

# Web 实验一 报告

PB20071446 赵钦林：

负责代码整合，报告整合。stage1，stage2合并同近义词、索引压缩、检索、展示

PB20061354 黄与进：

负责stage2 分词、去停用词、相同词合并、布尔表达式预处理

PB20111675 方越：

负责stage2 索引建立，stage3全部工作

报告主要讲解豆瓣电影的处理过程，书籍的处理过程和电影大同小异

## 一、爬虫

### 0. 代码框架：

```
.
|-- README.md          ----> 你在这里
|-- book_spider         ----> 书籍相关的代码
|   |-- doc
|   |   |-- Book_id.txt  ----> 要爬取的书籍id
|   |   |-- json
|   |       |-- info_book.json  ----> 最终解析之后的信息
|   |-- src
|       |-- bucket.py      ----> 无需关注
|       |-- fake_useragent.py  ----> 用户代理池
|       |-- html_parser.py  ----> 页面解析器，对爬下来的源码解析
|       |-- index.py       ----> 主程序
|       |-- spider.py      ----> 爬虫
|-- movie_spider        ----> 电影相关的代码
|   |-- doc              ----> 和书籍的一样，不再注释
|   |   |-- Movie_id.txt
|   |   |-- json
|   |       |-- info_movie.json
|   |-- src
|       |-- bucket.py
|       |-- fake_useragent.py
|       |-- html_parser.py
|       |-- index.py
|       |-- spider.py
```

# 1. 爬虫

本实验使用的爬虫方式为网页爬虫，即向豆瓣发起 `get` 请求，获取网页源码，进一步解析获得相关信息。爬虫核心代码如下：

```
def get_html(self, url):
    # 构造请求头
    headers = {
        'User-Agent': get_ua()
    }
    res = requests.get(url, headers=headers)

    if res.status_code != 200:
        if res.status_code == 404:
            print('unvaild link')
            return 0
        else:
            print('error ! the status code is' + str(res.status_code))
            return -1

    return res.text
```

封装请求头后，发起 `get` 请求，对返回的状态值进行分析，若非正常的200，返回0或-1，跳过当前电影。

## 2. 反爬与应对措施

豆瓣的反爬措施：

- 检测发起请求的是否为浏览器
- 检测是否同一个用户代理高频访问网站
- 检测是否有同一个IP高频访问网站

相应的应对措施如下：

- 构造虚假的 `UserAgent`，伪装成浏览器，向豆瓣发送请求

```
headers = {
    'User-Agent': get_ua()
}
res = requests.get(url, headers=headers)
```

- 构造 `UserAgent-list`，即多个用户代理，每次发起请求时随机挑选一个 `UserAgent`

```

ua_list = [
    'Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Maxthon 2.0',
    'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_0) AppleWebKit/535.11 (KHTML,
    like Gecko) Chrome/17.0.963.56 Safari/535.11',
    ... # 更多见fake_useragent.py
]

def get_ua():
    return random.choice(ua_list)

```

- 经过测试，爬取200部电影后爬虫休息5分钟，不会被封禁IP

```

if num % 200 == 0 and num != 0:
    print('开始必要休眠')
    sleep(300)

```

### 3. 页面解析

页面解析方法以bs4解析为主，正则匹配为辅。为此，实现了一个用于解析的类 `Parser`，具体实现请查看 `./lab1_stage1/movie_spider/src/html_parser.py`

本次实验共解析了9种信息：标题、图片链接、基本信息、剧情简介、演职员表、评分、评论、相关推荐、获奖情况。

下面以电影为例说明解析过程，书籍同理。具体解析方法如下：

#### (1) 解析

在得到一部电影的HTML源码后，首先使用bs4对其进行解析，将HTML结构转换为树形结构。

```

# 使用bs4库对html进行解析
soup = BeautifulSoup(html, 'html.parser')
parser = Parser(soup, movieId)

```

#### (2) 定位

接着初始化 `Parser`。利用 `soup.find()`，结合标签、属性、类型初步定位基本信息、剧情简介等所在位置。

如图，基本信息在 `<div id="info">...</div>` 中

## 肖申克的救赎 The Shawshank Redemption (1994)



导演: 弗兰克·德拉邦特  
 编剧: 弗兰克·德拉邦特 / 斯蒂芬·金  
 主演: 蒂姆·罗宾斯 / 摩根·弗里曼 / 鲍勃·冈顿 / 威廉姆·赛德勒 / 克兰西·布朗 / 更多...  
 类型: 剧情 / 犯罪  
 制片国家/地区: 美国  
 语言: 英语  
 上映日期: 1994-09-10(多伦多电影节) / 1994-10-14(美国)  
 片长: 142分钟  
 又名: 月黑高飞(港) / 刺激1995(台) / 地狱诺言 / 铁窗岁月 / 肖申克的救赎  
 IMDb: tt0111161

豆瓣评分

9.7 ★★  
27365

5星 12.8%  
 4星 1.2%  
 3星 0.1%  
 2星 0.1%  
 1星 0.1%

好于 99% 剧情  
 好于 99% 犯罪

想看 看过 评价: ☆☆☆☆☆

写短评 写影评 分享到



则使用如下代码获取相关信息

```
self.moreInfo = soup.find('div', id="info")
```

### (3) 进一步解析

以获取演员表为例，相关代码如下：

```
def parse_celebrites(self):
    castList = list()
    if not self.celebrities:
        return None
    for item in self.celebrities.find_all('li', class_='celebrity'):
        if not item.text:
            continue
        imgLink = item.div
        imgLink = imgLink['style']
        imgLink = re.findall(re.compile(r'url\((.*?)\)'), imgLink)[0]
        subItem = item.find('div', class_='info')
        person = subItem.find('a', class_='name').string
        work = subItem.find('span', class_='role').string
        personLink = subItem.a
        personLink = personLink['href']
        castList.append([person, work, personLink, imgLink])
    return castList
```

首先部分影片没有演员表，则需要判断是否为空，不为空继续解析。

依据**标签解析** `self.celebrities.find_all('li', class_='celebrity')` 获取每一个演员的信息，再利用**子标签解析**或**正则匹配**来得到最终信息，并加入列表中。

所有影片解析完将结果以json格式存储。

```
[
{
  "基本信息": {
    "标题": "肖申克的救赎 The Shawshank Redemption",
    "图片链接":
    "https://img2.doubanio.com/view/photo/s_ratio_poster/public/p480747492.jpg",
    "评分": [
      "9.7",
      "2703434"
    ],
    "获奖情况": [
      [
        "第67届奥斯卡金像奖",
        "最佳影片(提名)",
        "妮基·马文"
      ],
      ...
    ]
  }
}
```

全部数据请查看 `./lab1_stage1/movie_spider/doc/json/info_movie.json`

## 二、检索

电影和书籍分别建立索引表

### 0. 代码框架

```
.
|-- README.md          ---> 你在这里
|-- __init__.py         ---> 没写，也没用
|-- books               ---> 书籍相关的代码
|   |-- doc
|   |   |-- Book_tag.csv          ---> 新的tag
|   |   |-- new_books_id.txt      ---> 去除坏结点的电影id列表
|   |   |-- no_syn_words_books.json ---> 经过去停用词，合并近义词的分词结果
|   |   |-- posting_list_books.json ---> 书籍的倒排表
|   |   |-- stop_words_books.json ---> 书籍的停用词表
|   |   |-- syno_dict_books.json  ---> 书籍的同近义词表
|   |   |-- words_books.json      ---> 最原始的分词结果
|   |-- src
|   |   |-- parser_books.py        ---> 对书籍信息进行分词
|   |   |-- posting_list_book.py   ---> 建立倒排表的
|-- common
|   |-- bool_inquire            ---> 处理用户输入，显示最终结果
|   |   |-- compress.py          ---> 对压缩后的倒排表解码
|   |   |-- main.py              ---> 主程序
|   |   |-- search.py            ---> 进行布尔查询和合并
|   |   |-- user_input_process.py ---> 对布尔表达式预处理
|   |-- new_label.py            ---> 加新tag的程序
|   |-- synonym
```

```

|         |-- dict_synonym.txt          ----> 最原始的同义词表
|         |-- merge_synonym.py         ----> 进行同义词合并
|         `-- modify_synonym_list.py   ----> 针对特定集合生成特定的同义词表
|-- movies                             ----> 与书籍相同，不再注释
    |-- doc
    |   |-- Movie_tag.csv
    |   |-- new_movies_id.txt
    |   |-- no_syn_words_movies.json
    |   |-- posting_list_movies.json
    |   |-- stop_words_movies.json
    |   |-- syno_dict_movies.json
    |   `-- words_movies.json
    |-- src
    |   |-- __pycache__
    |   |-- parser_movies.py
    |   `-- posting_list_movies.py

```

## 1. 分词

本实验使用北大研发的分词工具[PKUSeg](#), 支持多领域分词, 支持用全新的标注数据来训练模型, 分词效果与其他主流分词工具相比更好:

### 各类分词工具包的性能对比

我们选择jieba、THULAC等国内代表分词工具包与pkuseg做性能比较, 详细设置可参考[实验环境](#)。

#### 细领域训练及测试结果

以下是在不同数据集上的对比结果:

MSRA	Precision	Recall	F-score
jieba	87.01	89.88	88.42
THULAC	95.60	95.91	95.71
pkuseg	96.94	96.81	<b>96.88</b>

WEIBO	Precision	Recall	F-score
jieba	87.79	87.54	87.66
THULAC	93.40	92.40	92.87
pkuseg	93.78	94.65	<b>94.21</b>

电影分词代码如下, 我们选取了标题、剧情简介、演职员、电影类型进行分词。相关代码请查看 `./lab1_stage2/movies/src/parser_movies.py`

分词结果以列表格式存储在 `./lab1_stage2/movies/doc/words_movies.json` 中, 格式如下:

```
[
  [
    "肖申克的救赎",
    "Shawshank",
    "Redemption",
    "剧情",
    "犯罪",
    ...
  ],
  ...
]
```

## 2. 去除停用词

本实验在[中文常用停用词表](#)的基础上，增加了[百度停用词表](#)中的英语停用词。此外，针对电影的特点，补充了“饰”，“配”等高频词语作为停用词。停用词表路径为 `./lab1_stage2/movies/doc/stop_words_movies.json`。去除停用词的代码被整合到分词代码中，每个电影或书籍的关键词列表都会运行一遍下面的代码：

```
key_words_temp = keywords[:]
for member in key_words_temp:    # 去除停用词
    if member in stop_words:
        keywords.remove(member)
```

思路很简单，遍历原始分词结果，如果词在停用词表中，则删除之。

## 3. 相同词语合并

这部分代码也被整合到分词代码中，每个电影或书籍的关键词列表都会运行下面的代码：

```
merged_keywords = []
for word in keywords:    # 合并相同词
    if word not in merged_keywords:
        merged_keywords.append(word)
```

思路也很简单，遍历原始列表并复制元素到新列表中，如果词语已经在新列表中，则跳过。

## 4. 合并同近义词

在1,2,3小节完成后，对产生的 `words_movies.json` 文件进行同近义词合并。

本实验采用[实验文档推荐的同近义词表](#)，基于该表的同近义词组，每组选取一个词作为标准，将遇到的组内其他词转换为“标准词”。“标准词”是该组中在分词结果中出现次数最多的那个词。

统计次数代码如下：

```

while wordList[i+1] != '*':
    synoList.append(wordList[i+1])
    if freqList[i+1] > freqList[max]:
        max = i+1
    i = i+1
if max != begin: # 将频率最高的元素放到第一个
    tmp = synoList[0]
    synoList[0] = synoList[max-begin]
    synoList[max-begin] = tmp

```

原始的同义词表路径为 `./lab1_stage2/common/synonym/dict_synonym.txt`

经过修改的同义词表路径为 `./lab1_stage2/movies/doc/syno_dict_movies.json`

合并同义词后的分词结果路径为 `./lab1_stage2/movies/doc/no_syn_words_movies.json`

合并程序的相关代码请查看 `./lab1_stage2/common/synonym/merge_synonym.py`

经过去除停用词和合并相同词后，movie的分词数量从92525减为90457

经过合并同义词后，词的种类从30743减为23708

至此，数据预处理完成

## 5. 建立倒排表

### (1) 前置操作

通过 stage 1 爬出来的数据建立 `new_movie_id.txt`，此时保存了 `id_num_list[]` 以供后续程序使用。

### (2) 倒排表结构

```

dic = {
    •   key1: [n1, ...],
    •   key2: [n2, ...],
    •   ...
}

```

### (3) 构建倒排表

- 根据分词结果，遍历 `no_syn_words_movies.json` 文件，将分词加入到 `dic` 的关键词 `key` 中。对于每个关键词 `key`，`value` 值为所有出现该分词的 `id` 对应的 `num` 的列表。



```

word_dic = {}
for film_id in range(0, len(input_words)):
    title = input_words[film_id][0]
    id = input_info[film_id]['Id']
    num = id_num_list.index(id)      # 在id_num_list中找到对应的数字（0~986）
    for key in range(0, len(input_words[film_id])):
        if input_words[film_id][key] not in word_dic.keys(): # 第一次加入关键词，初始化value链表并加入id
            tmp_list = []
            tmp_list.append(num)
            word_dic[input_words[film_id][key]] = tmp_list
        else:
            word_dic[input_words[film_id][key]].append(num) # 否则直接加入后面的链表

```

- 对于每个关键词 key 后的列表，利用 `sort()` 升序排列以便构造跳表。

```

for key_n in word_dic.keys(): # 排列n值，从小到大
    word_dic[key_n].sort()

```

## 6. 倒排表压缩

- 在第5步已经进行了一次压缩，即将电影id转换为从0开始的顺序编号。
- 进一步将编号替代为编号间隔进行存储，可以进一步压缩。




关键代码如下：

```

for key, value in word_dic.items():
    for i in range(len(value) - 1, 0, -1):
        value[i] -= value[i - 1]

```

两次压缩的效果如图：

 posting_list_movies_v1.json	2022/11/20 11:56	JSON 源文件	1,723 KB
 posting_list_movies_v2.json	2022/11/20 10:22	JSON 源文件	1,183 KB
 posting_list_movies_v3.json	2022/11/20 10:20	JSON 源文件	1,110 KB

文件大小从1723KB降为1110KB，缩减为原来文件的64%

## 7. 布尔表达式处理与输出

### (1) 布尔表达式预处理

预处理将任意布尔表达式转为与或式，即 `(A AND NOT B) OR (C AND D)` 的形式。同时对用户输入的关键词做同义词替换（因为在建立索引时将部分词转为“中心词”，所以将用户的关键词也转为相应的“中心词”可以更好的匹配）

本实验支持对由AND, OR, NOT, 括号, 关键词组成的**多层嵌套布尔表达式**进行布尔检索, 当多个AND, OR并列出现时, 默认从左到右计算, 如  $A \text{ OR } B \text{ AND } C = (A \text{ OR } B) \text{ AND } C$ 。

代码思路: 从左到右进行扫描, 按照空格分割词语, 根据词语是运算符还是关键词有不同的处理方式, 若遇到括号, 则递归调用自己处理括号内容得到括号内的与或式。扫描完成后根据各关键词和运算符进行处理, 得到整个式子与或式。如[['A', 'NOT B'], ['C', 'D']]表示

$$\overline{AB} + CD$$

相关代码请查看 `./lab1_stage2/common/bool_inquire/user_input_process.py`。

## (2) 检索

检索是基于上一步的与或式查询到符合规则的电影id集合, 以列表形式返回。

为此, 构造查找类 `Searcher`, 相关代码请查看 `./lab1_stage2/common/bool_inquire/search.py`。

为了提高查询效率, 按照**文档频率的顺序**进行处理, 例如在处理多个OR的时候, 首先进行频率排序:

```
while len(res) >= 2: # 均为NOT的情况没有考虑
    res.sort(key=lambda i: len(i), reverse=False) # 对文档进行到底频率排序
    tmp = self.inquire_or(res[0], res[1])
    del res[0:2]
    res.append(tmp)
```

**该模块没有设计跳表, 而是直接使用python提供的in操作进行查询。**因为在实际处理过程中, 对5个左右关键词进行布尔查询, 所需时间不超过0.5s。在这种数据规模较小的倒排表中, 无需增设跳表提高效率。

## (3) 展示

对查询到的每部相关电影, 展示它的标题, 导演, 类型, 简介。界面如下:

```
please input 电影 or 书籍 to search
>>> 电影
this is a search engine about Douban books and movies, please input a bool expression consist of keywords and logical operators:)
example: 你 AND (我 OR NOT 他)
>> 电影 AND 书籍
-----
相关电影1

标题: 肖申克的救赎 The Shawshank Redemption
导演: 弗兰克·德拉邦特 /
类型: 剧情 / 犯罪 / 惊悚 / 喜剧 / 人性 / 文艺 / 爱情 / 青春 / 大陆 / 经典 / 动作 / 动画 / 美国 / 悬疑 /
简介:
    一场谋杀案使银行家安迪(蒂姆·罗宾斯 Tim Robbins 饰)蒙冤入狱, 谋杀妻子及其情人的指控将囚禁他终生。在肖申克监狱的首次现身就让监狱“大哥”瑞德(摩根·弗里曼 Morgan Freeman 饰)对他另眼相看。本片获得1995年奥斯卡10项提名, 以及金球奖、土星奖等多项提名。
其他电影推荐:
    阿甘正传 当幸福来敲门 三傻大闹宝莱坞 摔跤吧! 爸爸 贫民窟的百万富翁 荒岛余生 心灵捕手 国王的演讲 放牛班的春天 美丽心灵
-----
相关电影2
```

考虑到有的查询相关电影很多, 设计分批显示, 一次显示5部电影。

用户可以选择退出或进入下一页。

```
天空之城 哈尔的移动城堡 龙猫 幽灵公主 风
查看下一页请输入next, 退出输入exit
```

## (4) 小组成员展示

找到学号最后两位对应排名的电影。46找排名为46的电影。

- 赵钦林 PB20071446

对应电影：钢琴家 The Pianist (2002)

布尔表达式： 剧情 AND 音乐 AND 传记

共找到7部影片，第1部便是钢琴家

```
please input 电影 or 书籍 to search
>>> 查询
this is a search engine about Douban books and movies, please input a bool expression consist of keywords and logical operators
example: 你 AND (我 OR NOT 他)
>> 剧情 AND 音乐 AND 传记
相关电影共7部
-----
相关电影1

标题: 钢琴家 The Pianist
导演: 罗曼·波兰斯基 /
类型: 剧情 / 音乐 / 传记 / 战争 / 人性 / 文艺 / 爱情 / 经典 / 动作 / 纪录片 / 美国 / 动画 /
简介:
    史标曼（艾德里安·布洛迪 Adrien Brody 饰）是波兰一家电台的钢琴师。二战即将爆发之时，他们全家被迫被赶进华沙的犹太区。在战争的颠沛流离中，家人和亲戚最终其他电影推荐:
    拯救大兵瑞恩 穿条纹睡衣的男孩 辛德勒的名单 1917 兵临城下 血战钢锯岭 敦刻尔克 黑鹰坠落 波斯语课 战争之王
-----
相关电影2

标题: 国王的演讲 The King's Speech
```

- 黄与进 PB20061354

对应电影：指环王1：护戒使者 The Lord of the Rings: The Fellowship of the Ring (2001)

布尔表达式： 彼得·杰克逊 AND 奇幻

找到5部影片，第1部为指环王。

```
please input 电影 or 书籍 to search
>>> 查询
this is a search engine about Douban books and movies, please input a bool expression consist of keywords and logical operators
example: 你 AND (我 OR NOT 他)
>> 彼得·杰克逊 AND 奇幻
相关电影共5部
-----
相关电影1

标题: 指环王1: 护戒使者 The Lord of the Rings: The Fellowship of the Ring
导演: 彼得·杰克逊 /
类型: 剧情 / 动作 / 奇幻 / 冒险 / 科幻 / 人性 / 文艺 / 爱情 / 青春 / 经典 / 美国 / 动画 /
简介:
    比尔博·巴金斯是100多岁的霍比特人，住在故乡夏尔，生性喜欢冒险，在年轻时的一次探险经历中，他从怪物咕噜手中得到了至尊魔戒，这枚戒指是黑暗魔君索伦打造的至尊魔戒，因为和魔戒的朝夕相处，比尔博的心性也受到了影响，在他111岁的生日宴会上，他决定把一切都留给侄子佛罗多（伊利亚·伍德 饰），继续冒险。
    比尔博的好朋友灰袍巫师甘道夫（伊恩·麦克莱恩 饰）知道至尊魔戒的秘密，同时，黑暗魔君索伦已经知道他的魔戒落在哈比族的手中。索伦正在重新建造要塞巴拉多，集结无数甘道夫说服佛罗多将魔戒护送到精灵王国瑞文希尔，佛罗多在好朋友山姆、皮平和梅利的陪同下，在跃马旅店得到了刚铎王子阿拉贡的帮助，历经艰难，终于到达了精灵王国。然而，精灵族并不愿意保管这个邪恶的至尊魔戒，中土各国代表开会讨论，达成意见，准备将至尊魔戒送到末日山脉的烈焰中彻底销毁，佛罗多挺身而出接受了这个任务，这次，一路上，魔戒远征军除了要逃避索伦爪牙黑骑士和半兽人的追杀之外，更要抵抗至尊魔戒本身的邪恶诱惑，前途困难重重。
其他电影推荐:
    指环王2: 双塔奇兵 霍比特人1: 意外之旅 加勒比海盗 哈利·波特与魔法石 少年派的奇幻漂流 勇敢的心 大鱼 奇异博士 倩女幽魂 角斗士
```

- 方越 PB2011675

对应电影：阿凡达 Avatar (2009)

布尔表达式：阿凡达 AND 动作

找到1部电影，就是这部。

```
please input 电影 or 书籍 to search
>>> 电影
this is a search engine about Douban books and movies, please input a bool expression consist of keywords and logical operators:)
example: 你 AND (我 OR NOT 他)
>> 阿凡达 AND 动作
相关电影共1部
-----
相关电影1

标题: 阿凡达 Avatar
导演: 詹姆斯·卡梅隆 /
类型: 动作 / 科幻 / 冒险 / 惊悚 / 喜剧 / 人性 / 爱情 / 青春 / 日本 / 经典 / 香港 / 美国 / 动画 /
简介:
战斗中负伤而下身瘫痪的前海军战士杰克·萨利（萨姆·沃辛顿 Sam Worthington 饰）决定替死去的同胞哥哥来到潘多拉星操纵格蕾丝博士（西格妮·韦弗 Sigourney Weaver
本片采用3D技术拍摄，并耗资5亿美元制作发行，是电影史上最昂贵的作品。本片荣获第82届奥斯卡最佳摄影、最佳视觉效果、最佳艺术指导等3项大奖。

其他电影推荐:
黑官帝国 盗梦空间 变形金刚 星际穿越 终结者2: 审判日 E.T.外星人 头号玩家 钢铁侠 第九区 源代码
```

## 三、个性化检索

### 1. 概要

#### (1) 主要工具选取

- 预测评分算法：选取 `lenskit` 库中的 `algorithms`
- NDCG计算：选取 `sklearn` 库中的 `ndcg_score()`
- 简易图形绘制：选取 `matplotlib` 库中的 `pyplot`

#### (2) 框架构成&基本步骤

```
├lab1_stage3
|   ├──doc                ---->各算法文件夹；原始数据与预处理后的数据
|   |   ├──als
|   |   ├──ii_100
|   |   ├──ii_20
|   |   ├──ii_40
|   |   ├──ii_60
|   |   ├──ii_80
|   |   ├──svd
|   |   └─uu_20
|   ├──fig
|   └─src
```

- 数据预处理(去除组内认为的corner case；以8：2划分训练集与测试集)；
- 利用不同算法/相同算法的不同参数，分别对上述数据进行用户的评分预测；
- 利用 NDCG 指标对上述预测进行评估；
- 对结果进行比较。

本次 stage-3 选择 `Movie_score.csv` 作为数据集，主要使用其中 `user_id`、`item_id`、`score` 三组数据。

### 2. 数据的预处理

## (1) 去除 Corner case

- 通过对原数据集的观察以及对豆瓣评分机制的理解，我们将 `score == 0` 的数据解释为“该用户并未对该电影进行打分”，认为是对预测没有帮助的数据，故删除

```
raw_data = raw_data.drop(raw_data[raw_data.rating == 0].index)
```

- 另一方面，为了排除用户“随意”打分的可能性，我们设想了一种情形为：某名用户对电影的打分基本相同。认为该种数据是不真实的，从数据集中删除

```
for user in range(len(user_id_list)):
    now_user_score = raw_data.loc[raw_data.user == user_id_list[user]]
    total_len = len(now_user_score)
    flag = 0 # 判断该用户是否为坏点
    for i in range(1, 6):
        # 该用户评分为 i 的电影个数
        tmp_df = now_user_score.loc[now_user_score.rating == i]
        i_len = len(tmp_df)
        if i_len/total_len > 0.95:
            flag = 1

    if flag == 0: # 是一个好的数据
        pro_data = pd.concat([pro_data, now_user_score])
```

## (2) 数据集划分

利用 `sklearn.model_selection` 中的函数，对上述处理后的数据集以8: 2进行划分

```
for i in range(len(user_id_list)):
    tmp_user_data = raw_data.loc[raw_data['user'] == user_id_list[i]]
    train_set, test_set = train_test_split(
        tmp_user_data, test_size=0.2, random_state=42)
    train_data = pd.concat([train_data, train_set])
    test_data = pd.concat([test_data, test_set])
```

也可以利用 `lenskit` 中的函数进行数据集划分：

```
for train, test in xf.partition_users(
    ratings[['user', 'item', 'rating']], 5, xf.SampleFrac(0.2)):
    test_data.append(test)
```

## 3. 评分预测与排序

使用 `lenskit` 库，本次实验选取 ALS、SVD、kNN 算法。其中对于 kNN 算法中的基于物品(Item-Item) 算法，选取不同的近邻数 k 值 (20, 40, 60, 80, 100) 来比较优劣。

### (1) 构造评估函数

```
def eval(aname, algo, train, test):
    fittable = util.clone(algo) # 复制这个算法
    fittable = Recommender.adapt(fittable)
    fittable.fit(train)
    # run the recommender
    recs = batch.predict(fittable, test)
    recs['Algorithm'] = aname
    return recs
```

## (2) 得到每个用户的 NDCG 值

值得注意的是，经过数据预处理后的数据可能存在单点的情况，即一名用户只对一部电影有评分。由于实验要求给出用户评分排序，单点情况可以默认已排序，无需进行 NDCG 评估。

```
def get_ndcg(df):

    df = df.sort_values(
        by=['user', 'prediction'], ascending=False)
    user_id_list = []
    ndcg_dic = {}

    # 将所有的user_id全放到一个列表中，用来查找df中的用户
    for i in range(len(df)):
        if user_id_list.count(df.iloc[i, 0]) == 0:
            user_id_list.append(df.iloc[i, 0])
    for user in range(len(user_id_list)):
        tmp_user_data = df.loc[df['user'] == user_id_list[user]]
        pred = list(tmp_user_data['prediction'])
        true = list(tmp_user_data['rating'])
        # 将每个用户的 ndcg 分数存入 ndcg_dic 中
        # 若只有单个评分，则无需考虑使用 ndcg 对其进行评估
        if len(pred) > 1:
            ndcg_dic[user_id_list[user]] = ndcg_score([pred], [true])

    ndcg_df = pd.DataFrame.from_dict(
        ndcg_dic, orient='index', columns=['ndcg'])
    ndcg_df = ndcg_df.reset_index().rename({'index': 'user'}, axis='columns')

    return ndcg_df
```

## (3) 写入文件 (以 Item-Item 为例)

```
for i in range(5):
    k = 20*(1+i)
    algo = item_knn.ItemItem(k)
    n_path = 'lab1_stage3\doc\ii_{}'.format(str(k))
    if os.path.isdir(n_path) == False:
        os.mkdir(n_path)
    pred_data = []
    pred_data.append(eval('II_{}'.format(str(k)), algo, train, test))
    # 得到预测的数据df
    pred_data = pd.concat(pred_data, ignore_index=True)
    pred_data = pred_data.sort_values(
```

```

        by=['user', 'prediction'], ascending=False)
    pred_data.to_csv(
        r'{}\pred_data.csv'.format(n_path), index=False)

    results = get_ndcg(pred_data)

    results.to_csv(
        r'{}\ndcg_results.csv'.format(n_path), index=False)

```

详细数据可以在 lab1\_stage3\doc\algo\_name 文件夹中查询。

## 4. 结果分析

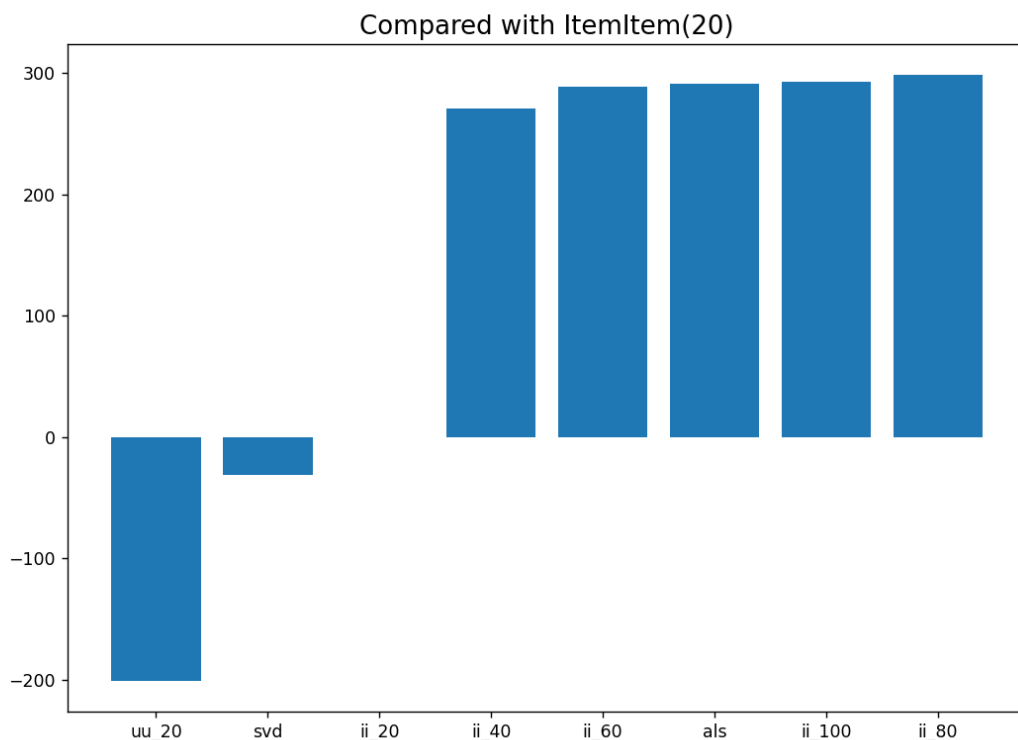
以 Item-Item(20)为基准，在不同的算法中，若某用户的 NDCG 值低于基准的 NDCG，则记为-1，反之+1。

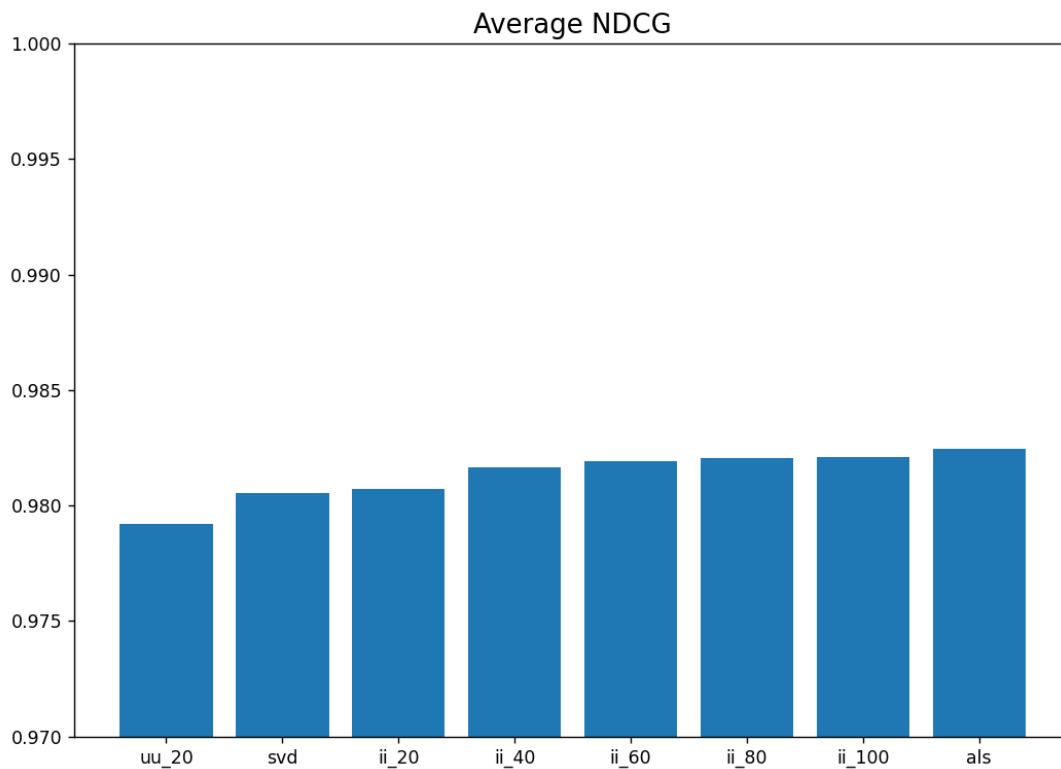
```

for i in algo_list:
    rst_path = "lab1_stage3/doc/{}/ndcg_results.csv".format(i)
    algo_rst_df = pd.read_csv(os.path.abspath(rst_path))
    num = 0
    flag = 0
    for k in range(len(algo_rst_df)):
        if algo_rst_df.iloc[k].iat[1] > base.iloc[k].iat[1]:
            num = num + 1
        elif algo_rst_df.iloc[k].iat[1] < base.iloc[k].iat[1]:
            num = num - 1
    rst_list.append(num)

```

利用 matplotlib.pyplot 绘制直方图：





从该实验中初步得到以下结论：

- 近邻数并不是越大越好。

k值过小，容易受到异常点的影响，易过拟合；k值过大，受到样本均衡的问题，容易欠拟合。

从整体上看，随着 k 的增大，整个用户的平均 NDCG 值增大，也即预测效果趋向于更好；近邻数相对于样本容量过小，如取20，此时增大k值对于评估结果的正确性会有较大提升。

- 基于用户的近邻预测结果欠佳。

推测可能是用户基数较少（预处理后只有535位用户）/用户个性化明显，预测值易受到影响。

- ALS 算法比较优秀。

ALS 同时考虑到 Item 和 User 两方面，所以综合性更好，相比起单独的 User-User 或者 k 值较小的 Item-Item 算法具有更高的 NDCG值。