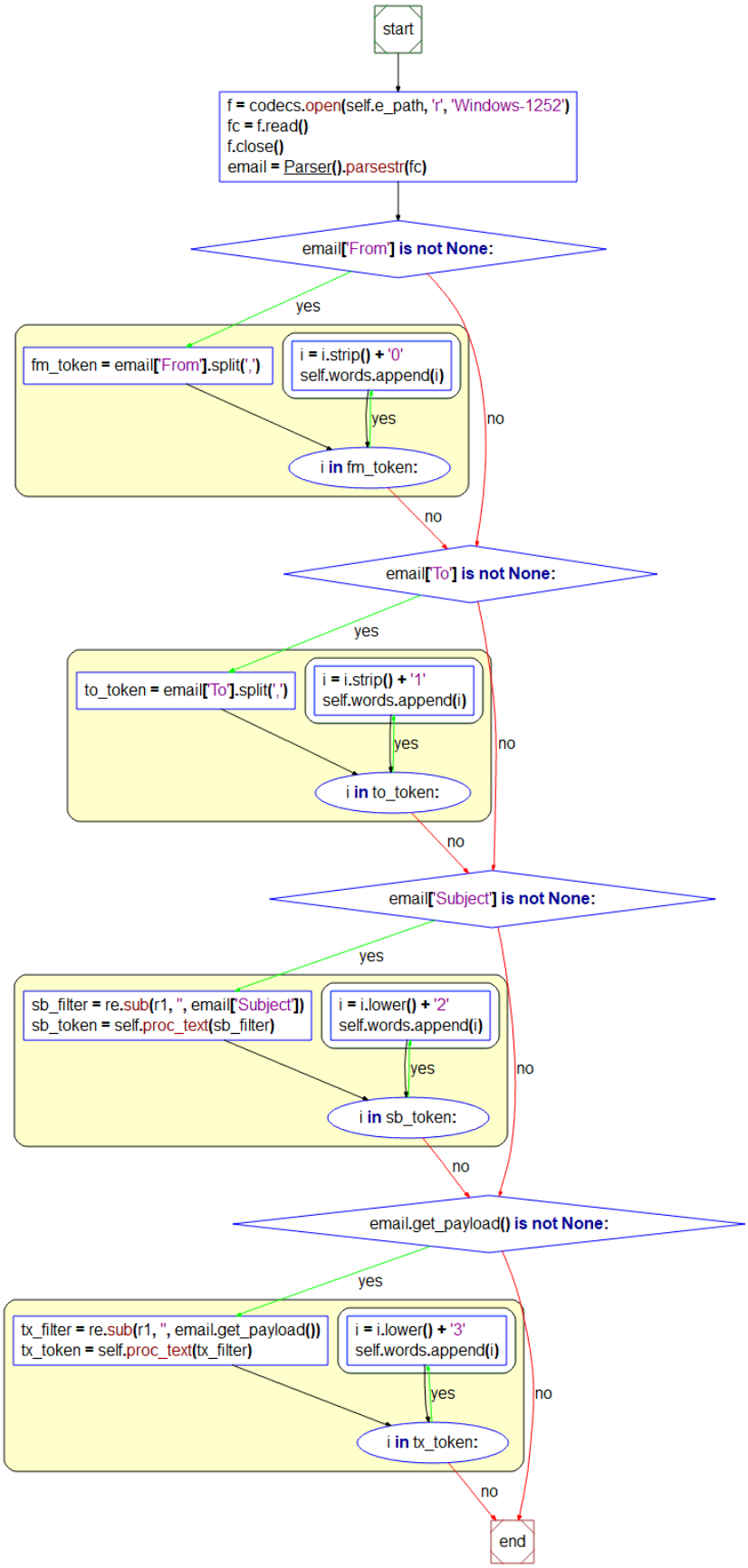

图 7 ClusterControlFlow-get_words

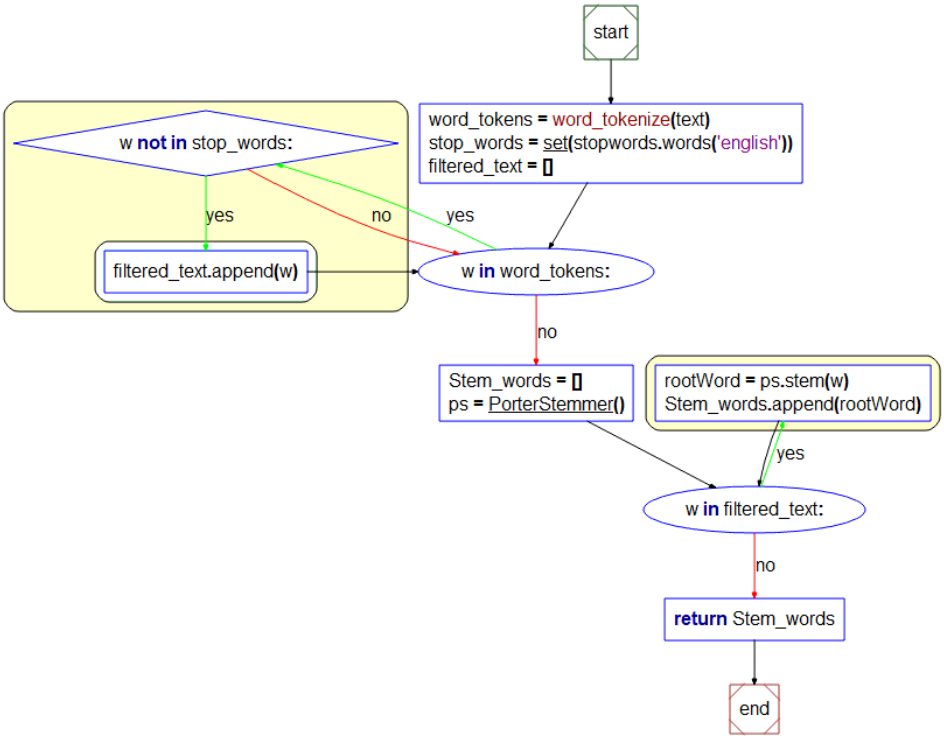

图 8 ClusterControlFlow-proc_text

关键代码：

```
if email['From'] is not None:
    fm_token = email['From'].split(',')  # 以','分割当前文档的 from 邮箱号
    for i in fm_token:
        i = i.strip() + '0'
        self.words.append(i)
# .strip()去空格（包括'\n', '\r', '\t', ' ')
if email['To'] is not None:
    to_token = email['To'].split(',')
    for i in to_token:
        i = i.strip() + '1'
        self.words.append(i)
# 邮件主题和内容中存在特殊符号、大小写、不同时态、单复数等，
# 用正则表达式过滤特殊符号，掉包删除停用词，以及对单词进行标准化
if email['Subject'] is not None:
    sb_filter = re.sub(r1, '', email['Subject'])
    sb_token = self.proc_text(sb_filter)
    for i in sb_token:
        i = i.lower() + '2'
        self.words.append(i)
# 文章内容
if email.get_payload() is not None:
```

```
22.     tx_filter = re.sub(r1, '', email.get_payload())
23.     tx_token = self.proc_text(tx_filter)
24.     for i in tx_token:
25.         i = i.lower() + '3'
26.         self.words.append(i)
```

Index 对象用于构造索引，根据每个 Email 对象传来的数据，调用 addIndex()函数，对字典进行修改或插入操作。addIndex()的流程图如图 9 所示：首先，遍历当前此封邮件的单词，统计词频，用字典记录，key 为单词，value 为词频，单词去重；其次，统计文档频率；最后，向索引中插入统计出来的结果，若单词为出现在词典中，则将 value 初始化为一个列表，若单词已经出现过，直接在列表中附加（ID，词频）元组。

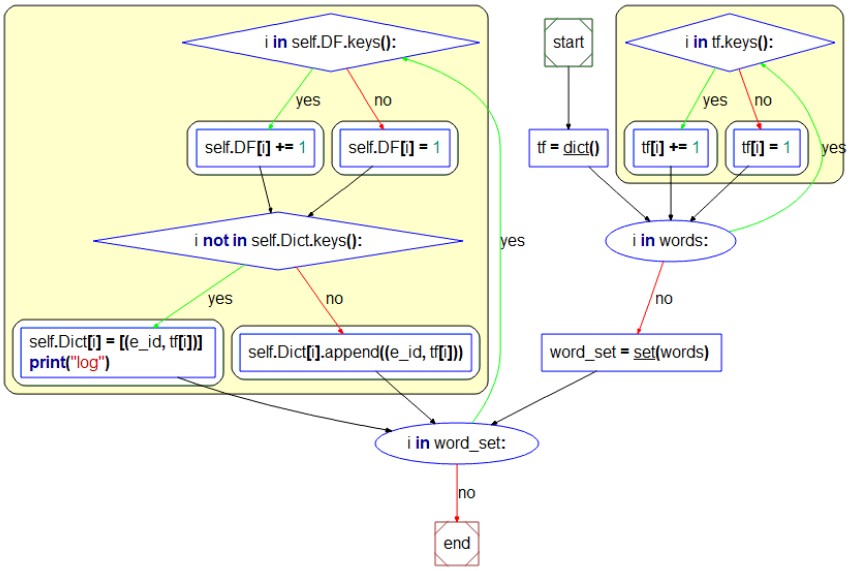

图 9 ClusterControlFlow-addIndex

关键代码：

```
1. for i in word_set:
2.     # 记录文档频率
3.     if i in self.DF.keys():
4.         self.DF[i] += 1
5.     else:
6.         self.DF[i] = 1
7.     # 向索引中加入这封邮件的信息
8.     if i not in self.Dict.keys():
9.         self.Dict[i] = [(e_id, tf[i])]  # 在这里加[]，将 value 变成一个 list，方便后面的 append
10.        print("log")
11.    else:  # 这个单词已经在词典中（使用 set 去重，该邮件 id 在之前一定未出现过，所以这一点不用考虑）
12.        self.Dict[i].append((e_id, tf[i]))
```

综合的 build_index.py 流程如图 10 所示，串行处理邮件的五个文件夹，每处理完一个文件夹，得到后一个文件夹的 ID 基础值（$base_i$），调用上面介绍的函数，将最终的字典结构以 json 格式写入磁盘。

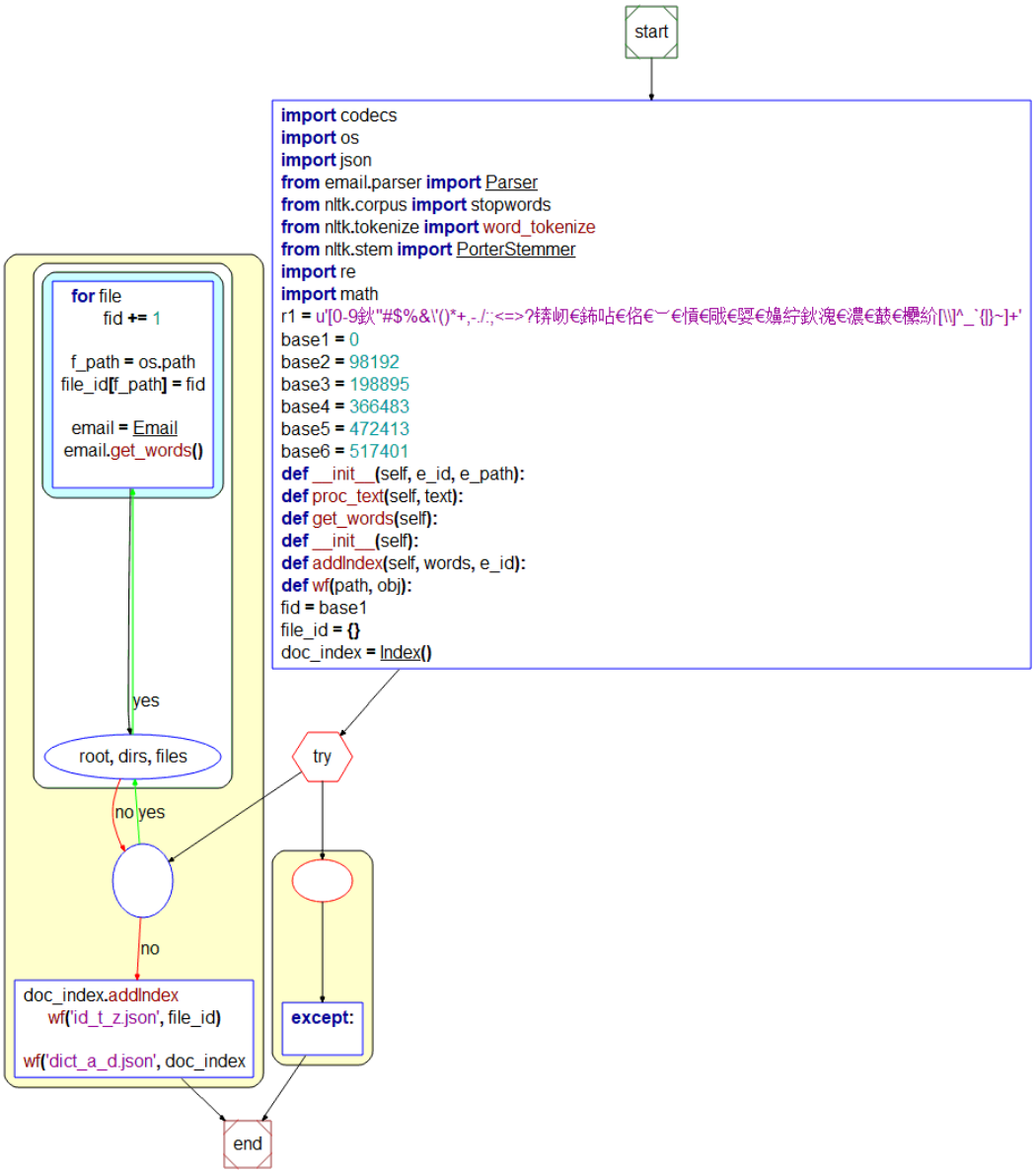

图 10 ClusterControlFlow-build_index

(2) normalize.py

normalize.py 的作用是对词频进行对数和归一化处理。对于每一个分块文件，首先，读出索引，调用 math 包中的 log()函数进行对数处理；之后，遍历整个索引，累计求出每篇文章的向量长度；最后，将每个词频除以对应的向量长度，完成归一化，写回磁盘。

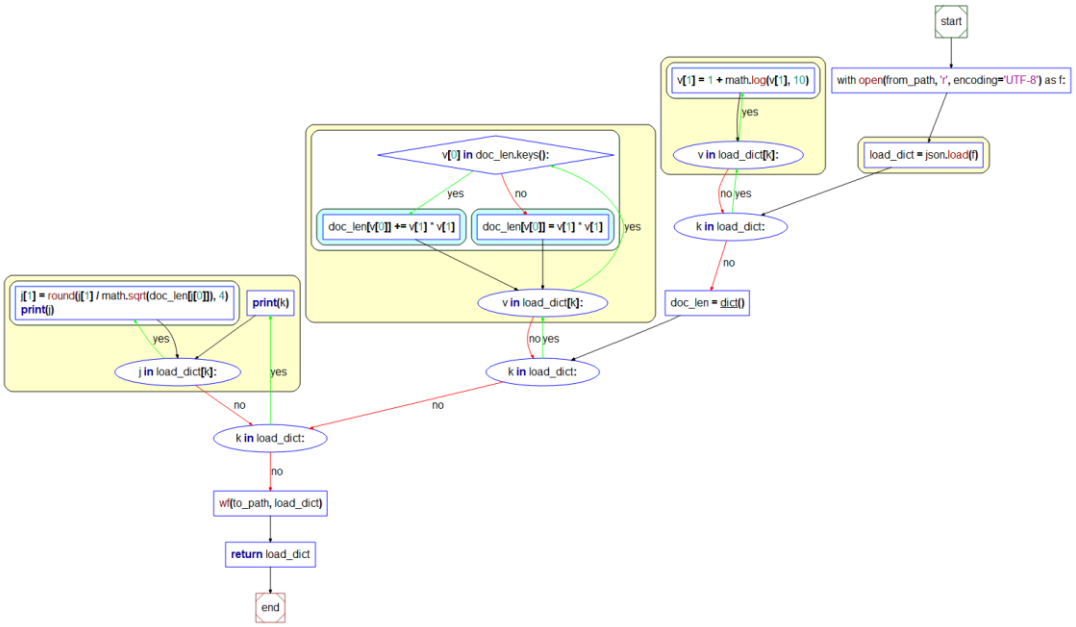

图 11 ClusterControlFlow-normalize

(3) merge_file.py

merge_file.py 是将分块的索引进行合并。图 12 和图 13 分别是合并 DF 文件、合并 ID 文件的流程图。合并 DF，就是在词典中加入未出现的单词及文档频率，将重复出现的单词的文档频率相加。合并 ID，就是将 5 个存有文章 ID 及路径的文件写入一个文件中，并且将 ID 作为 key，路径作为 value，便于查询阶段的搜索。

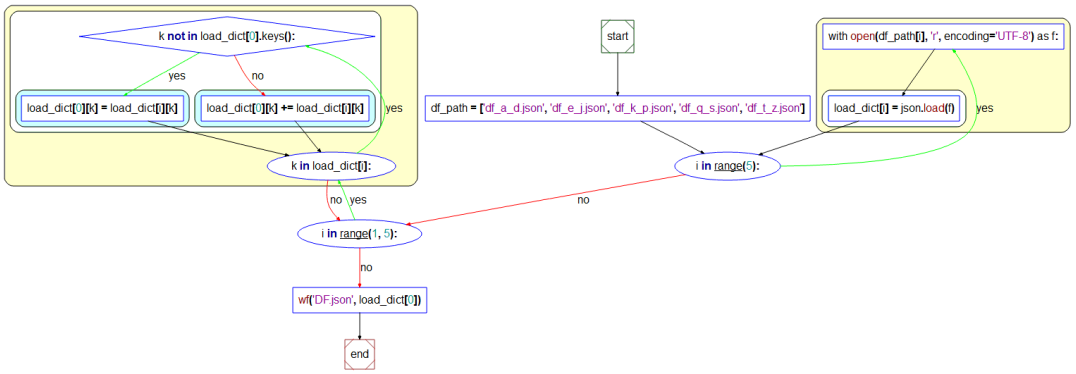

图 12 ClusterControlFlow-mergeDF

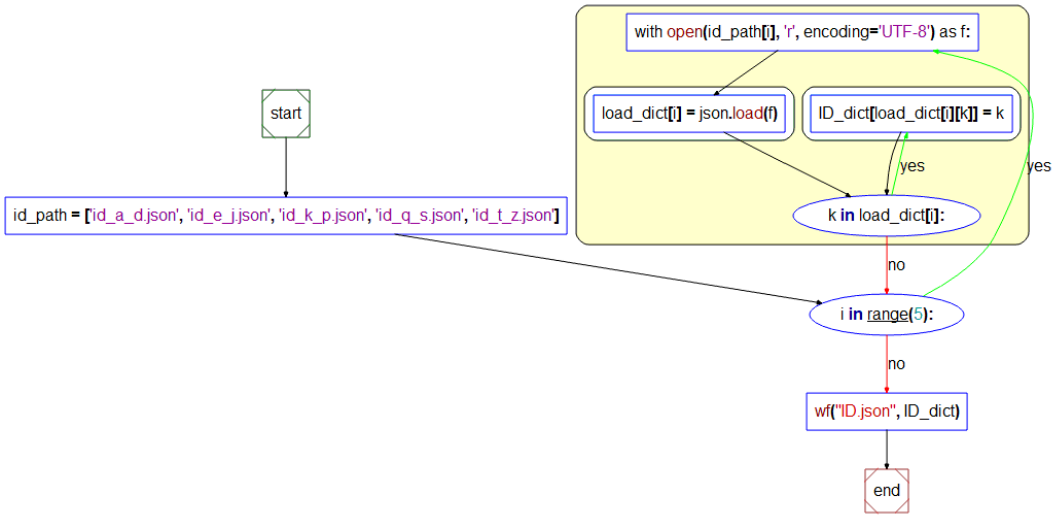

图 13 ClusterControlFlow-mergeID

图 14 是 merge_TF 函数的流程图。合并 TF 词典的过程比上面两个合并稍复杂一些，记录词频的文件体积较大，因此考虑将词频的信息按照单词首字母分别存入多个文件中。因为五个文件夹中的邮件不重复，因此词频并不需要加的操作，直接加入新的单词，或在已有单词的列表里附加元组，由于是按顺序读取的五个分块索引，所以附加的元组也是有序的，按照首字母存入不同的字典，最后写入磁盘。

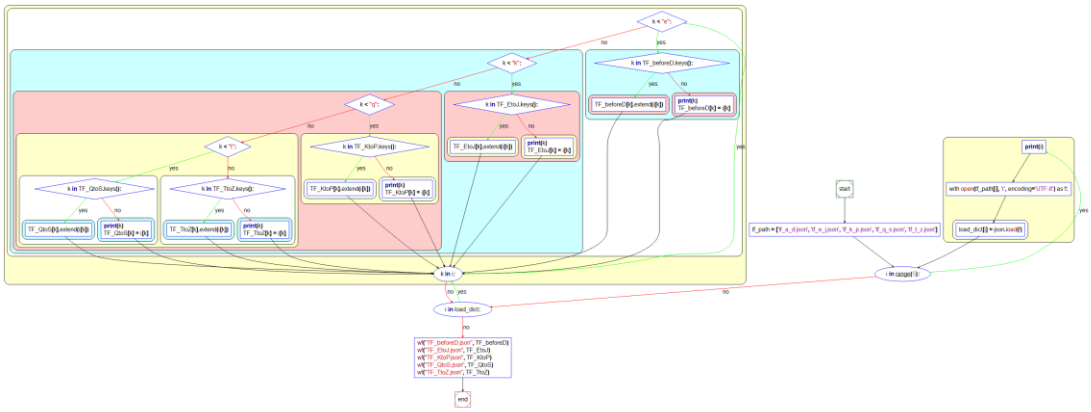

图 14 ClusterControlFlow-mergeTF

关键代码：

```
for i in load_dict:
    for k in i:
        # k 是字典 i 中的关键字，即单词
        if k < "e":
            if k in TF_beforeD.keys():
                # 追加文章 id 及 tf
                TF_beforeD[k].extend(i[k])
            else:
```

```
                # 新增单词
                print(k)
                TF_beforeD[k] = i[k]
        elif k < "k":
            if k in TF_EtoJ.keys():
                TF_EtoJ[k].extend(i[k])
            else:
                print(k)
                 TF_EtoJ[k] = i[k]
```

(4) inquiry.py

inquiry.py 的作用是读取用户的查询要求，构建向量计算余弦相似度，并返回查询结果。首先简单地说明 GUI 界面的设计：导入 tkinter 包，创建窗口，设计了五个输入框，分别是 From、To、Subject、Content、Number，用于获取用户想搜索的发件人、收件人、主题、正文，以及展示的路径数目，如图 15 所示。中间的按钮与 hit_btn()函数绑定，点击按钮后，系统会读取用户输入，将单词及位置列表传递给 search_id()函数。

图 15 查询界面

图 16 是 search_id()的流程图。计算 tf-idf 有很多种可选的方法，这里选用了上课时讲的 lnc.ltn 计算方法，即：文档: 对数 tf，无 idf 因子，余弦长度归一化；查询: 对数 tf，idf，无归一化。首先，读入文档频率的索引，对输入的单词提取词干，附加位置信息，统计查询语句中的单词词频，进行单词去重；之后，根据查询单词，读入需要用到的单词-文章 ID 索引，放入一个词典中；接着，计算出现的文章 ID 的向量，在建索引阶段已进行过预处理，这里只需要将处理过的词频填入相应的位置即可；然后，计算查询向量，这里需要用到上面统计的词频信

息以及文档总数；最后，计算每个文章向量与查询向量的余弦相似度，由于前面已经进行过归一化的处理，并且上面处理的过程中，文章与查询向量的每一维对应的是相同的单词，这里只需要做向量乘法即可。按照得分由高到低，返回所有相关的文章 ID。调用 find_path()函数，最终得到对应的文章路径，并按照用户的需求，将排名前 N 的邮件路径，显示在界面下方的文本框中（默认 N=0，表示展示全部）。

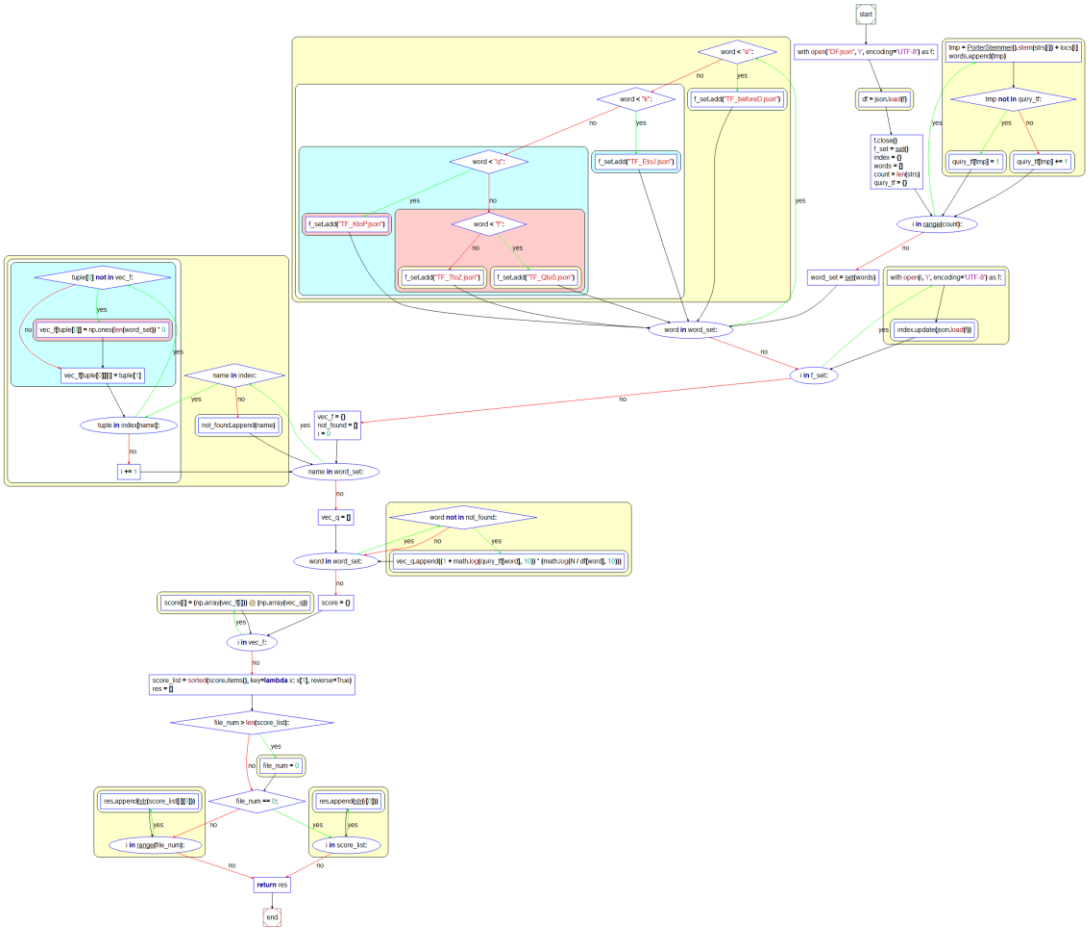

图 16 ClusterControlFlow-search_id

关键代码：

```
# 构建文章向量，放在一个词典里，key 是文章 id，value 是向量
    vec_f = {}
    not_found = []
    i = 0
    for name in word_set:
        if name in index:
            for tuple in index[name]:
                if tuple[0] not in vec_f:
                    # 如果文章 id（tuple[0]）已存在，那么直接改当前单词 tf 的值
                    # 对于新出现的每一篇文章
                    # 需要给他先初始化一个长度为集合大小的全 0 数组
                    vec_f[tuple[0]] = np.ones(len(word_set)) * 0
```

```
                    # vec_f[tuple[0]][i]是在 id 为 tuple[0]的文章向量中，单词 name 的位置
                    vec_f[tuple[0]][i] = tuple[1]
                i += 1
            else:
                # 若这个单词不在邮件中，那么所有文章的评分都不会受到它的影响，可以去掉这一个单词
                not_found.append(name)

    # 计算查询向量
    vec_q = []
    for word in word_set:
        if word not in not_found:
            # 出现在 tf 字典里的单词，一定也会出现在 df 字典里
            vec_q.append((1 + math.log(quiry_tf[word], 10)) * (math.log(N / df[word], 10)))

    # 算分，key 为文章 id，value 是得分
    score = {}
    for i in vec_f:
        score[i] = (np.array(vec_f[i])) @ (np.array(vec_q))
```

3. 功能演示

下面对邮件检索系统的功能进行演示：

(1) 查询内容中包含 Enron Global Technology 关键字的邮件，显示得分最高的前五封邮件地址，如图 17 所示。

图 17 查询（1）界面

抽取返回结果中的 E:\k_p\mcconnell-m\all_documents\335 和 E:\a_d\davis-d\inbox\103 做检验，结果如图 18 所示：

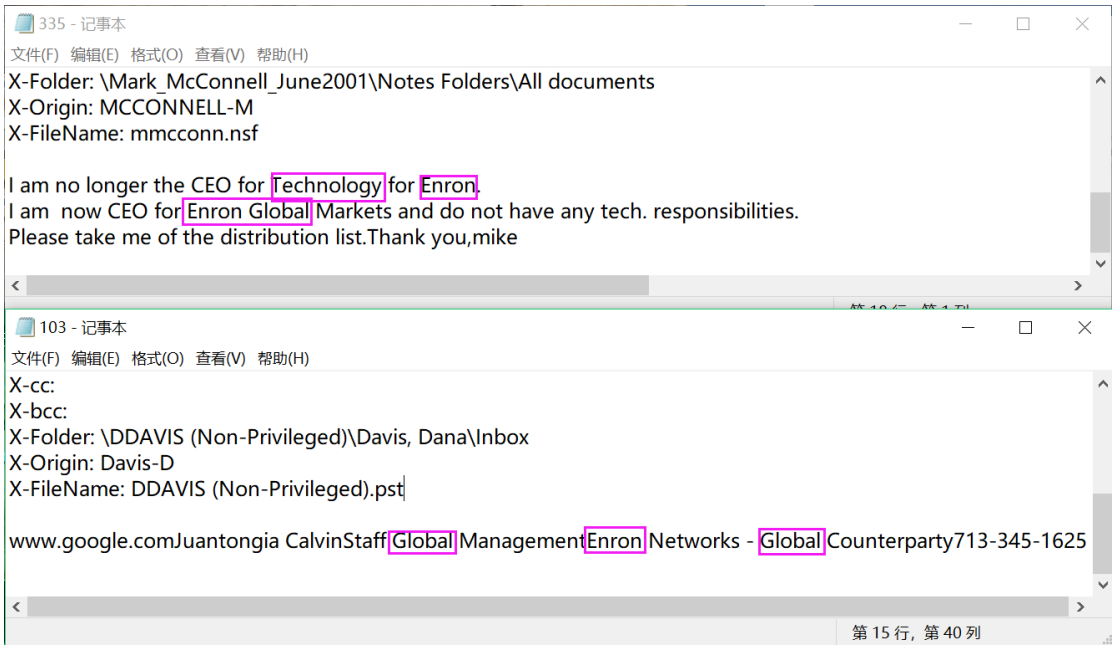

图 18 查询（1）验证

这里我们使用的是词袋模型，忽略了单词的顺序。另外，在其他邮件中也可以找到与 Enron Global Technology 相关的内容，但它们的正文长度较长，关键词不够突出，导致得分较低，未能出现在前五。

(2) 查询发件人是 rika.imai@enron.com，主题是 Fundamentals Meeting 的邮件，显示全部搜索结果，如图 19 所示。

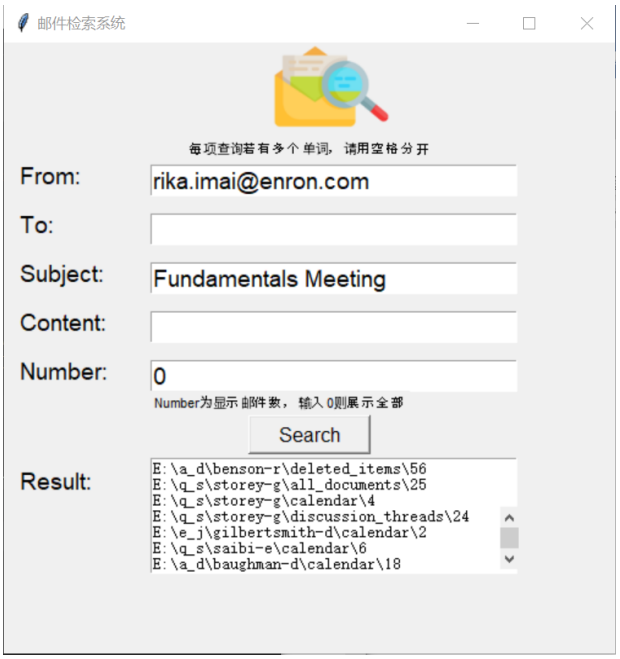

图 19 查询（2）界面

抽取返回结果中的 E:\a_d\benson-r\deleted_items\56 和 E:\a_d\benson-r\calendar\3 做检验，结果如图 20 所示：

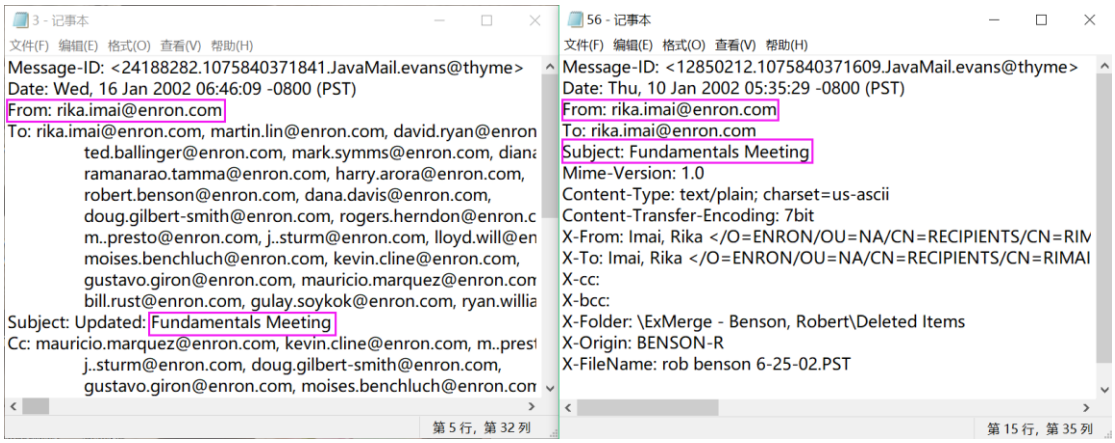

图 20 查询（2）验证

4. 实验收获

除了更深刻地理解了倒排索引和向量空间模型之外，还想记录一下自己踩的坑：

(1) 为了避免编码的问题，将路径全改成英文，在写文件路径时，要注意反斜杠的转义效果。

(2) 不同位置的单词用不同的处理方法。比如邮箱就只需要用逗号分隔，而不需要进行分词，否则会出现图 21 的结果，这样的分词结果是没有意义的。

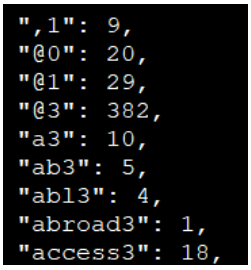

图 21 邮箱被过度拆分

(3) 关于是否去除停用词衡量了很久，通过测试小样本，发现去除停用词的过程十分耗时，加之老师上课总是提到“to be or not to be”特殊的例子，这样看来是不需要去停用词的；但是停用词占用的体积较大，也会降低查询的效率，按照网上的说法，我们可以在执行以下任务时删除停用词：

- 文本分类（垃圾邮件分类、语言分类、体裁分类）
- 标题生成
- 自动语言生成

而在机器翻译、语言建模、文本摘要中避免删除停用词。权衡之下还是选择去掉了正文中的停用词。

(4) 邮件内容需要过滤掉各种乱七八糟的东西，比如\n 和-；另外，看到建立的索引中有很多奇怪的字符串，在相关邮件中进行了验证，发现是原邮件中的单词串的影响，如图 22 所示。

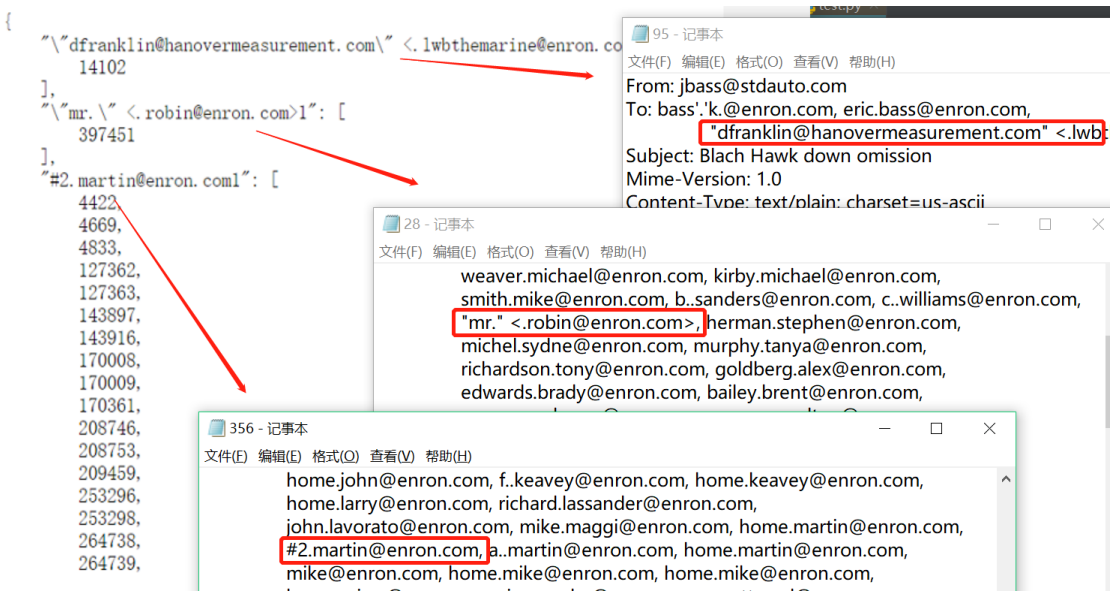

图 22 单词验证

(5) 在向列表添加元素时使用了 append()和 extend()两种方法，在这里记录下它们的区别：append() 方法向列表的尾部添加一个新的元素。只接受一个参数。 extend()方法只接受一个列表作为参数，并将该参数的每个元素都添加到原有的列表中。

(6) 对于邮件结构的认识还是不足，对于 X-cc、X-From 等内容不太了解；另外，没有处理附件信息，导致索引中会出现很长的字符串，但是数量极少，不会影响总的查询效果。如果时间允许的话，可以使用 is_multipart()这个函数，判断附件的存在，单独进行处理。