古诗词检索系统

计算机科学与技术专业　　1711436　　皮春莹

1. 问题描述

实现中华古诗词检索系统，对于给定的中华古诗词数据集（json 格式存储）创建复合索引（双词+位置），支持短语查询，支持与、或、非操作。在此基础上，进行 UI 展示以及对给定数据集的数据分析与古诗词翻译。

2. 实验环境

(1) Python 3.6
(2) JetBrains PyCharm Community Edition 2019.1.3

3. 实现思路

3.1 数据分析

相比于上次的邮件数据，这次诗词数据的数量和规范程度真让人感动。这次的数据集是 255000 首宋词和 58000 首唐诗，分批存储在 json 文件中，每个 json 文件中是一个列表，每个列表中有 1000 个元素，每个元素是一个字典，包含着一首诗词的 strains、author、paragraphs 和 title 信息。

文件名也很友好，都编好了号，所以在对诗词进行编号时，对于某个列表中的第 i 项（0<=i<999），它的 ID 对应为文件名中的数字加上 i。编号与诗词一一对应。在查询过程中，诗词的存储路径也可以通过诗词编号计算出来，不用再存储这个对应关系。

3.2 数据处理

对数据的预处理（在 rf()函数中）主要是使用 jieba 分词，统计每首诗词中的单词。具体过程：利用文件名和数组下标对诗词进行编号（记为 id），每首诗词的单词信息用列表存储（记为 single_poem），列表的每一项是单词和位置偏移构成的元组（word，pos）。对于当前文件中的每一首诗词，调用 jieba 分词包，使用搜索引擎模式对每一句进行分词，最开始因为考虑到用户输入的单词可能不是完整的短语，在精确模式的基础上，对长词再次切分，可以提高召回率，所以选用了“搜索引擎模式”而不是“精确模式”或“全模式”。后面在实际搜索时，发现“搜索引擎模式”对于单词偏移量的确定带来了一些问题，所以最后还是改

成了使用精确模式。

分词的结果过滤掉符号等无用信息，然后将单词和该词在诗词中的偏移量存入 single_poem 列表，最终将文件中的所有诗词信息放入一个列表中返回。另外，为了便于完成附加功能里的数据统计分析，在这里同时统计下出现过的单词和诗人。

3.3 构建索引

在这次作业中需要建两类索引——双词索引和带位置信息的索引。双词索引是将每两个连续的词组成词对(作为短语)来索引，索引构建时将每个词对看成一个词项放到词典中。这样的话，两个词组成的短语查询就能直接处理。位置索引是在倒排记录表中，对每个 term 在每篇文档中的每个位置(偏移或者单词序号)进行存储。

- 索引的结构

双词索引（记为 BiWordIndex）是一个字典结构，字典中的每一项中，key 为词项，value 为词项所在的诗词 ID 的列表。如图所示。位置索引（记为 PosIndex）也是一个字典结构，字典中的每一项，key 为词项，value 是一个列表，列表中的每个元素表示一首诗词关于该词项的位置信息，仍然是一个列表来存储，0 号元素存放诗词 ID，后续的元素是 key 在该诗词中的偏移量。

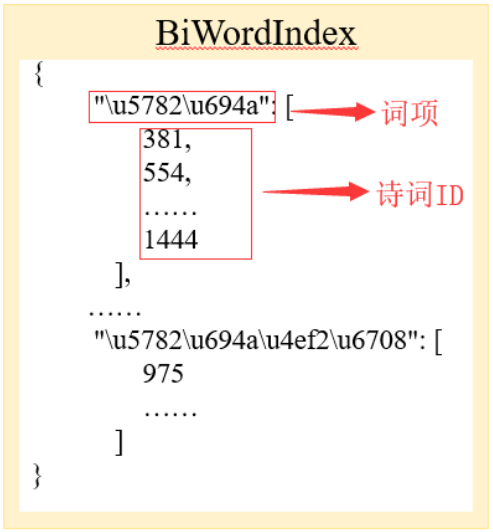

图 1 BiWordIndex

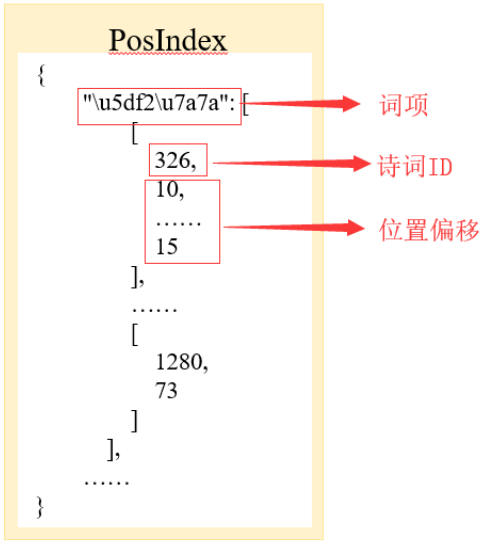

图 2 PosIndex

另外，统计诗人和单词的信息比较简单（分别记为 Poets 和 Bag），用两个列表分别存储就可以。

- 向索引中增加信息

对于每一个文件预处理得到的结果，依次处理单词集合的列表，添加双词索引时，将每两个连续的词作为一个词项，会有三种情况：

(1) 该词项在双词索引中尚未出现，那么，加入{词项，[PoemID]}的键值对；

(2) 该词项已在双词索引中出现，但词项对应的列表中没有当前的 PoemID，那么，在该列表中加入 PoemID；

(3) 该词项已在双词索引中出现，并且词项对应的列表中含有当前的 PoemID，

那么，直接处理下一个词项。

利用元组（单词 term，偏移 pos）向位置索引 PosIndex 添加信息时，会有以下三种情况：

(1) term 不属于 PosIndex 的关键字，那么，加入 key 为 term，value 为[[PoemID,pos]]的键值对；

(2) term 是已经存在的关键字，但它对应的列表中未出现 PoemID，那么，在他现有列表中附加一个列表[PoemID，pos]；

(3) term 是已经存在的关键字，并且对应的列表中的某个内层列表的 0 号元素是 PoemID，那么，在内层列表中追加元素 pos。

最后，将索引写入磁盘。由于位置索引的文件体积太大，所以在实际编程的过程中，我将唐诗宋词分成了三个文件夹，分块建了索引，每次只加载一部分信息，有需要时，才加载更多的索引。

3.4 数据查询

从用户图形化界面中获取到的是一个字符串，指明需要进行的短语查询、与或非操作，查找索引，返回相关诗词的编号，利用编号映射到诗词所在的文件，读出诗词内容。

- 提取查询信息

我设计的输入的字符串格式为“xxxx or/and/andnot xxxx or/and/andnot xxxx……”，其中，xxxx 是单词、短语或诗句，由 and、or 或 andnot 相连接，表示这些短语之间的与或非关系。在这里，考虑到优先级的关系，提取查询信息主要分为以下三个步骤：

(1) 先利用“or”拆分字符串，把拆分结果记录在 or_terms 列表中。

(2) 对于其中的每一项，继续用“and”拆分，若该词项中不含“and”，则不做操作；若该词项中含有“and”，则将拆分的结果记录在 and_terms 列表中，并且从 or_terms 中删去该词项。

(3) 对于 and_terms 中的每一项，用“not”进行拆分，若该词项中不含“not”，则不做操作；若该词项中含有“not”，则将拆分的结果记录在 not_terms 列表中，并且从 and_terms 中删去该词项。

至此，or_terms、and_terms、not_terms 中各自分别存着需要与其他两个列表进行或操作、与操作、非操作的词项；而在 or_terms 列表的内部，词项之间是或的关系，在 and_terms 和 not_terms 的内部，词项之间是与的关系。而对于每一个列表中的每一个词项，若词项是一个单词，那么不用做过多的处理，直接查索引，提取诗词编号的列表；如果词项不仅仅是一个单词，而是两三个单词组成的短语，或者是一句诗，那么在查找时需要保证单词之间的位置关系，首先要对该词项进行分词，然后将每两个相邻的单词组合成一个词项，在索引中找到该次相对应的诗词编号列表，多个列表之间进行“与”操作。

由于双词索引体积小，查询速度较快，但对查询词项的要求较高，有的词项在双词索引中可能搜索结果为空；而位置索引记录的信息较为全面，能提高召回率，但体积过大，搜索速度慢。因此在查索引时，利用双词索引对计算代价比较高的短语进行索引，若返回结果不够满意，再利用位置索引对短语进行处理，这样就充分发挥各自的优势。

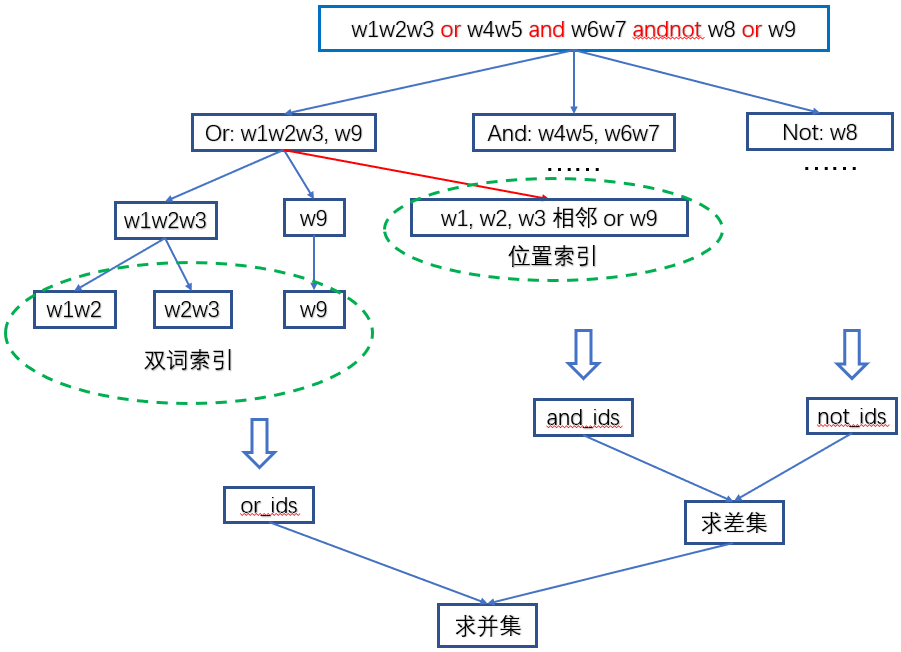

图 3 诗词查询

在用位置索引查询时，对于同一种运算关系中的由短语分词得到的的单词列表，每次取一个单词，用字典存储它涉及到的诗词 ID 及对应的偏移信息，以文章 id 为主键，值为列表，列表中存储关键词出现过的位置。以第一个单词为基准，每次将当前第 m 个单词的字典与它的字典进行对比，若在两个字典中，对于同一篇文章，得到的两个列表中出现位置相差为 m，证明当前 id 符合条件。

以“明月清風 or 平世功名”查询为例，先用“or”“and”“not”依次拆分，得到的结果：or_terms=[‘明月清風’,‘平世功名’]，and_terms=[]，not_terms=[]。对 or_terms 中的每一项进行分词，分别得到 ['明月', '清風']和['平世', '功名']。

先从磁盘加载双词索引，将分词结果的每两项一组合，两个列表分别得到了词项：“明月清風”、“平世功名”，这里不同于最初 or_terms 中元素的是，or_terms 中的元素可能是单词或两个及两个以上的单词组成的短语，但这里的处理结果一定是单词或两个单词组成的短语。

组合成词项之后，在双词索引中查找每个词项，记录他们对应的列表，多个列表分别排序去重后，进行或操作，得到对于 or_terms 的诗词编号列表 or_ids[]，由于 and_terms 和 not_terms 为空，所以对应的 and_ids[]和 not_ids[]也为空，将 or_ids[]和 and_ids[]进行或操作，再与 not_ids[]进行非操作，得到由双词索引检索的结果。判断首次检索的结果是否满意，这里我的判断标准是搜索到的诗词数量，如果数量较少的话，需要继续去位置索引中查找。

如果上面的例子需要在位置索引中进行查找，仍然需要使用逻辑词拆分、结巴分词之后的结果：['明月', '清風']和['平世', '功名']。先查找“明月”对应的列表，若关键词不存在，则返回空列表，再查找“清風”对应的列表，若关键词不存在，则返回空列表。当上面两个列表都不为空时，例如找到的分别是[[12,5,10],[20,8,20],[25,10]]，[[28,7],[12,9],[9,17,30]]，找出每个涉及到的诗词，即内层列表的首个元素，使用字典结构，以诗词 id 为主键，值为列表，列表中存储关键词出现过的位置，若在列表中出现连续长度等于外层列表数量时，证明当前 id 符合条件。在这里形成的字典是{9: [17,30], 12: [5,9,10], 20: [8,20], 25: [10], 18:

[7]}，检查可知，id=12 的诗词满足条件，对['平世','功名']也进行相同操作，将得到的结果进行或操作，返回诗词列表。

3.5 附加功能

在完成主要的功能之外，还编写了用户图形化界面。这个主要使用了 python 的窗口视窗设计的模块 tkinter，放置了用户输入框和结果展示的文本框。在展示查询结果时，由上面的介绍可知，搜索功能的函数返回的是诗词 ID 集合，需要在这里将 ID 翻译成对应的文件名，具体的对应关系在“数据分析”中已经说明。这里设计的每次只展示一首诗词的名称、作者和内容，文本框下面放置了“Next”按钮，点击按钮时，系统会根据下一个诗词 ID，加载诗词内容并显示在文本框中。

界面的左侧是关键字排名分析和作者产量分析。由于在数据处理的过程中，已经将分词产生的所有单词存入了 Bag，将诗人的名字存入了 Poets，这里只需要对数据进行汇总和展示。我计划将关键字和作者的信息做成多张词云图，然后将这多张图片合成 GIF 动图，在界面右侧展示出来。词云是对文本数据中出现频率较高的"关键词"在视觉上的突出呈现，在 python 中可以用 wordcloud 包比较轻松地实现：加载 Bag 和 Poets，分别将两个列表内的元素用空格连接，然后设置词云的背景色、字号、字体、配色等属性，这里需要注意的是，wordcloud 包不支持繁体中文，所以我另外下载了方正黑体繁体的字体包并进行了设置（font_path="FangZhengHeiTiFanTi-1.ttf"），之后将生成的词云图保存成 PNG 格式，并将多张 PNG 图片合成 GIF。

由于没有找到 python 的 tkinter 包中可以直接展示 GIF 的控件，我编写了 ImageLabel 类，用于展示 GIF 图。ImageLabel 类中定义了 load()、unload()和 next_frame()函数，load()使用了 PIL 的 Image 和 ImageTk 工具包，先调用 Image 的 load()函数加载 GIF，每隔固定的时间段，循环展示一帧，循环展示的过程由 next_frame()控制。界面如下图所示：

图 4 初始界面

4. 代码实现

本实验分为三个文件：splitSentence.py、draw_graph.py 和 search_words.py。其中，splitSentence.py 是对数据进行处理，并建立双词索引和位置索引。draw_graph.py 是将关键词和作者信息做成词云图。search_words.py 是显示查询界面，根据用户的输入，加载索引并进行查询，返回搜索结果。

4.1 splitSentence.py

读取诗词文件，对于每首诗词，使用 jieba 的精确模式进行分词，并给每个词附加位置偏移信息，将结果存入列表。

```
1.  def rf(path, obj):
2.      # 返回的应该是一个长度为 1000 的列表，每一项是当前文档内的一首古诗的内容分词结
    果
3.      tmp_file = open(path, 'r', encoding='utf-8')
4.      tmp_list = json.loads(tmp_file.read())
5.      for j in range(1000):
6.          # 对于每首诗词，先提取正文部分
7.          sentence_list = tmp_list[j]["paragraphs"]
8.          poet = tmp_list[j]["author"]
9.          poem_index.poets.append(poet)  # 便于完成附加功能里的数据统计分析
10.         word_count = 0
11.         # 每首诗词的单词信息用列表存储，列表的每一项是单词和位置偏移构成的元组
12.         single_poem = []
13.         for sentence in sentence_list:
14.             # 使用 jieba 的精确模式进行分词，把结果放入 obj
15.             words = jieba.lcut(sentence)
16.             for word in words:
17.                 # 过滤掉符号等无用信息
18.                 if word not in [', ', '。', '! ', '? ', ':
    ', '{', '}', '/', '（', '）']:
19.                     # 给每个词附加位置偏移信息
20.                     single_poem.append((word, word_count))
21.                     word_count += 1
22.         obj.append(single_poem)
23.     tmp_file.close()
```

定义了 Index 类，便于整理所有的数据处理结果、双词索引、位置索引，Index 类中定义了 add_index(self, poem_id, word_tuples)函数，参数 poem_id 是根据文件名和诗词的下标计算得到的，计算 poem_id 的代码比较简单，不再粘贴。word_tuples 是经过上面的 rf()函数处理过的文件中的一首诗词（每个文件中有一千首），add_index()的流程图如下：

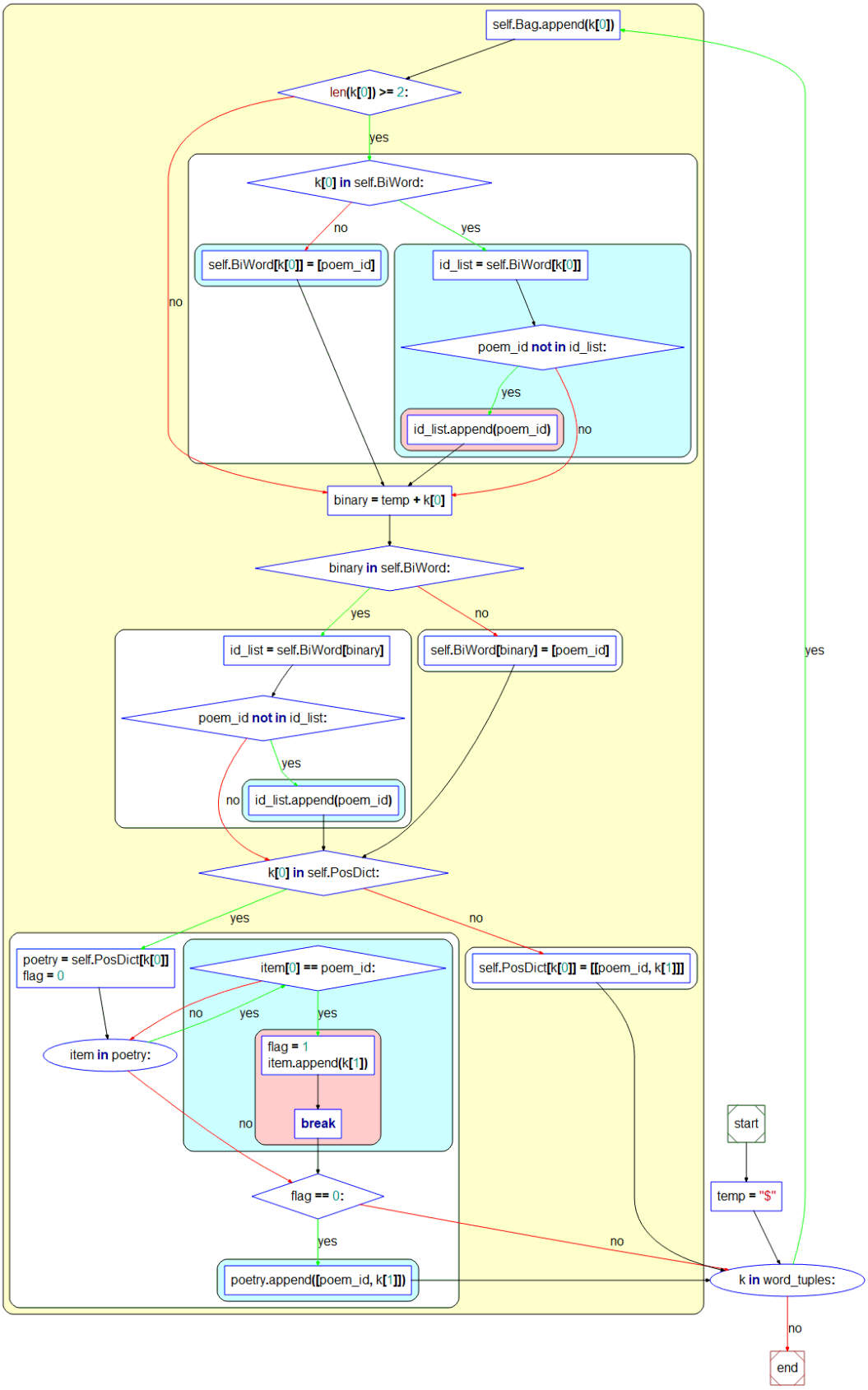

图 5 ClusterControlFlow-add_index

下面是 Index 类的定义：

```
class Index():
    def __init__(self):
        self.PosDict = dict()  # 位置索引
        self.BiWord = dict()  # 双词索引
        # 统计诗人和单词的信息
        self.Bag = list()
        self.poets = list()

    # 将每篇文章得到的单词集合插入索引
    def add_index(self, poem_id, word_tuples):
        temp = "$"
        # words 是一个元素为元组的列表
        for k in word_tuples:
            # k=(word,pos)
            self.Bag.append(k[0])
            if len(k[0]) >= 2:
                if k[0] in self.BiWord:
                    id_list = self.BiWord[k[0]]
                    if poem_id not in id_list:
                        id_list.append(poem_id)
                else:
                    self.BiWord[k[0]] = [poem_id]
            # 添加双词索引，
            # 将每两个连续的词组成一个词项放到词典中
            binary = temp + k[0]
            # 会有三种情况：
            # (1)词项在双词索引中尚未出现，加入{词项，[PoemID]}的键值对；
            # (2)词项已在双词索引中出现，但词项对应的列表中没有当前的 PoemID，在该列表中加入 PoemID；
            # (3)词项已在双词索引中出现，并且词项对应的列表中含有当前的 PoemID，直接处理下一个词项。
            if binary in self.BiWord:
                id_list = self.BiWord[binary]
                if poem_id not in id_list:
                    id_list.append(poem_id)
            else:
                self.BiWord[binary] = [poem_id]

            # 添加位置索引
            # key 为词项，value 是一个双层的列表，内层列表表示一首诗词关于该词项的位置信息，
            # 0 号元素存放诗词 ID，后续的元素是 key 在该诗词中的偏移量
            if k[0] in self.PosDict:
```

```
                # term 是已经存在的关键字
                poetry = self.PosDict[k[0]]
                # 在 poetry 列表中找编号 id 的列表，并加入位置信息
                flag = 0  # 0 表示 poem_id 不在当前的列表中
                for item in poetry:
                    if item[0] == poem_id:
                # 对应的列表中的某个内层列表的 0 号元素是 PoemID，那么，在内层列表中追加元素 pos
                        flag = 1
                        item.append(k[1])
                        break
                if flag == 0:
                    # 对应的列表中未出现 PoemID，那么，在现有列表中附加一个列表 [PoemID，pos]
                    poetry.append([poem_id, k[1]])
            else:
                # term 不属于 PosIndex 的关键字，那么，加入 key 为 term，value 为 [[PoemID,pos]]的键值对；
                self.PosDict[k[0]] = [[poem_id, k[1]]]
```

剩余的代码是递归读取文件夹下的每个文件，以及将索引写入磁盘，比较简单，不再粘贴在这里。

4.2 draw_graph.py

加载 Bag 和 Poets，用 wordcloud 包制作词云图：

```
from wordcloud import WordCloud
import matplotlib.pyplot as plt
import json


def draw_graph():
    with open("song_words_bag.json") as f:
        mytext1 = json.load(f)
    f.close()
    wl = " ".join(mytext1)
    # 加载背景图片
    # cloud_mask = np.array(Image.open("cloud.png"))
    # 设置词云
    wc = WordCloud(background_color="white",  # 设置背景颜色
                   # mask=cloud_mask,  # 设置背景图片
                   max_words=2000,  # 设置最大显示的字数
                   # stopwords = "", #设置停用词
```

```
                    font_path="FangZhengHeiTiFanTi-1.ttf",
                    # 设置中文字体，使得词云可以显示（词云默认字体是“DroidSansMono.ttf 字体库”，不支持中文）
                    max_font_size=50,  # 设置字体最大值
                    random_state=30,  # 设置有多少种随机生成状态，即有多少种配色方案
                    )
    myword = wc.generate(wl)  # 生成词云

    # 展示词云图
    plt.imshow(myword)
    plt.axis("off")
    plt.show()
    # 存储图片
    wc.to_file('a4.png')

if __name__ == '__main__':
    draw_graph()
```

4.3 search_words.py

界面上的控件比较简单，这里不再解释相关的代码。按键“Search”绑定了hit_btn()函数，用于读取用户输入的字符串，之后调用 search_poem()函数，得到相关诗词的编号，并将搜索结果中的第一首诗词信息展示在文本框中。

```
def hit_btn():
    # 定义一个函数功能供点击 Button 按键时调用，调用命令参数 command=函数名
    # 搜索结果总数
    global total
    # 搜索结果
    global ans_ids
    # 当前展示的诗词的计数
    global tmp_num

    # 获取用户输入的信息
    e_from = var_from.get()
    print(e_from)
    ans_ids = search_poem(e_from, "song")
    total = len(ans_ids)
    tmp_num = 0
    if total > 0:
        t.delete(0.0, END)
        # 对应到文件名
        file_id = int(ans_ids[0] / 1000) * 1000
```

```
        print("The first fileID and poemID-in-file to be show is")
        print(file_id)
        num = ans_ids[0] - file_id
        print(num)
        # 由于唐诗和宋词是分块建的索引，所以要先判断需要加载哪个文件
        if ans_ids[0] < 255000:
            # 宋词
            path = "E:\ir\chinesepoetry\poet.song." + str(file_id) + ".json"

        else:
            # 唐诗
            path = "E:\ir\chinesepoetry\poet.tang." + str(file_id) + ".json"


        with open(path, 'r', encoding='UTF-8') as f:
            tp = json.load(f)
        f.close()
        # 将搜索结果中的第一首诗词信息展示在文本框中
        poem_dict = tp[num]
        author = poem_dict["author"]
        para = poem_dict["paragraphs"]
        title = poem_dict["title"]
        t.insert(INSERT, title + "\n")
        t.insert(INSERT, author + "\n")
        for i in para:
            t.insert(INSERT, i + "\n")
    else:
        t.delete(0.0, END)
        t.insert(INSERT, "未搜索到相关内容\n")
```

“Next”按钮绑定了 show_btn()函数，通过改变全局变量 tmp_num 的值，来切换文本框的内容，代码比较简单，不再粘贴。

在 hit_btn()中调用了 search_poem()函数，search_poem()的作用是根据用户输入，返回相应的诗词 ID 集合。在 search_poem()中，根据逻辑运算的优先级，将查询语句拆分成三个列表（or_terms、and_terms 和 not_terms），具体的拆分方法在实验思路中已经阐述。然后加载双词索引。接下来分别对每个列表进行操作，列表内部的每个词项，是一个或多个相邻的短语，处理的方法类似，写在 internal_term()中，在这里直接对每个词项调用该函数。对于列表中每个词项得到的 PoemID 列表，在 or_terms 中需要进行或操作，因为只要有一个满足就可以选择该 ID，在 not_terms 中也需要进行或操作，因为只要有一个不满足就可以过滤该 ID，但在 and_terms 中需要进行与操作，因为需要满足全部词项的 ID 才可以被留下，最后调用 merge_vector()将 or_terms、and_terms 和 not_terms 得到的诗词编号列表进行合并。下面是 search_poem()的前半部分，是关于双词索引的部分：

```
or_terms = []
```

```
and_terms = []
not_terms = []
# 考虑优先级的关系，提取查询信息
# 先利用“or”拆分字符串，把拆分结果记录在 or_terms 列表中
or_terms.extend(quiry.split('or'))
for each in or_terms:
    print(each)
    and_terms.extend(each.split('and'))
    # 对于其中的每一项，继续用“and”拆分，若该词项中不含“and”，则不做操作；
    if 'and' in each:
        # 若该词项中含有“and”，则将拆分的结果记录在 and_terms 列表中，并且从 or_terms 中删去该词项
        or_terms.remove(each)
for each in and_terms:
    # 对于 and_terms 中的每一项，用“not”进行拆分，若该词项中不含“not”，则不做操作
    if 'not' in each:
        # 若该词项中含有“not”，则将拆分的结果记录在 not_terms 列表中，并且从 and_terms 中删去该词项
        not_terms.append(each.replace('not', ''))
        and_terms.remove(each)

print(or_terms)
print(and_terms)
print(not_terms)

filepath = prefix + "_poem_BiWordIndex.json"
# 先进行双词索引
with open(filepath, 'r', encoding='UTF-8') as f:
    BiWdIndex = json.load(f)
f.close()
# 要保证插入的 id 是有序且不重复的，集合（set）是一个无序的不重复元素序列
not_ids = []
ni = 0
nm = [[], [], []]
for each in not_terms:
    nm[ni] = internal_term(each, BiWdIndex)
    not_ids.extend(nm[ni])
    ni += 1

tm = [[], [], []]
ai = 0
# 两层求交集操作
for each in and_terms:
```

```
    tm[ai] = internal_term(each, BiWdIndex)
    ai += 1
and_ids = tm[0]
ii = 1
while ii < 3 and len(tm[ii]) > 0:
    for i in reversed(and_ids):
        # 反向删除
        if i not in tm[ii]:
            and_ids.remove(i)
    ii += 1

or_ids = []
oi = 0
om = [[], [], []]
for each in or_terms:
    om[oi] = internal_term(each, BiWdIndex)
    or_ids.extend(om[oi])
    oi += 1

poemIDs = merge_vector(not_ids, and_ids, or_ids)
count = len(poemIDs)
print("poemIDs: ")
print(poemIDs)
```

如果双词索引中未能找到符合条件的诗词，要继续在位置索引中查找。下面是 search_poem()的后半部分，主要是关于位置索引的查找。每个列表中的每个短语，需要先进行分词，对于分词得到的若干个单词，处理方法类似，写在 merge_with_pos()中，这里直接调用，得到的是该短语对应的是此编号列表，然后与上面类似，or_terms、and_terms、not_terms 的词项间采取相应的操作，最后调用 merge_vector()将 or_terms、and_terms 和 not_terms 得到的诗词编号列表进行合并。

```
# 如果需要的话再进行位置索引
if count == 0:
    # if flag == 1:
    print("----In PosIndex----")
    filepath2 = prefix + "_poem_PosIndex.json"
    with open(filepath2, 'r', encoding='UTF-8') as f:
        PosIndex = json.load(f)
    f.close()

    not_ids2 = []
    for each in not_terms:
        ws = jieba.lcut(each)
        not_ids2.extend(merge_with_pos(ws, PosIndex))
```

```
14.
15.     and_ids2 = []
16.     cn = len(and_terms)
17.     if cn > 0:
18.         ws = jieba.lcut(and_terms[0])
19.         and_ids2.extend(merge_with_pos(ws, PosIndex))
20.         ind = 1
21.         while ind < cn:
22.             and_ids3 = []
23.             ws = jieba.lcut(and_terms[ind])
24.             and_ids3.extend(merge_with_pos(ws, PosIndex))
25.             for i in reversed(and_ids2):
26.                 # 反向删除
27.                 if i not in and_ids3:
28.                     and_ids2.remove(i)
29.             ind += 1
30.
31.     or_ids2 = []
32.     for each in or_terms:
33.         ws = jieba.lcut(each)
34.         or_ids2.extend(merge_with_pos(ws, PosIndex))
35.
36.     return merge_vector(not_ids2, and_ids2, or_ids2)
37. else:
38.     return poemIDs
```

接下来对 search_poem()中调用的 internal_term()进行说明。internal_term()是对参数中的短语进行分词，然后将每两个相邻的单词组合成一个词项，对每个词项的查询结果取交集，返回编号列表。

```
1.  def internal_term(each, BiWdIndex):
2.      ws = jieba.lcut(each)
3.      tmp_ids = BiWdIndex.get(ws[0], [])
4.      tm = tmp_ids
5.      if len(ws) > 1:
6.          # 对该词项进行分词，然后将每两个相邻的单词组合成一个词项
7.          for ww in range(len(ws) - 1):
8.              tmp_ids = []
9.              obj = ws[ww] + ws[ww + 1]  # 构造双词结构
10.             tmp_ids.extend(BiWdIndex.get(obj, []))
11.
12.             ss = set(tmp_ids)
13.             tmp_ids = list(ss)
14.             tmp_ids.sort()
15.             # 对每个词的结果要取交集
```

```
            # 如果不在当前 tmp_ids 中出现，则删去
            if len(ws) == 2:
                return tmp_ids
            for tid in reversed(tm):
                # 反向删除
                if tid not in tmp_ids:
                    tm.remove(tid)
    return tm
```

merge_vector()的参数是三个数组，分别是由前面逻辑词拆分得到的词项的查询结果，记为 not_ids、and_ids 和 or_ids。这里需要对列表进行去重和排序。之后，在 and_ids 中去掉 not_ids 所含的 ID，再与 or_ids 求并集，关键代码如下：

```
len2 = len(not_list)
len1 = len(and_list)
res = []
p1 = 0
p2 = 0
if len2 == 0 and len1 == 0:
    print("仅 or 短语组合")
    return or_list
# 求差集
while p1 < len1 and p2 < len2:
    while p1 < len1 and p2 < len2 and and_list[p1] < not_list[p2]:
        res.append(and_list[p1])
        p1 += 1
    while p2 < len2 and p1 < len1 and and_list[p1] > not_list[p2]:
        p2 += 1
    if p2 < len2 and p1 < len1 and and_list[p1] == not_list[p2]:
        p1 += 1
        p2 += 1
while p1 < len1:
    res.append(and_list[p1])
    p1 += 1

len3 = len(res)
len4 = len(or_list)
p3 = 0
p4 = 0
res_list = []
# 列表求并集
while p3 < len3 and p4 < len4:
    while res[p3] < or_list[p4] and p3 < len3:
        res_list.append(res[p3])
        p3 += 1
```

```
    while res[p3] > or_list[p4] and p4 < len4:
        res_list.append(or_list[p4])
        p4 += 1
    if res[p3] == or_list[p4]:
        res_list.append(res[p3])
        p3 += 1
        p4 += 1
if p3 == len3:
    while p4 < len4:
        res_list.append(or_list[p4])
        p4 += 1
else:
    while p3 < len3:
        res_list.append(res[p3])
        p3 += 1
```

用位置索引查询时，调用了 merge_with_pos()函数，它的功能主要是筛选同一首诗词中单词出现偏移位置满足条件的诗词 ID。具体做法是：对于参数传进来的单词列表，每次取一个单词，用字典存储它涉及到的诗词 ID 及对应的偏移信息，以文章 id 为主键，值为列表，列表中存储关键词出现过的位置。以第一个单词为基准，每次将当前第 m 个单词的字典与它的字典进行对比，若在两个字典中，对于同一篇文章，得到的两个列表中出现位置相差为 m，证明当前 id 符合条件，代码的关键部分如下：

```
tmp_dict = {}
p = index.get(words[0], [])
print("for each word, take out the poemID-postion-list : ")
for li in p:
    print(li)
    if li[0] not in tmp_dict:
        tmp_dict[li[0]] = []
    for i in range(1, len(li)):
        tmp_dict[li[0]].append(li[i])
pids = list(tmp_dict.keys())
if c_word >= 2:
    m = 1
    while m < c_word:
        p = index.get(words[m], [])
        tmp_dict1 = {}
        # print("for each word, take out the poemID-postion-list : ")
        for li in p:
            #     print(li)
            if li[0] not in tmp_dict1:
                tmp_dict1[li[0]] = []
            for i in range(1, len(li)):
```

```
                tmp_dict1[li[0]].append(li[i])
        t2 = list(tmp_dict1.keys())
        for i in reversed(pids):
            # 反向删除
            if i not in t2:
                pids.remove(i)
        for pid in reversed(pids):
            tmp_list = tmp_dict[pid]
            tmp_list.sort()
            tmp_list1 = tmp_dict1[pid]
            tmp_list1.sort()
            j = 0
            flag = 0
            for tid in tmp_list:
                while j < len(tmp_list1):
                    if tid == tmp_list1[j] - m:  # m 表示第 m 个词，同时也是与第一个词的间距
                        flag = 1
                        break
                    j += 1
                if flag == 1:  # PoemID 得到了确认
                    break
            if flag == 0:
                pids.remove(pid)
        m += 1
```

至此，查询功能已经完成，系统可以满足短语、与或非关系混合的查询。为了展示对关键词和世人的分析结果，我将 draw_graph.py 文件生成的 PNG 格式的词云图合成了 GIF，然后编写了 ImageLabel 类，用于显示 GIF，ImageLabel 类显示 GIF 的主要原理是获取 GIF 的所有的帧，每个一段时间，显示下一帧，不断循环。代码如下：

```
class ImageLabel(tk.Label):
    def load(self, im):
        if isinstance(im, str):
            im = Image.open(im)
        self.loc = 0
        self.frames = []

        try:
            for i in count(1):
                self.frames.append(ImageTk.PhotoImage(im.copy()))
                im.seek(i)
        except EOFError:
            pass
```

```
14.
15.         try:
16.             self.delay = im.info['duration']
17.         except:
18.             self.delay = 250
19.
20.         if len(self.frames) == 1:
21.             self.config(image=self.frames[0])
22.         else:
23.             self.next_frame()
24.
25.     def unload(self):
26.         self.config(image=None)
27.         self.frames = None
28.
29.     def next_frame(self):
30.         if self.frames:
31.             self.loc += 1
32.             self.loc %= len(self.frames)
33.             self.config(image=self.frames[self.loc])
34.             self.after(self.delay, self.next_frame)
```

5. 结果展示

本实验中的诗词检索系统支持短语、与或非关系混合的查询，例如：“明月清風 or 平世功名”、“流水 and 行舟 andnot 古渡頭”、“西風立馬頻回首”等。

(1) 输入“古渡頭 or 行舟”，点击“Go”：

图 6 古渡頭 or 行舟（1）

点击“Next”：

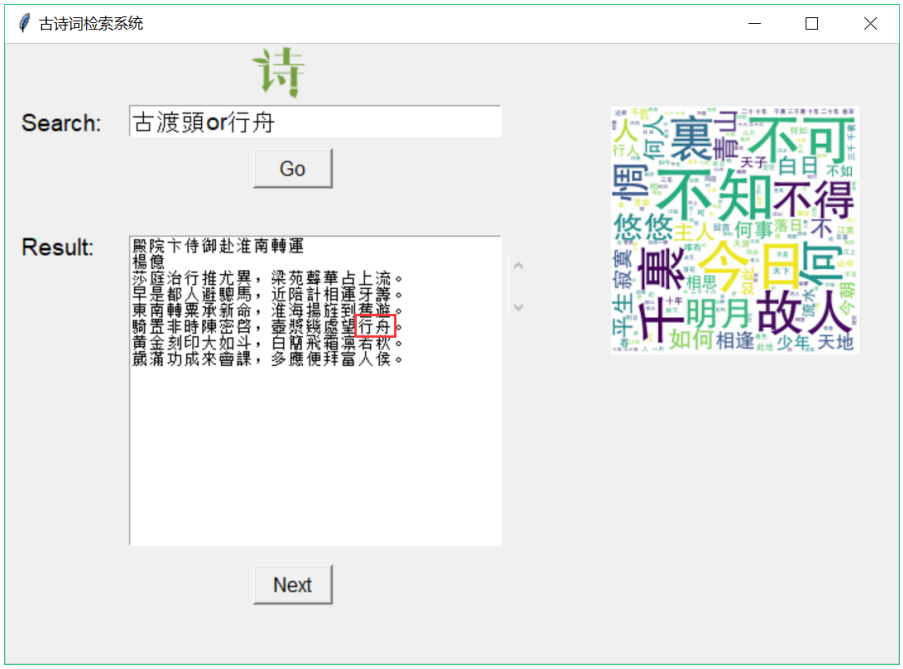

图 7 古渡頭 or 行舟（2）

下面是在这个过程中，控制台的输出信息：

图 8 控制台的输出信息

(2) 输入“流水 and 行舟 andnot 古渡頭”，点击“Go”：

图 9 流水 and 行舟 andnot 古渡頭

(3) 输入“西風立馬頻回首”，点击“Go”：

图 10 西風立馬頻回首

(4) 输入“明月清風 or 平世功名”，点击“Go”：

图 11 明月清風 or 平世功名(1)

点击“Next”：

图 12 明月清風 or 平世功名(2)

(5) 输入“芭蕉雨 and 蘩花地 and 梁州”，点击“Go”：

图 13 芭蕉雨 and 蘩花地 and 梁州

还进行了一些其他的查询测试，记录在表 1 中，不再给出图片。

表 1 查询测试

输入	控制台输出结果
古渡頭 or 行舟	178, 429, 576, 2164, 2610, 2936, 6227……
行舟 andnot 古渡頭	429, 576, 2164, 2610, 6227, 7052, 8366……
流水 and 行舟 andnot 古渡頭	51972
流水 or 行舟 or 古渡頭	25, 37, 167, 178, 196, 246, 429, 576……
春風 or 流水 or 楊柳	25, 37, 167, 178, 179, 196, 218, 223……
古往今來	1713, 1817, 3553, 11886, 12942, 31547……
月 and 水	141, 240, 278, 962, 1713, 1770, 2087……
明月清風	1811, 5687, 17888, 19708, 20052, 26432……
平世功名	10825, 13926, 48936, 62139, 62359, 62499……
明月清風 or 平世功名	405, 869, 1811, 5687, 7248, 9661, 10825……
明月清風 and 平世功名	未搜索到相关内容

6. 实验总结

优点

(1) 支持短语查询、逻辑查询以及短语和逻辑混合的查询
(2) 查询结果的召回率较高
(3) 查询结果的正确率很高

缺点

(1) 程序的鲁棒性不够好，没有做过多的错误捕获和处理，输入的查询字符串需要符合规范，才能正常地进行查询。
(2) 程序有点冗长，查询时间较长，由于用户输入的短语基本都是由两三个短语组成，所以一些地方还可以继续简化。可以换用更高效的数据结构。