# 实验报告

赵桐
2021201517

2022 年 9 月 5 日

## 1 整体介绍

本次搜索引擎的构建基于中国人民大学后勤集团网页数据，最终采用基于 tf-idf 的 BM25 算法对搜索结果进行排序，并通过 python 的 flask 拓展库进行网页渲染。

在测试样例上，该搜索引擎具有较好的准确度，较高的 mrr 得分。同时我设计了较为简洁美观的 web ui 搜索界面以及结果展示页面，在结果展示页面我将搜索出的网页文章标题，url 以及包含搜索关键词的文字摘要进行了展示。

## 2 实现流程

该搜索引擎的实现流程主要由三个部分组成，分别是**数据爬取**，**算法**以及**网页渲染**

### 2.1 数据爬取

数据爬取采用 python 实现的爬虫程序对后勤集团网页及其域名下的所有网页数据进行爬取，并将网页 html 文件以及网页 url 保存到本地以便后续操作，其中主要使用了 request 库，BeautifulSoup 库以及用于实现多线程的 threading 库。在对网页进行筛选后，最终得到了 8990 个有效网页。

在对网页爬取的过程中，我实现了**单线程**以及**多线程**的爬虫操作。在实现单线程爬虫对网页数据进行爬取时，出于礼貌我设置了 wait 时间进行 sleep。通过对程序的分析，我发现代码运行的大部分时间其实被 sleep 占据，

于是我便想要通过多线程的方法提高对时间的利用率。经过探索与调试之后，我实现了 10 线程并行的爬虫程序。在 wait 仅设为 0.1s 时，单线程大约需要 1700s 完成，而多线程只需约 1400s 完成。

在将网页 html 文件全部保存至本地之后，我另行设计了程序借助 BeautifulSoup 库提取网页中的标题以及正文部分，并将其保存至本地以供后续实现排序算法时计算 tf-idf。

## 2.2 算法

### 2.2.1 布尔查询

在布尔查询部分，我首先构建了一个简单的分词判别函数，然后基于2.1部分保存的网页文本信息进行分词，进而构建倒排索引，即对每个词构建一个变长列表，记录其所出现的文档，最终实现了布尔查询中的 AND 和 OR 查询。

在实现简单的布尔查询后，我从中发现了一些问题。即若采用 AND 查询，很容易造成查询到的网页数量过少；若采用 OR 查询，则可能造成查询到的网页数量过多。同时这两种布尔查询方法都存在一个问题，即返回的网页并没有顺序，无法知道哪个最可能是用户想要查询的网页。为了解决以上问题，我用了接下来学到的知识实现排序检索。

### 2.2.2 排序检索

在排序检索部分，我先后实现了**基于余弦相似度进行排序**，**原始的 tf-idf 加权算法**以及**在原始 tf-idf 算法上进行改进的 BM25 算法**。

在实现算法部分之前，我首先和 2.2.1部分一样对网页文本实现分词。然后对每篇文档中词的 $tf_{t,d}$ 与 $idf_t$ 进行计算，并计算 tf-idf 加权后的文档长度。最终将以上得到的全部信息进行保存，以便后续运行时直接读取，节省时间。

每篇文本中词的 $tf_{t,d}$ 为词在该篇文章中出现的次数，我对其进行对数化处理，这里通过调参我发现采用自然对数的效果最好, 即:

$$tf_{t,d} = \begin{cases} 1 + \ln tf_{t,d} & tf_{t,d} > 0 \\ 0 & tf_{t,d} = 0 \end{cases} \tag{1}$$

用 $df_t$ 来表示出现词 $t$ 的文档数目,$N$ 为待检索集合中的文档数量，则

词的逆文档频率 (idf) 为：

$$idf_t = \ln(N/df_t) \tag{2}$$

同样对其采用对数权重。

接下来对词进行 tf-idf 加权，即：

$$w_{t,d} = tf_{t,d} \times idf_t \tag{3}$$

这种加权方式表明一个词在文档中出现的次数越多，或者这个词在待检索集合中越罕见，那么这个词对相关性分数的贡献越大。接下来基于 tf-idf 加权计算文档的长度

$$doc\_length_d = \sum_{i=0}^{N} (w_{t,d})^2 \tag{4}$$

初次运行后对以上数据进行保存。

接下来首先实现的是**基于余弦相似度进行排序**的算法。该算法将查询语句和每个文档都作为一个高维向量，每个词作为一个坐标轴，词的 tf-idf 权重作为坐标。计算余弦值的公式如下：

$$\cos(\vec{q}, \vec{d}) = \frac{\vec{q} \cdot \vec{d}}{|\vec{q}||\vec{d}|} = \frac{\sum_{i=1}^{|V|} q_i d_i}{\sqrt{\sum_{i=1}^{|V|} q_i^2} \sqrt{\sum_{i=1}^{|V|} d_i^2}}$$

其中 $q_i$ 为第 $i$ 个词在查询中的 tf-idf 权重，$d_i$ 为第 $i$ 个词在文档中的 tf-idf 权重。文本向量的模长在 (4) 处已经进行计算和保存。

基于余弦值进行相似度计算可以很好的避免因两个向量模长相差过大而导致的使用欧氏距离效果很差的问题。但即便如此余弦相似度的效果依旧不是很好，经过分析可知，余弦相似度可能过于优待 tf-idf 权重高的词汇，因为在计算余弦值时将 $w_{t,d}$ 进行了相乘，这会在很大程度上扩大 tf-idf 权重高的词相较于其他词汇的优势。于是我在之后尝试了其它的算法。

在采用余弦相似度算法过后，我尝试了简化算法，即采用**原始的 tf-idf 加权算法**。在该种算法中，每篇文档的得分计算公式如下：

$$Score(q, d) = \sum_{t \in q \cap d} tf.idf_{t,d} \tag{5}$$

该算法在给出的测试样例上效果要好于向量相似度算法，两次测试集上 mrr 得分分别为 0.58 及 0.84。但该算法同样存在问题，即对文档长度考

虑不足，可能会将较长的文档排在前面。考虑到这个问题，结合查询的相关资料，我将原始的 tf-idf 加权算法进行了升级。

升级后的算法为 **BM25 算法**，该算法对于每篇文章的得分计算公式如下：

$$BM25_{t,d} = tf_{t,d} * \frac{(k+1)tf_{t,d}}{tf_{t,d} + k(1-b+b\frac{|d|}{|avdl|})} * idf_{t,d} \tag{6}$$

可以看到在该得分公式中，相较于原始的 tf-idf 加权算法多了一项中间项 $\frac{(k+1)tf_{t,d}}{tf_{t,d}+k(1-b+b\frac{|d|}{|avdl|})}$。该项实际上是对 tf 的结果进行的一种替换。当我们假定 $b=0$ 时，该项退化为 $\frac{(k+1)tf_{t,d}}{tf_{t,d}+k}$。显然该项对词频做了惩罚，随着词频的增加，影响程度会越来越小，最大值为 $k+1$。通过调参，我们通常设定 $k=1.2$。

在中间项中的 $k(1-b+b\frac{|d|}{|avdl|})$ 为对文档长度进行归一化处理的因子，其中 $d$ 为当前查询文档基于 tf-idf 的长度，$avld$ 为所有文档的平均长度。对该项进行分析可知，当文档长度较小时，该项的值较小，由于其在中间项的分母部分，所以导致中间项变大，进而使该文档的得分提高。在一定程度上降低了文档长度对搜索结果的影响。

BM25 算法的 mrr 得分相较于以上实现的两种算法在两次测试集上均有较大提升，同时拥有较高的泛化性能。

# 3 代码细节

## 3.1 数据爬取

在数据爬取部分的单线程爬虫实现并没有特别多的内容，多线程爬虫的实现我主要是借助了 threading 拓展库。同时对单线程爬虫程序做了一些修改。比如将爬虫过程单独定义为一个函数，线程间共用计数变量和 query：

```
def craw():
    global queue # 线程间共用队列
    lock.acquire()
    while len(queue) > 0:  # 控制迭代次数
        global count  # 线程间共用计数变量
        count = count + 1
        url = queue.pop(0)  # 弹出队前一个url
```

以及多线程实现的主要细节如下：

```
lock = threading.Lock()
max_connections = 10
pool_sema = threading.Semaphore(max_connections)
for i in range(10):
    pool_sema.acquire()
    thread = threading.Thread(target=craw)
    thread.start()
    thread_list.append(thread)
for t in thread_list:
    t.join()
```

数据爬取部分的代码主要还是基于课上所给的代码框架。

## 3.2 算法

### 3.2.1 布尔查询

在布尔查询部分，由于我接下来主要依靠的算法部分并没有使用布尔查询，所以我仅仅只是为模板中的布尔查询增添了 OR 查询功能：

```
def logic_or_query(inverted_index, collections, queries):
    l1 = inverted_index[queries[0]][1:]
    for q in queries[1:]:
        l2 = inverted_index[q][1:]
        l1 += intersection(l1, l2)
    results = [collections[docid] for docid in l1]
    return results
```

### 3.2.2 排序检索

在排序检索的算法部分我在报告中仅展示最终采用的 BM25 算法的细节，三种算法的完整代码我会一并上传。

首先是简单的判断分词是否合格的函数：

```python
# 判断分词是否合格的函数
def judge_word(word):
    if len(word.strip()) <= 1:
        return 0
    elif word in stop_words:
        return 0
    elif word.encode('utf-8').isalpha():
        return 0
    elif word.encode('utf-8').isdigit():
        return 0
    return 1


stop_words = []  # 构建停用词表
with open('百度停用词表.txt', 'r', encoding='utf-8') as f:
    for word in f.read():
        stop_words.append(word)
with open('中文停用词表.txt', 'r', encoding='utf-8') as f:
    for word in f.read():
        stop_words.append(word)
```

下面这部分则是对文本进行分词，保存分词结果以免多次分词可能造成分词结果不一样，并保存所有词汇的 $df$ 值：

```python
doc_length = {}
headers = []
word_df = {}  # 储存每个词的df
terms = []  # 储存所有的分词结果
for docid, filename in enumerate(collections):  # 获得每个词的df
    try:
        with open(os.path.join('hqjthtml', filename), encoding='utf-8') as fin:
            terms.append([term for term in jieba.cut(fin.read()) if judge_word(term) == 1])  # 必要的过滤
            terms_set = set(terms[docid])
            for term in terms_set:
                term_docid_pairs.append((term, docid))
                word_df[term] = word_df.get(term, 0) + 1
    except:
        pass
```

接下来则是借助已经保存的分词结果在每篇文章中计算词汇的 tf-idf 权重，并根据 tf-idf 权重计算文档长度，经过调参发现使用自然对数的效果较好：

```
for docid, filename in enumerate(collections):
    try:
        N = word_df.__len__()
        term_counts = np.array(list(Counter(terms[docid]).values()))
        log_tf = np.vectorize(lambda x: 1.0 + np.log(x) if x > 0 else 0.0)
        tf = log_tf(term_counts)
        term_idf = np.array([])
        for word in set(terms[docid]):
            term_idf = np.append(term_idf, word_df[word])
        log_idf = np.vectorize(lambda n, df: np.log(n / df) if df > 0 else 0.0)
        idf = log_idf(N, term_idf)
        w = []
        for i in range(len(tf)):
            w.append(tf[i] * idf[i])
        doc_length[docid] = np.sqrt(np.sum(np.array(w) ** 2))
    except:
        pass
```

基于原始 tf-idf 加权实现的 BM25 算法细节如下：

```
# 基于原始tf-idf加权升级后的算法
def BM25(inverted_index, query, n=3, k=1.2, b=0.75):
    scores = defaultdict(lambda: 0.0)  # 保存分数
    query_terms = Counter(term for term in jieba.cut(query) if judge_word(term) == 1)  # 对查询进行分词
    N = word_df.__len__()
    count = 0
    for length in doc_length:
        count += length
    avdl = count / len(doc_length)  # 计算平均文档长度
    for q in query_terms:
        try:
            postings_list = get_postings_list(inverted_index, q)
            for posting in postings_list:
                w_tf = tf(posting.tf)
                w_td = w_tf
                w_td = w_td * np.log(N / word_df[q])
                w_td = w_td * (k + 1) * w_tf / (w_tf + k * (1 - b + b * doc_length[posting.docid] / avdl))
                scores[posting.docid] += w_td  # 将计算后的w_td累加计算得分
        except:
            continue
    results = [(docid, score) for docid, score in scores.items()]
    results.sort(key=lambda x: -x[1])
    return results[0:n]
```

## 3.3 web ui

根据关键词的 tf-idf 值排序或获取包含该词的摘要：

```
words = [term for term in jieba.lcut(key) if judge_word(term) == 1]
word_idf = {}
for term in words:
    word_idf[term] = get_idf(term)
sorted_word = sorted(word_idf.items(), key=lambda item: item[1])

for url in urls:
    html_txt = get_html(url)
    if html_txt is None:
        results.append({'title': '', 'url': '', 'abstract': ''})
        continue
    soup = BeautifulSoup(html_txt, features="lxml")
    title = soup.find('h1')
    pos = None
    i = 0
    while pos is None:
        pos = soup.find(string=re.compile('.*{0}.*'.format(sorted_word[i][0])), recursive=True)
        i += 1
    if title is None:
        title = soup.title
    title = title.string
```

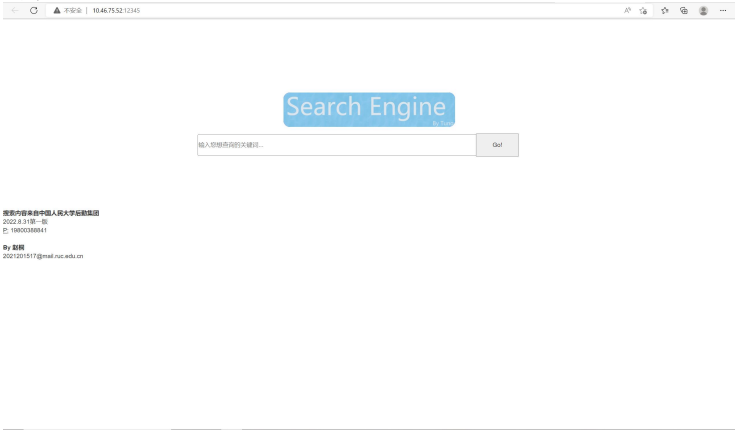
实现关键词标红：

```
for word in words:
    title = title.replace(word, '<span style="color:red">'+word+'</span>')
    pos = pos.replace(word, '<span style="color:red">'+word+'</span>')
tem_dict = {'title': title, 'url': url, 'abstract': pos}
results.append(tem_dict)
```

# 4 界面展示

## 4.1 搜索界面

在搜索界面我简单将页面进行了美化，同时包含了数据来源，联系方式等内容，搜索界面展示如下图：



## 4.2 结果界面

在结果展示界面，我将搜索得到的网页标题，url，包含关键词的文字摘要进行了展示。同时对每个结果页面进行了简要美化，并实现了关键词标红功能。结果界面展示如下图：


# 5 实验感想

总体来说，这八天的集训对我来说还是几乎全都被乐占据了的，可以说是一段让我感觉非常良好，记忆深刻的经历。

一方面我觉得课程任务容量设计的很好，每一天都有每一天的事情做，规划明确。刘勇老师讲课的效果也非常好，我也选择了线下听课，听得更为清晰。而且将总体的任务分散到 7 天来完成，每天的任务量我也觉得刚刚好。虽然有的同学说每天除了课上的 6 个小时，课下还要额外花很多时间来编程，但是其实我每天的话就基本就只有在立德教室的六个小时在完成相关的任务，5 点下课就解放了。而且我的进度说实话好像还是一直领先教学进度的，在第五天的时候我已经开始优化我的 web ui 了，当时就已经把三种算法全部搞好了。相较于我大一下学期所上的程序设计 2 这门课来说，暑期集训的体验可以说是非常好了，当时为了完成程设 2 的大作业，真的是好几天熬夜爆肝连轴转。虽然当时最后也拿到了很高的分数，但是说实话学到的感觉并不是很多，可以说是比这次暑期集训少太多了。所以我感觉这次暑期集训还是给我很好的体验的。

另一方面我感觉课程内容设计的也非常好。在整个课程学习的过程中，我真切地体会到了一步步探索地过程。课程本身的设计便有一个梯度的过

程，逐渐深入，而且非常完整。同时这门课又为我们留下了很多可以深入去探索的地方，比如单线程多线程，比如如何一步步分析已经给出算法的不足，并针对其进行改进，如何自己探索新的算法，如何一步步提高 mrr 得分。这样一个发现问题，探求根本，做出改进，解决问题的过程是非常有成就感的。

在这门课中我也发现我的同学们都非常有想法，和这样一批优秀的同学们，还有我们高瓴的优秀而充满活力与温馨的老师们一起学习生活是一件非常令人心身愉悦的事情。总之，我非常期待接下来在高瓴的生活！