系统设计框架

本次课程的结业目标是完成一个简易化的“迷你”搜索引擎，通过在课程上学习老师讲述的知识，同时在网上搜索一些具体的实现，在时间可允许的范围内，尽可能完成一个处理流程类似的迷你搜索引擎。

商用搜索引擎需要处理大规模数据，往往需要分布式运行，且存在数据更新操作；处理网页数据时有筛选出正文，去重等操作；建立倒排索引时需要保存大量的信息；排序时可能使用 PageRank 以及 TF-IDF，位置信息等。由于时间、技术、知识的局限，本次完成的系统均不会有上述的考虑。所以，最后完成的系统，仅仅是一个流程式的 TOY-SEARCH-ENGINE.

本系统分为 4 部分，分别是爬虫，预处理，建立倒排索引，搜索。系统采用 Python 快速开发，使用了第三方 BeautifulSoup 库来快速提出网页内容及链接信息。

爬虫

如概述时所言，互联网可以看作一个图。从图中某些点开始，我们可以使用图遍历算法，就能遍历与这些点相连的所有顶点。即是说，我们选择某些种子网页，采用图遍历算法，遍历一个网页时，我们将网页信息下载下来，并提取出其中的链接，加入到待爬取的网页列表中，接着从待爬取列表中选择一个网页，开始下次遍历。如此循环，直到所有关联的网页都被遍历。为了防止重复爬取，我们需要建立一个已爬取网页列表，在爬取前（或者，在加入到待爬取网页列表前）检查该网页是否已被爬取。如果已经被爬取，我们就应该直接跳过。

在我们的系统实现时，我选择的种子链接是 http://bbs.hupu.com ，采用的图遍历算法是深度优先搜索（使用栈来存储待爬取网页），使用集合数据结构来检测该网页是否已被爬取。为了限制爬虫爬取到其他域名下（因为只想在此域下爬取）以及非网页 URL(如 javescript 操作

或锚记都会出现在 href 字段中)，加入了简单的链接过滤机制。最后，限制了网页爬取数量。

技术上，使用 urllib2 库来下载网页，BeautifulSoup 快速抽取出网页内容及链接，最后，使用 JSON 格式存储结果（读取简单，且结果文件直观可视，便于 DEBUG）。需要说明的是，在用 BeatifulSoup 处理网页时，我们区分性地抽出了标题(head 下的 title 项)的内容和正文(body)内容并分别存储。这是因为在最终计算排序分数时，标题域的权重与正文的权重是不同的。

预处理

预处理部分的主要工作是对得到的文本进行分词处理。目前用于信息处理的数据的基本单位还是词，所以要做信息检索，首先要做的就是将文本进行分词。有了分词后，我们才能够建立词到文档的倒排索引。所有文档中的不重复单词（经过去停用词处理）是倒排索引中的索引项，而文档则是索引对应的值。

具体到实现上，我们首先根据常用的标点符号，将段落进行分句。分句一定程度上是必须的，因为分词程序对输入句子的长度还是有一定的限制的，而且句子的长度影响分词的解码过程，也即是影响分词效果。因此，为了稳妥起见，分句操作是需要做的。完成分句后，我们使用 Pyltp 工具完成分词，得到分词后的文档数据。为了减少存储数据的大小，我们建立了停用词表。停用词表主要来自 GitHub 上的一个项目。我们将所有文档中不重复的词经过停用词过滤后，就得到了词典数据。

总结一下，经过预处理后，我们最终得到的结果包括经分词处理后的文档数据，不重复且经过去停用词的词典数据。

建立倒排索引

倒排索引是检索的关键。为了迅速找到用户查询的文档，我们需要预先建立此种数据结构，

即完成从词到文档的映射。如此，当我们将用户的查询处理变为关键词后，我们只需查询该倒排索引，就可以立刻拿到包含这些关键词的文档。否则，我们不得不去遍历整个文档集合，这样的时间代价将是难以忍受的。

实现时，我们先读入预处理建立的词典，建立索引项。接着，我们依次处理每个文档，对每个文档，记录每个词在文档的标题、正文域出现的频率（TF），出现的位置。与爬虫时类似，标题与正文需要分别对待。处理完一篇文档后，我们将生成的信息作为值以词为 key 加入到索引中。由此即完成了倒排索引。

搜索

搜索的第一步是处理用户的查询。这非常简单，我们只需对其分词即可。用户输入的查询可能是包含空格的（用户常用的一种查询是将关键词以空格分隔），我们可能需要先分隔开，则依次分词处理。

接着我们就需要进行检索了。对用户输入的每个单词，我们通过倒排索引，可以迅速找到包含该单词的网页。将多个单词的检索结果合并起来，得到一个文档集合。

接着要完成的是对得到的文档集合进行打分处理。打分算法我们参照了 Lucene 中的评价函数[2]，如下所示

$$\sum_{t \in q} (tf(t \in d) \times idf(t)^2) \times boost(t.field \in d) \times lengthNorm(t.field \in d)) \times coord(q,d) \times queryNorm(q)$$

其中$tf(t \in d)$表示词t在文档 d 中的出现频率；$idf(t)$表示该词 t 对应的反文档频率，计算公式是$\text{idf} = \log \frac{\#(\text{DOC})}{\#(tDOC)}$，其中$\#(DOC)$表示文档集个数，$\#(\text{tDOC})$表示包含词 t 的文档数；$\text{boost}(\text{t.field}) \in d$可以认为是对每个域的加权，比如说标题域权重为 2，正文域权重为 1；$lengthNorm(t.field \in d)$表示单词 t 所在的域的长度加权，可以这样认为，如果包含该词

的文本越长，其权值应该越小——因为单词越多，说明整个文本与该词的可能的相关性越小，其计算公式为$\frac{\sqrt{1}}{length}$，其中$length$表示域中文本的长度（单词个数）；$coord(q,d)$可以认为表示该文档与查询 query 间有多少个共有的词（即文档包含查询中多少个词语），当然其实际计算公式是一个比例的形式，即$\frac{\#(Q\cap D)}{\#(Q)}$；最后一项是对查询的归一调整，与用户的设置有关。这个值只影响最终的分数大小，但不影响各文档间相对大小，因为该值对所有文档都是相同的。

最后，有了上述的评价标准，我们就可以对文档进行打分了。tf在倒排索引中可以直接查到，idf也直接可以求得，而域的权重我们就认为设置为标题权重为 2，正文为 1，每个域的长度归一化因子也比较容易求得，由此我们把前面的加和项就求出了。最后，$coord$求取也非常简单，这里，因为查询中单词数对所有文档来说都是定值，所以我们也忽略了该值，所以coord分数直接变为文档中包含的查询单词个数。如前所述，我们忽略$queryNorm$项。

有了分数，我们就有了排序结果。最后一步，就是输出结果了。

我们输出包括序号、标题、摘要、URL 信息。其中的难点是摘要的生成。很遗憾，我没有想到比较好的实现算法，时间有限，只能简单地取每个单词出现的第一个位置，将其周围 40 个单词抽取出来作为上下文显示出来。

以上，就是整个搜索模块。整个模块功能比较繁杂，既包括检索，又包括排序，最后还有输出，代码实现时也写得非常糟糕。

完成该模块后，我们还需要完成与用户交互的接口，这里直接在 main 函数中完成，每次让用户输入一个查询，每次只返回 10 条结果，用户可以查看下一页或者开始下次查询。

应用程序具体实现及说明

整个代码已将放到 GitHub 上，点击链接可以直接访问。

爬虫模块

在文件 spider/spider.py 中，我们实现了如下逻辑

```
S = STACK()
VISITED_LIST = SET()
S.PUSH(START_URL)
WHILE ! S.EMPTY() :
    target_url = S.TOP()
    S.POP()
    VISITED_LIST.ADD(target_url)
    page_content = DOWNLOAD(target_url)
    new_url_list = ABSTRACT_URL(page_content)
    title = ABSTRACT_TITLE(page_content)
    content = ABSTRACT_CONTENT(page_content)
    # save data
    SAVE_NEW_DOC(target_url , title , content)
    # add new url
    FOR url IN new_url_list :
        IF FILTER(url) DO              # filter
            CONTINUE
        IF url IN VISITED_LIST DO  # is_visited
            CONTINUE
        S.PUSH(url)
```

预处理模块

预处理模块，我们实现了分词处理：

```
pages = LOAD_DATA()
word_dict = DICT()
stop_words = BUILD_STOP_WORDS()
FOR page_data IN pages :
    tile = page_data.title
    content = page_data.content
    url = page_data.url
    title_seged = SEGMENT(title)
    content_sent_split = SPLIT_SENT(content)
    content_seged = LIST()
    FOR sent IN content_sent_split :
        content_list.EXTEND(SEGMENT(sent))
```

```
    SAVE_SEGED_DATA(url , title_seged , content_seged)
    all_words = title_seged + content_seged
    FOR word IN all_words :
        IF word NOT IN stop_words And word NOT IN word_dict :
            word_dict.add(word)
SAVE_WORD_DICT(word_dict)
```

建立倒排索引

在建立倒排索引阶段，我们依次处理每个文档，从每个文档中记录单词在标题和正文出现频率及位置信息，建立 POSTING 项，最后再添加到倒排索引中。

```
DEF     INVERTED_INDEX
        words2doc = DICT()
DEF     POSTING_ITEM
        doc_id = -1
        title_field_tf = 0
        title_field_pos = []
        content_field_tf = 0
        content_field_pos = []
doc_data = LOAD_SEGED_DATA()
words_dict = LOAD_WORD_DICT()
inverted_index = NEW INVERTED_INDEX()
FOR doc IN doc_data :
    posting_items_dict = DICT()
    doc_id = doc.id
    doc_title = doc.title
    doc_content = doc.content
    FOR word IN doc_title :
        IF word NOT IN posting_items_dict DO
            posting_items_dict[word] = NEW POSTING_ITEM
        posting_items_dict[word].add_title(word)
    FOR word IN doc_content
        IF word NOT IN posting_items_dict DO
            posting_items_dict[word] = NEW POSTING_ITEM
        posting_items_dict[word].add_content(word)
    FOR word IN posting_items_dict :
        IF word NOT IN inverted_index DO
            inverted_index[word] = LIST()
        inverted_index[word].ADD(posting_items_dict[word])
SAVE_INVERTED_DATA()
```

搜索

在搜索模块中，我们首先将用户的查询分词，然后去倒排索引中查询该分词对应的文档列表。

然后按照上节介绍的排序方法对检索结果进行排序。最后，将排序结果构建输出字符窗，输

出给用户。

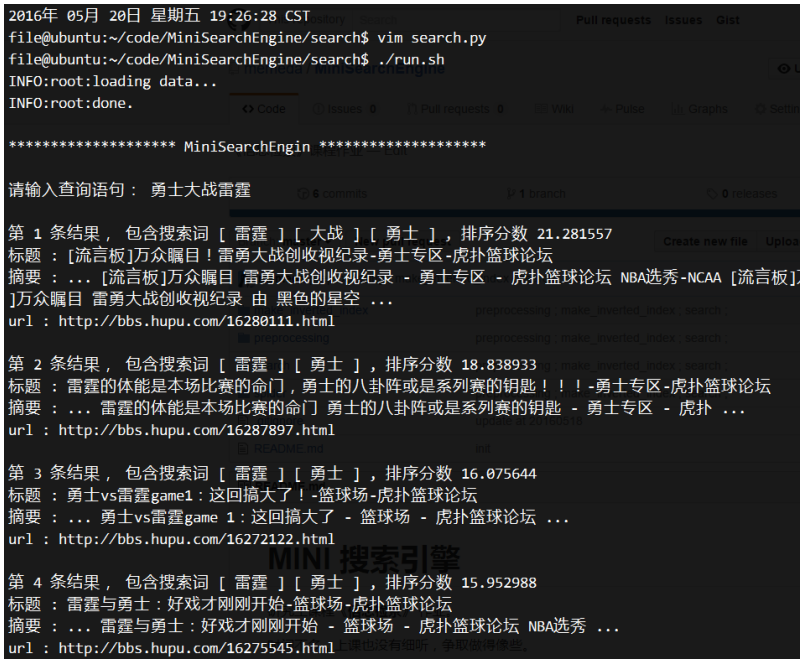
```
doc_data = LOAD_DOC_DATA()
words_dict = LOAD_WORDS_DICT()
inverted_index = LOAD_INVERTED_DATA()

qeury_words = PARSE_QUERY(query)
searched_docs = LIST()
FOR query_word IN query_words :
      searched_docs.EXTEND(inverted_index.GET_DOCS(query_word))
searched_docs.MERGE() # merge doc which has matched to different query word
RANK(searched_docs)
OUTPUT(searched_docs)
```

因为其中细节太过琐碎，所以这里只写出了总体的框架，细节实现见 search/search.py

实验结果及分析

搜索"勇士大战雷霆"的结果截图：



参考文献

[1] 搜索引擎-维基百科，

https://zh.wikipedia.org/wiki/%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E

[2] Lucene 评分算法解释， http://www.hankcs.com/program/java/lucene-scoring-algorithm-explained.html