Web 搜索引擎

姓名：李娅琦
学号：2213603
专业：计算机科学与技术

一. 概述

本次实验针对南开校内新闻构建Web搜索引擎，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：

- 查询服务为用户提供**站内查询、文档查询、短语查询、通配查询、查询日志、网页快照**六种高级搜索功能
- 实现了**注册/登录系统**，通过提供不同的用户信息实现内容排序算法的修正
- 个性化推荐实现的是**搜索上的联想关联**

文件结构如下：

```
├─data/  #网页抓取与捕获，索引的构建
│  ├─catch.py
│  ├─handle.py
│  ├─文档.py
│  ├─t-index.py
│  ├─index.py
│  ├─clean_title-.py
│  └─文档-index.py
└─search/
    ├─__pycache__/
    ├─instance/
    ├─history.json #保存的历史记录
    ├─users.json   #保存的用户
    ├─app.py    #程序入口
    ├─routes/
    │  ├─__pycache__
    │  ├─history.py      #历史记录
    │  ├─personal.py     #个性化推荐
    │  └─search.py       #搜索实现
    └─templates
       ├─log.html        #登陆注册
       ├─index.html      #搜索
       └─snapshot.html   #快照
```

二. 网页抓取

因为在网页快照实现时，需要本地的网页的html，因此在进行网页抓取时，选择抓取网页的源代码并保存到本地，在对保存下来的源代码进行标题，内容，编辑，链接以及文档等内容。

因为数据集要求较大，选择抓取南开大学新闻网中的南开要闻、多彩新闻等等各板块以及南开大学报，并将抓取到的文件保存为csv和json。

但是由于**新闻网上并没有附件文档**，因此又额外选择**https://jwc.nankai.edu.cn/tzgg/list.htm**进行带附件的网页抓取，将附件下载到本地，以便文档查询的进行

1. 网页源代码

使用异步优化，加快爬取速度；使用 **Parsel** 解析 HTML 内容并保存至本地：

```
async def parse_page(url):
    async with sem:
        async with ...:
            if url.startswith('http') or url.startswith('https'):  # 检查 URL 是否
是有效
                response = await client.get(url)  # 异步请求网页内容
                selector = Selector(response.text)  # 使用 Parsel 解析 HTML 内容
                title = selector.css('title::text').get()  # 获取页面标题
                # 保存 HTML 源代码到本地文件
                if title:
                    async with aiofiles.open(f'./html/{title}.html', mode='w',
encoding='utf-8') as f:
                    await f.write(response.text)
```

2. 源代码处理

利用**BeautifulSoup**对网页源代码进行提取，需要提取**标题、正文内容、编辑、url、链接、时间、期数和阅读量**，其中期数和阅读量是由于大学报有的，其他的基本都为空。

以内容、来源时间和链接提取为例：

- 正文内容：

```
# 获取正文内容
content_p = [p.get_text() for p in soup.find_all('p')]  # 获取所有 <p> 标签的文本
content_div = [div.get_text() for div in soup.find_all('div', {'id':
'text_content'})]  # 获取 <div id="text_content"> 的文本
_content = content_p + content_div
content=content.replace...
```

- 来源与发稿时间：

```
source = soup.find('span', style="margin-right:10px;")  # 根据 style 来定位
source = source.get_text(strip=True).replace("来源：", "") if source else ""

publish_time_tag = soup.find('span', string=re.compile(r'发稿时间：(\d{4}-\d{2}-
\d{2} \d{2}:\d{2})'))
publish_time = publish_time_tag.get_text().split("：")[-1].strip()
```

- 页面中链接：

```
def is_valid_link(link):
    """ 检查链接是否符合 http://news.nankai.edu.cn/... 格式，并以数字.shtml结尾 """
    return bool(re.match(r"^http://news\.nankai\.edu\.cn/.+\d+\.shtml$", link))

links = [a['href'] for a in soup.find_all('a', href=True)] #提取所有<a>标签的href属性
filtered_links = [link for link in links if is_valid_link(link)] #过滤
```

3. 附件抓取与下载

对于网页中携带的附件下载链接，包括文档（doc/docx，pdf等），会对网页源代码进行提取，提取的文件名和url链接保存，并在提取时下载文档。

```
# 提取文档链接和文件名
document_links = selector.css('a::attr(href)').getall()
document_titles = selector.css('a::attr(sudyfile-attr)').getall()  # 获取文件的标题信息
document_links = [urljoin(url, link) for link in document_links if link.endswith(('.pdf', '.docx','.doc'))]
document_names = []  # 用于存储文档的名称
# 下载文档到本地
for i, doc_link in enumerate(document_links):
    async with client.stream("GET", doc_link) as stream:
    if stream.status_code == 200:
        doc_path = os.path.join('./文档/下载文档', file_name)
            async with aiofiles.open(doc_path, mode='wb') as f:
                async for chunk in stream.aiter_bytes():
                    await f.write(chunk)
```

三. 文本索引

利用Elasticsearch构建索引，并装载了插件ik分词器。为了兼容，ik与Elasticsearch为同版本。

一般查询

对于一般的高级查询等，非文档查询，构建的索引结构如下：

```
doc = {
    'settings': {
        'analysis': {
            'analyzer': 'ik_max_word',
            "search_analyzer": 'ik_max_word'
        }
```

```
    },
    'mappings': {
        'properties': {
            'url': {'type': 'text'},
            'title': {'type': 'text'},
            'source': {'type': 'text'},
            'publish_time': {'type': 'date'},
            'content': {'type': 'text'},
            'links': {'type': 'keyword'},
            'pagerank': {'type': 'float'}
        }
    }
}
```

文档查询

对于文档查询，需要利用**docx和pymupdf等**对文档内容进行解析，存入elasticsearch中。

内容解析和索引构建如下：

```
# 提取内容
def extract_text_from_pdf(pdf_path):
    doc = fitz.open(pdf_path)
    text = ""
    for page in doc:
        text += page.get_text("text")
    return text
def extract_text_from_docx(docx_path):
    doc = Document(docx_path)
    text = ""
    for para in doc.paragraphs:
        text += para.text + "\n"
    return text
#索引构建
es.indices.create(index=INDEX_NAME, body={
    "mappings": {
        "properties": {
            "title": {"type": "text"},
            "content": {"type": "text"},
            "url": {"type": "keyword"}
            }
        }
})
```

四. 链接分析

因为在对网页源代码提取时，已经提取出页面包含的链接。
在此基础上，利用**networkx的PageRank**模块进行链接分析，评估网页权重，并将其用于返回索引结果的排序中。

```
async def compute_pagerank():
    G = nx.DiGraph()
    for item in index_data:
        main_url = item["url"]
        related_links = item["links"]
        G.add_node(main_url)
        for link in related_links:
            G.add_edge(main_url, link)
    # 计算 PageRank
    pageranks = nx.pagerank(G)
    for item in index_data:
        url = item["url"]
        item["pagerank"] = pageranks.get(url, 0) #无，则默认给 0
```

五. Web界面

登录注册

- 注册：
  - 需要输入用户名，密码，学院和专业
  - 学院与专业是为了个性化查询服务
  - 注册信息会保存在本地 **user.json** 中以便提取
- 登录：
  - 需要输入用户名及密码
  - 确认是否存在对应用户，并匹配密码
  - 登陆成功进入搜索界面

搜索引擎

- 文档搜索单独一个按钮
- 其他搜索统一按钮，根据搜索内容进行类别判定
- 在搜索栏下方显示：
  - 历史记录

    - 随输入进行联想

- 搜索结果页面

    - 点击url可进网站
    - 点击网页快照进入本地html渲染

六. 查询服务

查询类别判定

通过输入进行查询的类别判定，例如输入英文+数字+“/”+“.”代表是一个站内查询。

```
if (/^[a-zA-Z0-9/.]+$/.test(query)) {
    return 'site_search';  // 站内查询
} else if (query.includes(' ')) {
    return 'multi_phrase_search';  // 短语查询(多个term)
} else if (/^[\u4e00-\u9fa5]+$/.test(query)) {
    return 'one_phrase_search';  // 短语查询(一个term)
} else if (query.includes('*') || query.includes('?')) {
    return 'wildcard_search';  // 通配符查询
} else {
    return 'simple_search';  // 普通文档查询
}
```

站内查询

使用通配符模糊匹配 URL:

```
"query": {
    "wildcard": {
        "url": {"value": f"*{query}*"}
    }
}
```

文档查询

对标题和内容做匹配，进行文档查询。查询结果显示文档与文档部分内容。

```
"query": {
    "multi_match": {
        "query": query,
            "fields": ["title", "content"]
        }
}
document = {
    "title": hit["_source"]["title"],
    "url": hit["_source"]["url"],
```

```
        "content": hit["_source"]["content"][:150] + "..." # 提取前150字符作为摘要
    }
```

短语查询

1. 单个term

依次对标题，内容和编辑者做匹配进行查询，查询到的结果标题中先出现的在前面。

```
"query": {
    "bool": {
        "should": [{
            "multi_match": {
                "query": query,
                "fields": ["content","title","editor"],
            }
        },
        {"bool": {
            "should": [
                {"match_phrase": {"content": query}},
                {"match_phrase": {"title": query}},
                {"match_phrase": {"editor": query}},
            ]
        }}
    ]}
}
```

2. 多个term

- 根据空格将查询分割为多个短语
- 构建 `must` 查询（与条件，所有短语必须匹配）
- 构建 `should` 查询（或条件，至少一个短语匹配）
- 与和或都进行查询

- 为整个查询添加 boost给 `must` 查询，使得 “与” 查询优先

因此，结果会优先显示能够匹配的短语更多的。

```
phrases = query.split()
must_clauses = [{"match_phrase": {"content": phrase}} for phrase in phrases]
should_clauses = [{"match_phrase": {"content": phrase}} for phrase in phrases]
# 构建 bool 查询
bool_query = {
    "should": should_clauses,
    "must": must_clauses,
    "minimum_should_match": 1  # 至少匹配一个 should
}
boosted_query = {
    "query": {
        "size": 1000,
        "bool": {
            "must": must_clauses,
            "should": should_clauses,
            "minimum_should_match": 1
        }
    }
}
boosted_query["query"]["bool"]["boost"] = 2.0  # 增加整个查询的 boost
```

通配查询

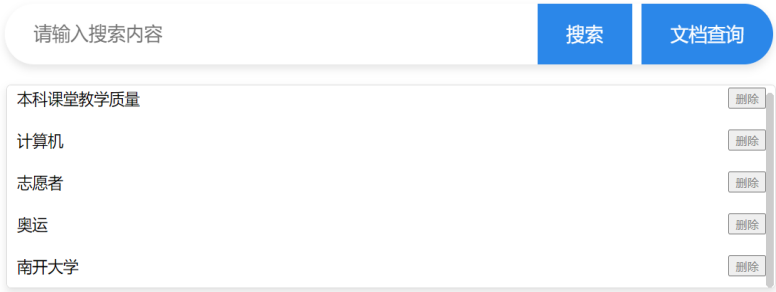
利用 Elasticsearch 自带的通配查询：

```
"query": {"bool": {
    "should": [
        {"wildcard": {"title": {"value": query}}},
        {"wildcard": {"content": {"value": query}}},
        {"wildcard": {"editor": {"value": query}}},
        {"wildcard": {"url": {"value": query}}}
    ]
}}
"sort": [{"_score": {"order": "desc"}}]# 根据得分排序
```

查询日志

- 主体代码在**history.py**文件中
- 搜索页面：
  - **点击搜索框**显示查询日志
  - **正在输入或点击其他位置**查询日志清除
- 记录对应用户的查询历史，并在使用json文件存储于本地，可以回溯
- 个性化查询和个性化推荐会用到

显示的查询日志和被保存的历史记录格式如下：



网页快照

- 查询结果显示时，通过**标题参数调用网页快照路由**
- 根据标题找到本地对应文件夹下的文件
- 返回网页快照，网页快照使用 **Django** 模板渲染

```
@search_bp.route('/snapshot')
def _snapshot():
    title = request.args.get('title')# 从请求中获取标题
    title = title.replace...
    html_file_path = f'./routes/html/{title}.html'
    if not os.path.exists(html_file_path):
        return "网页快照文件不存在"
    # 网页快照
    with open(html_file_path, 'r', encoding='utf-8') as f:
        snapshot = f.read()
    return render_template('snapshot.html', snapshot=snapshot)
```

http://news.nankai.edu.cn/dcxy/system/2021/03/18/030044955.shtml
查看网页快照

计算机学院和网络空间安全学院志愿者体验劳动之美

来源： 南开大学新闻网 发稿时间：2021-03-18 14:45

南开新闻网讯（通讯员 李少歌 薛颖）为帮助学生树立爱绿、护绿意识，倡导生态环保精神，近日，计算机学院和网络空间安全学院学生志愿者赴双港总部经济产业园郭黄庄产业林参观学习，体验树木培植过程，清理垃圾，美化环境。

七. 个性化查询

- 根据用户学院、查询历史等信息为用户提供不同的检索内容排序
- 学院与查询历史在用户查询时都记录在**history.json**中，可通过用户名直接获得
- 使用**lambda**对查询结果进行重排，标题或内容如果包括历史记录或者学院，进行加分
- 返回固定数量的查询结果

```
def reorder_results(results, user_info):
    user_history = user_info.get('history', [])
    user_college = user_info.get('college', '').lower()
    # 给每个结果计算分数
    results_with_scores = []
    for doc in results:
        score = 0
        title = doc['_source'].get('title', '').lower()
        content = doc['_source'].get('content', '').lower()
        # 根据用户历史记录给文档打分
        for term in user_history:
            if term.lower() in title:
                score += 1
            if term.lower() in content:
                score += 5
        # 根据用户的学院给文档加分
        if user_college and user_college in title:
            score += 2
        results_with_scores.append({**doc, 'score': score})
    # 根据分数降序排序，分数高的排在前面
    results_with_scores.sort(key=lambda x: x['score'], reverse=True)
    return results_with_scores[:10]
```

使用与不使用个性化查询的结果对比如下：

八. 个性化推荐

我选择实现的是搜索上的联想。

- 主体代码在**personal.py**
- 跟随输入，下面搜索的联想实时变化
- 根据历史记录和百度进行联想
- 查找包括已输入词为前缀的历史搜索
- 利用**百度智能补全 API**，根据用户的输入返回搜索建议
- 为了提高相关性，对百度API的搜索建议进行重排
    - 按与已输入词为前缀的历史搜索的相似性
    - 使用**Levenshtein** 计算
    - 对联想结果按相似度排序
- 历史记录排在前面，百度补全建议排在其后，返回结果并在前端进行显示

主要的代码如下：

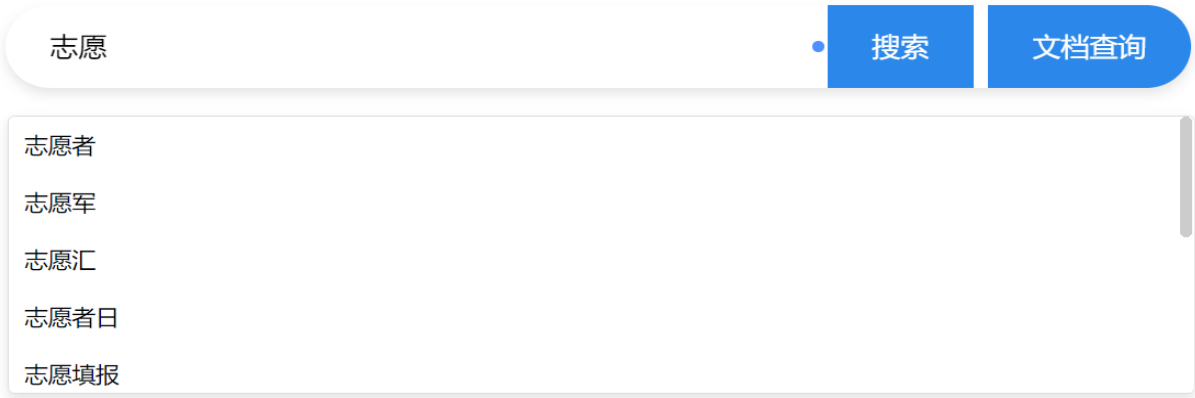
```
def calculate_levenshtein_similarity(term, query):
    distance = lev_distance(term.lower(), query.lower())
    similarity = 1 / (1 + distance)  # 距离越小，相似度越高
    return similarity

def get_baidu_suggestions(user_info, query):
    user_history = user_info.get('history', [])
    try:
        url = f"http://suggestion.baidu.com/su?wd={query}&json=1&p=3"
        headers = {"User-Agent": "Mozilla/5.0"}
        response = requests.get(url, headers=headers, timeout=5)
        response.raise_for_status()  # 检查 HTTP 状态码
        raw_data = response.text
        match = re.search(r'window.baidu.sug\((.*)\)', raw_data)
        if match:
```

```
            data = json.loads(match.group(1))
            return data.get("s", [])
    except Exception as e:
        print(f"Error fetching Baidu suggestions: {e}")
    return []
```

随输入实时向后端发送联想请求如下：

```
127.0.0.1 - - [10/Dec/2024 15:33:17] "GET /personal/get_suggestions?q=z HTTP/1.1" 200 -
127.0.0.1 - - [10/Dec/2024 15:33:18] "GET /personal/get_suggestions?q=zh HTTP/1.1" 200 -
127.0.0.1 - - [10/Dec/2024 15:33:18] "GET /personal/get_suggestions?q=zhi HTTP/1.1" 200 -
127.0.0.1 - - [10/Dec/2024 15:33:18] "GET /personal/get_suggestions?q=志 HTTP/1.1" 200 -
127.0.0.1 - - [10/Dec/2024 15:33:18] "GET /personal/get_suggestions?q=志 HTTP/1.1" 200 -
```

一个个性化联想的结果如下（其中志愿者在搜索历史记录中）：



九. 总结

在实验过程中遇到过挺多问题的，比如搜索的结果不理想。为了避免重复工作，使用了IK和Elasticsearch构建索引。目前的个性化也只是利用了历史记录和学院专业，有待提升。

当然，最大的问题是在个性化联想，本来的想法是寻找热搜词库或者其他的词库，直接放到本地进行读取就可以，但是结果不理想。后来选择了谷歌API，可能因为地区原因效果也不太好。最后选择了百度的API进行使用。