南开大学
Nankai University

南　　开　　大　　学

信息检索系统原理

搜索引擎实现

姓名：卢艺晗　学号：2213583

2024 年 12 月 18 日

目录

一、 实验目标及完成情况

针对南开校内资源构建 Web 搜索引擎，为用户提供南开信息的查询服务和个性化推荐。功能实现要求主要包含网页抓取、文本索引、链接分析、查询服务、个性化查询五个步骤。结合作业要求，本次实验我共实现了以下功能：

1. **网页抓取：** 主要基于南开大学新闻网、南开大学办公网等官方网页进行爬虫，收集网页和文档总数为 100367。
2. **索引构建：** 借助 Whoosh 工具构建索引，主要有四个索引域：标题、url、正文内容、锚文本。
3. **链接分析：** 借助 networkx 工具构建 pagerank 图，计算每个网页的 pagerank。
4. **查询服务：** 借助 QueryParser 等工具，实现了基于向量空间模型的站内查询、文档查询、短语查询、通配查询，支持网页快照，通过数据库维护查询日志。
5. **个性化检索：** 用数据库维护不同用户的 id 信息、查询历史、兴趣等。用户在注册时提交的三个“兴趣关键词”将用于参与检索结果的加权计算、排序。
6. **个性化推荐：** 实现了基于内容分析的个性化推荐。为每个用户统计历史查询的词频，执行检索，并根据词频加权排序，呈现个性化推荐的内容。
7. **Web 界面：** 借助 flask 的开发者模式搭建网站。主要有四个界面：搜索页面、查询结果页面、登录页面、注册页面。相关页面之间有逻辑跳转。

二、 实验内容

(一) 网页抓取

为了构建搜索引擎资源库，我在南开大学校内网站抓取网页十万余个。抓取网页的技术实现可以分为三个主要部分：网页 url 集合构建–目录页分析–详情页收集。

（1）网页 url 集合构建

为了**高效、纯净**地抓取有价值的.shtml 网页，我最终选择手动构建 url 集合：根据南开大学新闻网、南开大学办公网等的共 17 个板块的网页 url 特点构建 url 集合。

Listing 1: url 集合构建示例

```
url_list = []
## 1.南开要闻
url_list.append('http://news.nankai.edu.cn/ywsd/index.shtml')
for i in range(1, 10):
    url = f'http://news.nankai.edu.cn/ywsd/system/count
        //0003000/000000000000/000/000/c000300000000000000_00000000{i}.shtml
        '
    url_list.append(url)
for i in range(10,100):
    url = f'http://news.nankai.edu.cn/ywsd/system/count
        //0003000/000000000000/000/000/c000300000000000000_0000000{i}.shtml'
    url_list.append(url)
for i in range(100,641):
```

```
    url = f'http://news.nankai.edu.cn/ywsd/system/count
        //0003000/000000000000/000/000/c000300000000000000_000000{i}.shtml'
    url_list.append(url)
```

上述代码是《南开要闻》板块目录页的 url 构建，使用列表 url_list 存储所有 url，在构建时注意分析 url 的构成规律，以及位数等细节处理。

（2）目录页分析

构建了 url 集合后，我们根据集合指定的 url 解析相应的目录页，其中的超链接指向具体的详情页，这是我们要收集的目标网页。

Listing 2: 目录页解析

```
def Spider(url):
    global doc_id,total_num
    if not url_id.get(url,None):#去重
        doc_id +=1
        this_id = doc_id#当前网页的doc_id
        response = requests.get(url,allow_redirects=True)
        if response.status_code not in [200]: #错误响应报错
            print(f'Error in Requesting Page: {response.status_code} - {url}'
                )
            return
        selector = Selector(response)
        title = selector.css('title::text').get()


        # 获取links
        links = selector.css('a::attr(href)').getall()
        for link in links:
            # 跳过邮件
            if link.startswith('mailto:'):
                continue
            if not link.startswith("http"):
                link = urljoin(url, link)
            if not url_id.get(link,None):
                doc_id +=1
                link_id=doc_id
                #对该网页进一步分析
                Page_extract(link,link_id)
```

在上面的代码中，Spider 函数用于解析目录页。我通过一个 url–id 字典来维护网页 url 和 id 的映射关系，同时每次抓取网页时用来判断避免重复。用全局变量 doc_id 唯一地标识每个网页。请求 url 获取响应，首先对网页状态进行检查，只处理状态码为 200 的正常网页。利用 selector 选出所有超链接，它们就是指向目标详情页的链接 link。对于详情页，调用 Page_extrack 函数进一步处理。

（3）详情页收集

在爬虫过程中，我们的主要目标是收集目标网页的 url、title 等基本信息，并保存整个网页（便于后续工作解析及网页快照等功能的实现）。

在 Page_extract 函数中，首先也是根据 url 请求网页，保留响应正确的网页。接下来，根据 url 后缀是.pdf 还是.shtml，选择下载 pdf 文件，或通过 response.text 获取整个网页脚本。最终，将网页或 pdf 保存至本地，并将 url、title、是否是 pdf 的 is_pdf 标志和文件保存路径写入 csv 文件中。

这里我设置网页的保存路径为'h'/'p'+doc_id+'.html'/'pdf'；主要是为了避免标题重复导致文件覆写、url 中的非法字符等，因此用唯一的 doc_id 标识保存路径。

Listing 3: 详情页抓取核心代码

```
save_path = ''
is_pdf = 0
# 下载pdf文件
if real_url.endswith('.pdf'):
    is_pdf = 1
    save_path,title =download_pdf(real_url,this_id)
#保存网页内容
else:
    path_name = f'h{str(this_id)}'
    save_path = f'./FinalPages3/{path_name}.html'
    with open(save_path, mode='w', encoding='utf-8') as f:
        f.write(response.text)
#将信息登记入csv文件
content_df.loc[doc_id] = [url,title,is_pdf,save_path]
total_num += 1
url_id[url]=this_id
print(f'Total: {total_num} - Id: {this_id} - {url}')
```

这样，我们就完成了全部的网页抓取工作。通过 csv 文件记录关键信息，并将整个网页/pdf 文件内容保存至 pages 文件夹。图1显示，pages 中总的网页（含 pdf）数量为 100367，超过 10 万。

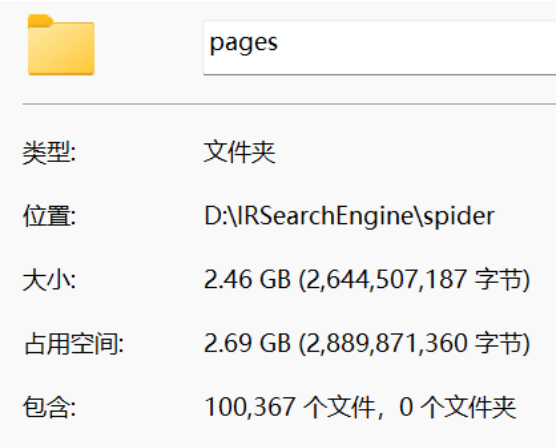

图 1: pages 文件夹属性截图

(二) 索引构建

我借助 Whoosh 工具构建索引，Whoosh 是一个 python 实现的搜索引擎库，提供了强大的全文索引和查询功能，支持倒排索引，且属于轻量级工具，简单高效。

为了构建索引，我需要对之前爬虫保存的网页进行解析，提取出需要的内容——url,title, 正文内容，锚文本。这四点也是我构建索引域的四个方面。

（1）解析网页内容

我遍历 entry.csv 文件，获取每一项的 url、title，并根据记录的网页文件路径读取 html/pdf 文件，并根据不同文件类型以不同的方式提取正文内容和锚文本：

- **html 网页：**经过对网页结构的分析，得知正文内容存储在标签为'td'，id 为'txt' 的块中。使用 BeautifulSoup 提取此块，并通过 get_text 函数过滤掉 html 脚本中的标签等无意义内容，得到的就是正文内容。所有标签为'a' 的块内容即为锚文本。
- **pdf 文件:** 我利用 PyMuPDF 提取 pdf 文档的正文内容。它遍历 pdf 文档的每一页，加载页面并获取纯文本，添加到 content 字符串中，得到完整的 pdf 文本内容。pdf 文档不含锚文本。

Listing 4: 解析网页内容

```
# 读取并解析 HTML/pdf 文件
url = entrys.iloc[i,1] if pd.notna(entrys.iloc[i, 1]) else "NoUrl"
title = entrys.iloc[i,2] if pd.notna(entrys.iloc[i, 2]) else "NoTitle"
is_pdf = entrys.iloc[i,5]
file_path = entrys.iloc[i,6]
if not is_pdf:#对于非pdf的html
    with open(file_path, 'r', encoding='utf-8') as file:
        html_whole_content = file.read()

    soup = BeautifulSoup(html_whole_content, 'html.parser')
    # 提取正文内容 和 锚文本
    content = soup.find('td', id_='txt').get_text(strip=True) if soup.find('
        td', id_='txt') else ""
    anchor_texts = [a.get_text(strip=True) for a in soup.find_all('a')]
    anchor_texts = anchor_texts if anchor_texts else ""
else:#对于pdf文件
    content = ""
    with fitz.open(file_path) as doc:
        for page_num in range(len(doc)):
            page = doc.load_page(page_num)  # 加载页面
            content += page.get_text("text")  # 获取页面的纯文本
    anchor_texts=""
# 正文内容清洗
content = process_text(content) if content!= "" else ""
```

（2）对正文内容进行清洗

为了使得检索更加精准高效，我对提取出的正文内容进行**分词**和**去停用词**处理。Whoosh 工具本身不支持中文分词，因此需要自定义分词方法：

在 process_text 函数中，我对文本内容依次分词、去停用词。使用 jieba 分词包的精确模式，返回分词后的列表；然后使用经典的**哈工大停用词表**，删去无意义的停用词。

Listing 5: 正文内容清洗

```python
import jieba
#去停用词
with open('hit_stopwords.txt', 'r', encoding='utf-8') as f:
    stopwords = f.read().splitlines()
def filter_stopwords(text):
    filtered_words = [word for word in words if word not in stopwords]
    return '␣'.join(filtered_words)

def process_text(text):
    #分词
    words = jieba.lcut(text)
    #去停用词
    content = filter_stopwords(words)
    return content
```

（3）将网页加入索引

创建索引目录，并使用 create_in 函数根据索引目录和 schema 创建索引。我主要对四个方面定义索引域：网页标题 title，网址 path，锚文本 anchr_texts，以及正文内容（为了检索精准，我存储了分此后的 content 段用于检索；为了后续展示方便，我同时也存储了未经处理的正文内容 raw_content 用于在检索结果中展示。）

Listing 6: 索引构建

```python
# 定义索引模式
schema = Schema(
    title=TEXT(stored=True),          # 标题
    content=TEXT(stored=True),        # 正文内容（分词、去停用词后）
    raw_content=TEXT(stored=True),    # 原始内容
    path=ID(stored=True),             # 网址
    anchor_texts=TEXT(stored=True), # 存储锚文本
)
# 创建索引目录
index_dir = "index"
if not os.path.exists(index_dir):
    os.mkdir(index_dir)
# 创建索引
ix = create_in(index_dir, schema)
```

在 with 模式打开 ix.writer() 的情况下，通过 add_document 函数将每个网页的相关内容加入到索引中。

Listing 7: 索引构建

```python
writer.add_document(
    title=title,
    content=cleaned_content,
    raw_content=content,
    path=url,
    anchor_texts="␣".join(anchor_texts) if anchor_texts!= "" else ""
)
```

由此，便实现了索引的构建。

(三) 链接分析

我们通过计算每个网页的 pagerank 评估网页的重要性，用于在提供检索结果时进行排序。

我借助 networkx 工具进行链接分析。networkx 是一个强大的图计算工具，也支持 pagerank 的计算。在 PageRank 中，整个网络可以被表示为一个有向图。每个节点表示一个网页，节点之间的有向边表示网页之间的超链接。NetworkX 提供的 pagerank() 函数默认采用幂迭代方法来计算 pageRank。

为了计算 pagerank，我们需要在解析网页的同时提取其中的超链接，用于图的构建。

Listing 8: pagerank 图的构建

```
# 创建有向图 存储网页之间的链接关系
G = nx.DiGraph()
# .......
# 提取超链接, 构建pagerank图
links = [a['href'] for a in soup.find_all('a', href=True)]
for link in links:
    if not link.startswith('http'):
        link = urljoin(url, link)
    G.add_edge(url, link)
```

在上面的代码中，我们首先创建一个图 G。在遍历每个 html 网页进行解析时，提取所有超链接，通过 urljoin 拼接得到完整的目标网址。通过 add_edge 函数，向图 G 中添加一条从此 url 指向其超链接目标网页 url 的有向边。

Listing 9: pagerank 计算

```
pagerank = nx.pagerank(G)
with open('./pagerank.json', 'w') as f:
    json.dump(pagerank, f, indent=4)
```

遍历完所有网页, 构建好图 G, 最后调用 networkx 中的 pagerank 函数计算每个网页的 pagerank, 并将结果保存到 json 文件中。

如此，完成了所有网页 pagerank 的计算。

(四) 查询服务

1. 站内查询

本实验的搜索引擎支持站内查询，基础查询的实现思路如下:

basic_search 实现基于向量空间模型的网页和文档检索，结合 PageRank 对结果进行排序。首先加载索引和 pagerank 字典; 对于每次查询, 通过 QueryParser 进行解析查询语句 query_str, 生成查询对象。接着使用 Whoosh 的 searcher.search 方法在分词后的正文内容”content “中进行检索，得到相关的文档集合。对检索结果排序：首先按照相关性 score 排序，score 相同时再按 pagerank 排序。这样，会将最符合用户查询、最有价值的内容优先展示出来。

Listing 10: 基础查询

```
#读取索引
```

```
ix = open_dir("./spider/index")
#读取pagerank字典
with open("./spider/pagerank.json", 'r') as f:
    pagerank = json.load(f)
def basic_search(query_str,limit=None):
    with ix.searcher(weighting=scoring.TF_IDF()) as searcher:#加权策略: TF-IDF
        query = QueryParser("content", ix.schema).parse(query_str)# 解析查询
        results = searcher.search(query, limit=None)#执行查询

        # 排序: 先按相关性排序, 再按PageRank排序
        sorted_results = sorted(results, key=lambda r: (r.score, pagerank.get(r['path'], 0)), reverse=True)

        return sorted_results
```

2. 文档查询

本搜索引擎支持文档查询，即用户输入查询语句时，能够通过检索匹配文档内容来返回合适的文档。

实现文档查询最关键的有两步：

（1）解析文档内容，构建索引

本次实验我爬取到的文档主要是 pdf 类型，关于如何解析 pdf 收集正文内容，已经在”索引构建“部分介绍过，在此不再赘述。

（2）执行检索

由于用户往往只关注查询内容而不关注查询结果是网页还是文档，因此我们在构建索引和进行检索时，网页和文档是一起的。我们从含有文档的索引中进行检索，就可以实现文档查询。具体查询方法已经在“站内查询”部分介绍，在此不再赘述。

实现文档检索的效果将在视频中演示。

3. 短语查询和通配查询

我借助 QueryParser 工具实现短语查询和通配查询:

QueryParser 是一个用于解析用户输入查询字符串的工具，它根据指定的字段和索引模式将查询文本转换为可执行的查询对象。在下面的代码中，parse 方法根据一定的语法，将用户输入的查询字符串 query_str 转化为查询对象。

- **短语查询：**如果用户输入的查询是包含空格的短语（如”machine learning”），QueryParser 会将其视为一个短语进行处理，而非两个独立的单词。在倒排索引中，它会寻找包含这两个词且按顺序排列的文档。即，”machine learning” 会匹配包含”machine” 和”learning” 且这两个词顺序相连的文档，而不匹配包含这两个词但顺序不同的文档。
- **通配查询：**QueryParser 也支持通配符查询，允许用户使用如 * 和? 等符号进行模糊匹配。如，查询”apple*” 会匹配以”apple” 开头的所有单词（如”apples”, ”applepie” 等）。? 符号则可以匹配一个单字符，允许进行更精确的匹配。例如，”appl?” 会匹配”apple” 和”applw”。

Listing 11: 查询方法

```
query = QueryParser("content", ix.schema).parse(query_str)
```

具体查询效果将在视频中演示。

4. 网页快照

本搜索引擎为所有查询结果保存网页快照（时间为爬虫时下载网页/文档的时间）；在每个查询结果栏目中都可以点击“网页快照”链接，在浏览器中查看当时的网页状态。

（1）后端支持：

网页快照的实现是为检索结果添加一个'local_snapshot_path' 字段，local_path 是该网页/文档在本地的存储路径（存储在 static 资源下）；通过一个字典记录所有网页 url 到本地存储路径 local_path 的映射，就可以方便地添加此功能。

Listing 12: 网页快照

```
def snapshot(results):
    for result in results:
        url = result['path']
        # 根据url查找对应的本地快照路径
        local_path = url_path.get(url)
        result['local_snapshot_path'] = local_path  # 添加本地路径字段
    return results
```

（2）前端支持：

在每个检索结果栏目，添加一个超链接，url_for 是 flask 用于生成路由对应 url 的方法，这里用于生成静态文件的 url，拼接本地存储路径，用户点击”网页快照“这一锚文本，就可以在浏览器中加载网页快照。

Listing 13: 网页快照

```
<a href="{{ url_for('static', filename=result['local_snapshot_path']) }}"
    target="_blank">
    <small>网页快照</small>
</a>
```

5. 查询日志

本搜索引擎维护一个用户资源数据库，用历史查询表记录所有查询的**用户 id、查询内容、时间戳**。用户可以点击”清空历史记录“来清除所有记录，也可以点击某一条历史记录进行查询。

（1）后端支持：

每当用户执行一次查询，就会将此次查询的语句 query 和用户 id、时间戳存入数据表中。

Listing 14: 记录查询日志

```
conn.execute("INSERT INTO search_history (user_id, query, timestamp) VALUES
    (?, ?, ?)",
                             (session['user_id'], query, time.time()))
```

下面的代码展示了从数据库中加载某个用户的查询日志。查询数据表时，根据 query 进行 group，如果有相同的 query，只提取时间戳最近的那一次记录。

值得注意的是，历史记录表中会记录下每一次查询记录，包括重复的查询。这将为后续的个性化推荐中统计查询频率提供作用。而在网页端的查询日志显示时，不会显示重复的历史记录。

Listing 15: 加载查询日志

```
def get_history_to_display(conn,user_id):
    history = conn.execute("""
        SELECT query, MAX(timestamp) AS latest_timestamp
        FROM search_history
        WHERE user_id = ?
        GROUP BY query
        ORDER BY latest_timestamp DESC
        LIMIT 10
    """, (user_id,)).fetchall()
    return history
```

（2）前端支持：

仅当用户登录后，展示此模块。使用 for 循环，遍历 history 列表，为每一个历史记录条目创建一个列表项，显示历史记录内容。

Listing 16: 查询日志前端显示

```
{% if username %} <!—仅当已登录时显示搜索历史—>
    <p>搜索历史</p>
    <div class="history">
        <ul>
            {% for record in history %}
                <li class="history-item" data-query="{{ record['query'] }}"
                    onclick="searchHistory(this)">
                    {{ record['query'] }}
                </li>
            {% endfor %}
        </ul>
    </div>
    <!— 清空历史记录按钮 —>
    <form method="POST" action="/">
        <button type="submit" name="clear_history" value="1">清空历史记录</
            button>
    </form>
{% endif %}
```

onclick 事件触发 searchHistory 函数，即当用户点击此条记录时，会自动将记录内容填充到搜索框中并执行搜索，跳转到搜索结果页面。

Listing 17: onclick 事件触发的函数

```
// 点击历史记录跳转到相应搜索页面
function searchHistory(element) {
    // 获取 data-query 数据
    const query = element.getAttribute('data-query');
    console.log("Search Query: " + query);

    // 填充搜索框 提交表单
    document.getElementById("query").value = query;
```

```
    document.forms[0].submit();  // 提交表单进行搜索
}
```

图 2: 查询日志 Web 截图

（五） 个性化查询

本搜索引擎支持个性化查询。在每个用户注册时，会收集三个”兴趣关键词”。在每次执行检索时，会将查询结果的相关性 score 和查询结果与”兴趣关键词“的相关性 score 进行加权, 根据加权后的 score，对检索结果排序。

主要实现思路如下:

- **首先，执行基础查询**。将用户输入的字符串根据索引字段进行查询，并将查询结果按照相似度 score 和 pagerank 排序，得到 formatted_results。（此步骤代码同“基础查询”部分，不再展示;
- **其次，从数据库中查询用户的三个兴趣关键词**。
- **接着，对查询结果的前 50 个结果，计算其与各个兴趣关键词的余弦相似度**。这里调用了 updata_weights 函数，对每个检索结果，分别计算他与三个关键词的余弦相似度，将 score 相加，结果存储在 weights 字典中。（PS：考虑到用户可能不会浏览全部的查询结果，为了提升检索性能，在此只对基础检索的排序后前 50 个结果，结合兴趣进行重排序，以此来模拟个性化检索的功能。）
- **最后，将基础查询时得到的 score 与文档与关键词的相似度加权，按照权重排序**。首先对基础查询得到的 score（称为查询 score）进行归一化，然后，将查询结果文档与三个兴趣关键词的相似度 score 取平均值（称为兴趣 score），最终权重 weight= 查询 score*0.8+ 兴趣 score*0.2。按照此权重排序后的顺序即为最终顺序。

Listing 18: 个性化检索函数

```
# 获取用户兴趣
interest1 = conn.execute("""SELECT interest1 FROM users WHERE id = ?""", (
    user_id,)).fetchall()
interest2 = conn.execute("""SELECT interest2 FROM users WHERE id = ?""", (
    user_id,)).fetchall()
interest3 = conn.execute("""SELECT interest3 FROM users WHERE id = ?""", (
    user_id,)).fetchall()
interest_keywords = [interest1[0][0], interest2[0][0], interest3[0][0]]

#个性化排序：计算与兴趣词的余弦相似度
```

```
weights={}
resorted_length = min(50,len(results))
weights=update_weights(interest_keywords, formatted_results[:resorted_length], weights)

# 结合查询score与兴趣，综合加权,重新排序
max_score = sorted_results[0].score
min_score = sorted_results[resorted_length-1].score
base = max_score-min_score

for i in range(resorted_length):
    score= (sorted_results[i].score-min_score)/base
    weights[formatted_results[i]['path']]=weights[formatted_results[i]['path']]*0.2+score*0.8*3
final_results = sorted(formatted_results[:resorted_length], key=lambda x: weights[x['path']], reverse=True)
if len(sorted_results)>50:
    final_results.extend(formatted_results[resorted_length:])
```

(六) Web 界面

本搜索引擎支持 Web 界面搜索，借助 flask 工具搭建网站。由于网页设计不是这门课程的重点，在此我们不再赘述界面的实现，只简要介绍各个界面的功能：

(1) 未登录界面

可以点击左上角进行”登录“或注册”，未登录时不会显示查询日志。

图 3: 未登录界面

(2) 注册界面、登录界面

注册成功、登录成功会有提示，并跳转回到主页面。

注册界面 登录界面

图 4: 注册/登录界面

（3）搜索界面

登陆后，会显示查询日志。可以在输入框中进行查询。可以点击左上角“退出登录”。

图 5: 搜索界面

（4）检索结果界面

左侧显示检索结果。包括检索的问题、结果总数、检索时间，以及各个检索结果（可以点击跳转到对应的网页或网页快照）。右侧显示相关推荐，点击跳转到对应网页。

图 6: 检索结果界面

（七） 个性化推荐

我实现的是**内容分析后的推荐**。主要思路是：统计用户查询历史中的词频，使用历史查询加权后的结果作为相关推荐的网页/文档。

主要实现分为以下几步：

- **从数据库中提取该用户 id 的所有查询历史（含重复的），统计词频：**首先，获取该用户的查询历史列表，使用 Counter 方法统计每个不同查询的出现次数，除以总查询记录个数得到词频。对查询列表排序，提取 query_counter 字典中的键得到不重复的查询列表。
- **对历史记录中词频最高的三个词执行基础查询：**若历史记录不足三，则对所有记录执行查询。这里的查询跳过通配查询。为所有查询结果添加一个字段'weight', 等于此查询语句在历史记录中的频率。接着，将查询结果的相似度 score 与 weight 相乘，最为此查询结果在所有查询中的总 score。
- **对所有查询结果基于总 score 排序，提取前 9 个结果作为相关推荐的条目。**

Listing 19: 个性化推荐的实现

```
    # 提取所有查询历史中的查询词
    query_list = [q['query'] for q in history]
    if len(query_list)==0:
        return []
    length = float(len(query_list))

    # 统计每个查询词的出现次数
    query_counter = Counter(query_list)#次数
    query_frequency = {query: query_counter[query] / length for query in
        query_counter}#频率

    # 对查询列表去重，排序
    all_queries = [item[0] for item in query_counter.most_common()]

    recommendations = []
    limit_n = 10/len(all_queries) if len(all_queries) <3 else 3
```

```
    times = min(3,len(all_queries))
    for i in range(times):#只对频率最高的三个进行查询
        query_text = all_queries[i]
        if '*'in query_text or '?' in query_text:#跳过通配查询
            continue
        with ix.searcher(weighting=scoring.TF_IDF()) as searcher:
            query = QueryParser("content", ix.schema).parse(query_text)
            results = searcher.search(query, limit=limit_n)
            for result in results:
                result_copy = {
                    'title': unquote(result['title']),
                    'path': result['path'],
                    'score': result.score,
                    'pagerank': pagerank.get(result['path'], 0),
                    'weight': query_frequency[query_text]
                }
                result_copy['score']*=result_copy['weight']
                recommendations.append(result_copy)
    # 去重
    recommendations = clean_recommendation(recommendations)
    #将所有结果排序
    sorted_recommendations = sorted(recommendations, key=lambda r: (r['score'
        ], r['pagerank']), reverse=True)

    # 返回最多9个推荐结果
    return sorted_recommendations[:9] if len(sorted_recommendations) >= 9
        else sorted_recommendations
```

三、 总结

本次实验，我实现了一个简单的搜索引擎。实验过程中，学习了爬虫、网页搭建等新的技能，也对信息检索的课内知识如 pagerank 原理、向量空间模型、查询类型等概念进一步巩固。其中，爬虫耗费了大量的时间，而多种工具的配合使用加快了我的开发效率。非常感谢老师、助教和几位同学的指导和帮助，我感到受益匪浅。

不过，实验过程中也有一些不足之处。如个性化推荐、个性化检索等方法的实现，只是相对简易的功能实现，实际上当代个性化检索方面有更多高级的方法，如用户点击率等等都可能参与个性化的内容建构。爬虫时多次重来也让我明白了一个道理：在做事之前必须先明确目标是什么，即在爬虫之前必须要明确具体需要收集哪些信息，有什么用处，这样才能更加高效和顺利。