

实验报告

实验目标

- ① 爬虫与检索
 - 爬取指定的电影、书籍主页，解析并保存基本信息
 - 结合给定的标签信息，实现电影和书籍的检索
- ② 推荐系统
 - 结合用户评价信息与社交网络，进行个性化电影、书籍推荐

Part1

本部分提交的文件目录如下：

```

├──part1
│   ├──search
│   │   ├──data
│   │   │   ├──cn_stopwords.txt    # 停用词表
│   │   │   ├──dict_synonym.txt    # 同义词表
│   │   │   └──
│   │   ├──res
│   │   │   ├──index.txt    # tag-index 索引表
│   │   │   ├──inverted_index.txt    # 倒排索引表
│   │   │   ├──inverted_index_compress.txt    # 压缩后的倒排表
│   │   │   └──tags.txt # tag 数据集
│   │   └──src
│   │       ├──index_compress.py    # 建立压缩版倒排表
│   │       ├──index_init.py    # 建立倒排表
│   │       ├──search.py    # 检索
│   │       ├──search_compress.py    # 依据压缩的倒排表检索
│   │       └──tags_init.py    # 生成 tag 数据集
│   └──spider
│       ├──data
│       │   ├──Book_id.csv # 待爬取的书籍 id
│       │   └──Movie_id.csv    # 待爬取的电影 id
│       └──res
│           ├──book.csv # 书籍信息爬取结果
│           └──error.txt # 404 的 id

```

```

|      |      movie.csv    # 电影的爬取结果
|      |
|      |      └─src
|      |
|      |      book_spider.py  # 书籍爬虫
|      |      get_fake_ip.py  # 生成 ip 池
|      |      get_fake_ua.py  # 生成 ua 池
|      |      ip_info.txt    # ip 池
|      |      movie_spider.py # 电影爬虫
|      |      ua_info.txt    # ua 池

```

爬虫

“

源码路径: `./part1/spider`

我们选择了网页爬取，主要使用 `requests` 库获取网页，用 `BeautifulSoup` 模块辅以 `re` 库进行内容解析，解析结果以文件形式存储。另外，用代理 IP 和生成的 `user-agent` 来应对反爬，用多线程并行来加速爬虫效率。

反爬策略

最基本的反爬策略当然就是生成 `User-Agent` 伪装成浏览器发送请求，直接使用 `fake_useragent` 库即可生成，保存到 `ua_list` 列表中供选用。

但由于我们要爬取的页面数很大，而且因为用了多线程，爬取的速度很快，所以豆瓣很快就封了我们的 IP。因此，我们又写了另一个爬虫，用于爬取网上的免费 ip 代理，测试后选择可用的构建 IP 代理池，保存到 `ip_list` 列表中供选用。

每次访问网页时，随机选择一个 `ua` 和 `ip` 组合，传入 `get()` 函数请求网页，若返回的代码是 403，说明 `ip` 又被封了，就换个 `ip` 重试。

虽然成功的爬取了所有数据，但是爬完之后我主机 `ip` 还是被豆瓣封了，推测因为是网上的免费代理匿名性不够。

相关代码片段如下：

```

1  ua_list = get_fake_ua.get_ua_pool()
2  ip_list = get_fake_ip.get_ip_pool()
3  def create_headers(self):
4      ua = random.choice(ua_list)
5      headers = {
6          'User-Agent': ua
7      }
8      return headers

```

```

9  def create_proxies(self):
10     self.ip = random.choice(ip_list)
11     proxies = {
12         'http': 'http://' + self.ip
13     }
14     return proxies
15 def get_html(self, url):
16     headers = self.create_headers()
17     proxies = self.create_proxies()
18     try:
19         response = requests.get(url, headers=headers, proxies=proxies)
20     except:
21         return None
22     return [response.text, response.status_code]

```

解析方法

为了方便保存结果，先设计一本书籍的数据结构如下：

```

1  book = {
2      'id': '',
3      'name': '',
4      'author': '',
5      'date': '',
6      'score': (), # (评分, 评价人数)
7      'intro': '',
8      'author_intro': '',
9      'recommend': [] # 推荐书籍的 id 列表
10 }

```

对于网页内容的解析，第一版我们直接使用 `re` 库进行正则匹配，但效果不好，因为网页源码格式不太统一，正则表达式不是很好写，经常匹配不到或匹配错误，而且解析速度比较慢。

之后我们换用了 `BeautifulSoup` 进行解析，可以快速方便的分离标签和文本内容，然后通过 `soup.find()` 函数来查找相应标签对应的内容。`BeautifulSoup` 是一个解析器，可以将网页解析为树状结构，可以通过标签查找到对应内容，省去了我们编写正则表达式的麻烦。对解析出的文本，使用简单正则表达式就可以提取出我们需要的内容，然后保存到文件中即可。

相关代码片段如下：

```

1  def parse_html(self, html, book_id):
2      # beautifulsoup 解析

```

```

3     soup = BeautifulSoup(html, 'html.parser')
4     # 书名
5     self.book['id'] = book_id
6     name = soup.find('span', attrs={'property': 'v:itemreviewed'})
7     if not name:
8         return
9     self.book['name'] = name.text
10    # info
11    info = soup.find('div', attrs={'id': 'info'}).text
12    # print(info)
13    # 从 info 中匹配出作者和出版年份
14    # 可能没有
15    content = re.findall('作者:(.*?)出', info, re.S)
16    if not content:
17        author = ' '
18    else:
19        author = content[0].replace('\n', ' ')
20    self.book['author'] = ' '.join(author.split()) # 去除多余空格

```

多线程

每次爬取一个页面效率有点低，几秒钟才能爬取一个页面，因此我们用 `threading` 库实现了多线程处理，效率提升了很多。为了防止爬取重复或遗漏，我们将 1200 条 id 分为了 12 份不重叠的组，每组由一个爬虫处理，共 12 只爬虫，理论上爬取速度最多可以提高 12 倍。另外，某些操作需要保证原子性，所以还需要通过 `Lock()` 函数加锁。

相关代码片段如下：

```

1 # 多线程执行
2 spider = []
3 threads = []
4 for i in range(0, 12):
5     spider.append(BookSpider(list[100*i : 100*i + 100]))
6     threads.append(threading.Thread(target=spider[i].main))
7 for t in threads:
8     t.start()
9 for t in threads:
10    t.join()
11 print('\nCongratulations!\n')

```

检索

“

源码路径: `./part1/search`

我们先比较了 `jieba` 和 `hanlp` 两种分词工具的效果, 然后选用了 `jieba` 进行分词, 把电影或书籍的剧情简介分为了一个个关键词, 然后去除同义词和停用词之后作为 `tag`, 然后建立倒排索引表,

分词

我们首先写了个测试函数 `test.py`, 分别用 `jieba` 和 `hanlp` 两种分词工具对同一个剧情简介进行分词, 计算分词时间, 观察分词效果。

运行结果如下:

```
HanLP分词时间: 4.487566878556641
```

```
HanLP分词结果:
```

```
《/ 青鸟/ 》/ 是/ 梅特林克/ 的/ 最/ 著名/ 的/ 代表作/ 。/ 原作/ 是/ 直到/ 今天/ 仍/ 在/ 舞台/ 上/ 演出/ 的/ 六/ 幕/ 梦幻剧/ , / 后/ 经/ 梅特林克/ 同意/ , / 他/ 的/ 妻子/ 乔治特·莱勃伦克/ 将/ 剧本/ 改写/ 成/ 童话/ 故事/ , / 以便/ 更/ 适合/ 小/ 读者/ 阅读/ 。/ 改编/ 成/ 的/ 中篇/ 童话/ 《/ 青鸟/ 》/ 在/ 1908年/ 发表/ 。/ 故事/ 讲述/ 两/ 个/ 伐木/ 工人/ 的/ 孩子/ , / 代表/ 人类/ 寻找/ 青鸟/ 的/ 过程/ 。/ 青鸟/ 在/ 这里/ 是/ 幸福/ 的/ 象征/ 。/ 通过/ 他们/ 一路上/ 的/ 经历/ , / 象征性/ 地/ 再现/ 了/ 迄今为止/ , / 人类/ 为了/ 寻找/ 幸福/ 所/ 经历/ 过/ 的/ 全部/ 苦难/ 。/ 作品/ 中/ 提出/ 了/ 一个/ 对/ 人类/ 具有/ 永恒/ 意义/ 的/ 问题/ : / 什么/ 是/ 幸福/ ? / 但是/ 作品/ 所/ 得出/ 的/ 结论/ 却/ 是/ 出乎意料/ 的/ : / 其实/ 幸福/ 并/ 不/ 那么/ 难/ 找/ , / 幸福/ 就/ 在/ 我们/ 身边/ 。
```

```
jieba分词时间: 0.5829215812683105
```

```
jieba分词结果:
```

```
/ 《/ 青鸟/ 》/ 是/ 梅特林/ 克/ 的/ 最/ 著名/ 的/ 代表作/ 。/ 原作/ 是/ 直到/ 今天/ 仍/ 在/ 舞台/ 上/ 演出/ 的/ 六幕/ 梦幻/ 剧/ , / 后/ 经/ 梅特林/ 克/ 同意/ , / 他/ 的/ 妻子/ 乔治/ 特/ ·/ 莱勃/ 伦克/ 将/ 剧本/ 改写/ 成/ 童话故事/ , / 以便/ 更/ 适合/ 小/ 读者/ 阅读/ 。/ 改编/ 成/ 的/ 中篇/ 童话/ 《/ 青鸟/ 》/ 在/ 1908/ 年/ 发表/ 。/ 故事/ 讲述/ 两个/ 伐木工人/ 的/ 孩子/ , / 代表/ 人类/ 寻找/ 青鸟/ 的/ 过程/ 。/ 青鸟/ 在/ 这里/ 是/ 幸福/ 的/ 象征/ 。/ 通过/ 他们/ 一路上/ 的/ 经历/ , / 象征性/ 地/ 再现/ 了/ 迄今为止/ , / 人类/ 为了/ 寻找/ 幸福/ 所/ 经历/ 过/ 的/ 全部/ 苦难/ 。/ 作品/ 中/ 提出/ 了/ 一个/ 对/ 人类/ 具有/ 永恒/ 意义/ 的/ 问题/ : / 什么/ 是/ 幸福/ ? / 但是/ 作品/ 所/ 得出/ 的/ 结论/ 却是/ 出乎意料/ 的/ : / 其实/ 幸福/ 并/ 不/ 那么/ 难/ 找/ , / 幸福/ 就/ 在/ 我们/ 身边/ 。
```

可以看出, `jieba` 分词速度明显快得多。分词效果差不多, 但对于英文人名的分词, 比如梅特林克和乔治特-莱勃伦克等, `hanlp` 效果明显更好。综合考虑, 我们选择了 `jieba` 作为本次实验的分词工具, 因为使用方便效率高。

分词之后根据停用词表和同义词表筛选即可得到 `tag` 数据, 保存到 `tag.txt` 中。

相关代码片段如下:

```
1 #预处理停用词
2 stopwords = []
3 stopwords_path = './part1/search/data/cn_stopwords.txt'
4 with open (stopwords_path, 'r', encoding='utf-8') as f:
5     stopwords = [line.strip() for line in f.readlines()]
6 #预处理同义词近义词
7 combine_dict = {}
```

```

8 for line in open("./part1/search/data/dict_synonym.txt", "r",
encoding='utf-8'):
9     seperate_word = line.strip().split(" ")
10    num = len(seperate_word)
11    for i in range(2, num):
12        combine_dict[seperate_word[i]] = seperate_word[1]
13    #获取sentence的tag
14    def get_tag(sentence):
15        sentence = strQ2B(sentence)
16        text = jieba.lcut(sentence)
17        clean_text = [word for word in text if word not in stopwords]
18        combined_text = []
19        for word in clean_text:
20            if word in combine_dict:
21                word = combine_dict[word]
22            combined_text.append(word)
23        exist_dict = []
24        final_tags = []
25        for word in combined_text:
26            if not (word in exist_dict):
27                exist_dict.append(word)
28                final_tags.append(word)
29        return final_tags

```

倒排表

有了 `tag` 数据集之后，只需要遍历一遍就可以得到索引表 `index.txt`（这里索引从 1 开始，而不是直接用 `id`，这样可以节省空间）。再遍历一遍索引表，对索引表的 `key` 和 `value` 反转即可得到倒排索引表 `inverted_index.txt`。由于数据量并不是很大，所以我们这里并没有使用跳表指针（或者说是跳表指针长度为 1，实测检索速度已经很快了）。

相关代码片段如下：

```

1 with open('./part1/search/res/tags.txt', 'r', encoding='utf-8') as f:
2     list = f.read().splitlines()
3     # 生成索引
4     index = {}
5     i = 1
6     for item in list:
7         index[str(i)] = item
8         i += 1
9     with open('./part1/search/res/index.txt', 'w', encoding='utf-8') as f:

```

```

10     for key, value in index.items():
11         f.write(key + ' ' + value + '\n')
12 # 生成倒排索引
13 inverted_index = {}
14 # 将值作为 key, 将 key 作为值
15 for key, value in index.items():
16     for tag in value.split(' '):
17         if tag == '':
18             continue
19         if tag not in inverted_index:
20             inverted_index[tag] = []
21         inverted_index[tag].append(key)
22 with open('./part1/search/res/inverted_index.txt', 'w', encoding='utf-8')
23 as f:
24     for key, value in inverted_index.items():
25         f.write(key + ' ' + ' '.join(value) + '\n')

```

查询

处理 bool 表达式, 因为 `or` 的优先级最低, 所以可以通过 `or` 将表达式分隔为若干只含 `and` 或 `not` 的部分分别处理, 然后取并集即可。对于只含 `and` 或 `not` 的部分, 首先分离出 `not` 对应的 tag, 然后剩下的 tag 分别检索, 取交集, 最后与分离出的 tag 的检索结果取差集。最后根据检索出的 id, 回到 `book.csv` 文件中查找相关信息并返回给用户。

相关代码片段如下:

```

1 # 获取倒排索引
2 inverted_index = {}
3 with open('./part1/search/res/inverted_index.txt', 'r', encoding='utf-8')
4 as f:
5     lines = f.read().splitlines()
6     for line in lines:
7         inverted_index[line.split(' ')[0]] = line.split(' ')[1:]
8 # 处理 and 和 not
9 def search_and(query):
10     tags = query.split('and')
11     not_list = []
12     for tag in tags:
13         if 'not' in tag:
14             not_list.append(tag.split('not')[1].strip())
15             tags.remove(tag)
16 # 可能不存在

```



```

16     try:
17         result = set(inverted_index[tags[0].strip()])
18     except:
19         return set()
20     for tag in tags:
21         result = result & set(inverted_index[tag.strip()])
22     for tag in not_list:
23         result = result - set(inverted_index[tag.strip()])
24     return result
25 # 处理 or
26 def search(query):
27     tags = query.split('or')
28     result = set()
29     for tag in tags:
30         result = result | search_and(tag)
31     return result

```

运行结果如下图:

请输入查询语句: (q 退出)

乡村 and 爱情 or 动作 and not 武侠 or 科幻

查询结果:

书籍: 麦田里的守望者

简介: 霍尔顿是出身于富裕中产阶级的十六岁少年, 在第四次被开除出学校之后, 不敢贸然回家, 只身在美国最繁华的纽约城游荡了一天两夜, 住小旅店, 逛夜总会, 滥交女友, 酗酒……他看到了资本主义社会的种种丑恶, 接触了各式各样的人物, 其中大部分是“假模假式的”伪君子。霍尔顿几乎看不惯周围发生的一切, 他甚至想逃离这个现实世界, 到穷乡僻壤去假装一个又聋又哑的人, 但要真正这样做, 又是不可能的, 结果他只能生活在矛盾之中: 他这一辈子最痛恨电影, 但百无聊赖中又不得不在电影院里消磨时间; 他厌恶没有爱情的性关系, 却又糊里糊涂地叫来了妓女; 他讨厌虚荣庸俗的女友萨丽, 却又迷恋她的美色, 情不自禁地与她搂搂抱抱。因此, 他尽管看不惯世道, 却只好苦闷、彷徨, 用种种不切实际的幻想安慰自己, 自欺欺人, 最后仍不免对现实社会妥协, 成不了真正的叛逆, 这可以说是作者塞林格和他笔下人物霍尔顿的悲剧所在。

书籍: 银河系漫游指南

简介: 地球被毁灭了, 因为要在它所在的地方修建一条超空间快速通道。主人公阿瑟·邓特活下来了, 因为他有一位名叫福特·长官的朋友。这位朋友表面上是个找不着工作的演员, 其实是个外星人, 是名著《银河系漫游指南》派赴地球的研究员。两人开始了一场穿越银河的冒险, 能够帮助他们的只有《银河系漫游指南》一书中所包括的无限智慧。旅途中, 他们遇上了一批非常有趣的同伴: 赞福德·毕博布鲁克斯: 长着两个头、三条胳膊的银河大盗, 他的另一个身份是银河帝国总统。崔莉恩: 赞福德的同伴, 除阿瑟·邓特之外唯一幸存下来的地球人。事实上, 阿瑟从前认识崔莉恩, 而且曾经试图勾搭人家, 可惜没有成功。马文: 天才机器人, 疑心病极其重, 极其沮丧, 极其唠叨。 这些人物结成一个小团队, 他们将揭开一个骇人听闻的大秘密……

书籍: 美国众神

简介: 《美国众神》描述主人公——影子从监狱释放后, 穿越美国大陆的旅行过程中的一系列奇遇。讲他与生活在美国土地上的各种神祇相遇, 由此引发了出许多精彩动人、奇诡绚丽的故事。影子为一个叫做星期三的老头跑腿当差使, 而星期三其实是一个名叫奥丁的老神仙。奥丁是在9世纪的时候, 搭乘怀着早期维京探险者的挪威梦想来到北美的。他不过是美国的无数神祇之一。影子随后还遭遇了主神奥丁的兄弟、狡诈之神洛奇, 来自埃及的圣猫女神巴斯特, 斯拉夫的黑暗与死亡之神岑诺博格, 来自西非的骗术之神南西, 印度教的毁灭之神伽梨, 埃及神话中的冥界之神阿努比斯, 盎格鲁-撒克逊神话中的黎明之神伊斯特, 等等。

电影: 枪王之王

简介: 在IPSC实战射击赛中, 警员庄子维(吴彦祖 饰)打破了赛会纪录, 但新纪录很快被香港基金经理关友博(古天乐 饰)改写, 后者夺魁。赛后, 关友博巧遇蒙面歹徒打劫解款车, 解款员被歹徒打死。此时, 一个交通警(连凯 饰)路过, 被歹徒打伤。为救人, 关友博击毙了4名劫匪, 并报警。另一歹徒落荒而逃, 同时4亿美元债券遭抢。交警获救, 护理他的是关有博的女友(蔡卓妍 饰)。因非法持枪伤人, 关友博被拘。庄子维审问时发现对方超常冷静。面对检方控诉, 关友博沉着应对, 结果当庭获释。女上司(李冰冰 饰)接他回家, 但他对她的殷勤并不领情。关友博的账户存在资金周转问题, 他正想方设法拖延还款时间。庄子维在查案中毫无头绪, 于是请教老枪王(方中信 饰), 从对话中, 他似乎嗅到了关键性的线索……



电影: 国家宝藏: 夺宝秘笈 National Treasure: Book of Secrets

简介: 冒险家本·盖茨(尼古拉斯·凯奇饰)是次要挑战更艰难的探险寻宝之旅。美国第16任总统亚伯拉罕·林肯之死一直是美国历史上的一宗悬案, 据调查, 凶手约翰·沃克斯·布斯实际是受人唆使犯下惊天命案。盖茨参加一次公开演讲, 台下的一位叫米奇·威尔金森(艾德·哈里斯饰)的男子宣称, 盖茨的祖父与林肯之死有千丝万缕的联系。于是盖茨开始调查约翰·布斯相关的线索, 发现一本叫“国家密宗”的书, 是解答一切问题的关键。但国家密宗只有美国总统本人知道其所在, 为了探寻真相, 盖茨不惜铤而走险, 绑架总统。

请输入查询语句: (q 退出)

压缩

对于关键词，可以将其压缩为一整个字符串，倒排表只记录对应 **tag** 在字符串中的起始位置（这个也可以压缩为只记录差值），对于索引，可以只记录差值。由于文件本身就不大，压缩后文件大小只减小了 100 多 kb。

 inverted_index.txt	2023/10/30 22:39	文本文档	2,164 KB
 inverted_index_compress.txt	2023/10/30 22:39	文本文档	2,011 KB

搜索时需要先解压缩，得到完整的倒排索引表，再搜索。实际上，因为这里压缩方案很简单，可以不用解压缩，直接找到对应的 **tag**，但是每次查询都要重复找。一开始就解压缩可以显著提高效率。

测试结果如下：

压缩前：

```
请输入查询语句：(q 退出)
乡村 and 爱情 or 动作 and not 武侠 or 科幻
查询耗时：0.0s
查询结果：
书籍：真名实姓
简介：百年科幻史留下无数的经典，这本中从中精选六篇。选定这六篇的理由，不是因为它们出自名家，也不是因为它们获奖，而是因为它们对后来的科幻乃至现实产生了巨大的影响——回顾科幻史，没有人能够回避它们的存在。当然，还有另一个理由：它们从不同的侧面最大程度地呈现了科幻小说独有的魅力，让人读过便永难忘怀。《过去·现在·未来》，美国的纳特·沙克纳著，单伟健译。《美国制》，美国的杰·梯·麦金托什著，朱荣译。《霜与火》，美国的雷·布雷德伯里著，陈珏译。《沙王》，美国的乔治·马丁著，凌寒译。《大机器要停止运转了》，英国的E.M.福斯特著，何明译。《真名实姓》，美国的弗诺·文奇著，罗布顿译。
```

压缩后：

```
请输入查询语句：(q 退出)
乡村 and 爱情 or 动作 and not 武侠 or 科幻
查询耗时：0.0s
查询结果：
书籍：银河系漫游指南
简介：地球被毁灭了，因为要在它所在的地方修建一条超空间快速通道。主人公阿瑟·邓特活下来了，因为他有一位名叫福特·长官的朋友。这位朋友表面上是个找不着工作的演员，其实是个外星人，是名著《银河系漫游指南》派赴地球的研究员。两人开始了一场穿越银河的冒险，能够帮助他们的只有《银河系漫游指南》一书中所包括的无限智慧。旅途中，他们遇上了一批非常有趣的同伴：赞福德·毕博布鲁克斯：长着两个头、三条胳膊的银河大盗，他的另一个身份是银河帝国总统。崔莉恩：赞福德的同伙，除阿瑟·邓特之外惟一一个幸存下来的地球人。事实上，阿瑟从前认识崔莉恩，而且曾经试图勾搭人家，可惜没有成功。马文：天才机器人，疑心病极其重，极其沮丧，极其唠叨。这些人物结成一个小团队，他们将揭开一个骇人听闻的大秘密……

电影：太极旗飘扬 태극기 휘날리며
简介：李镇泰（张东健饰）是生活在汉城的一个修鞋匠，一家人的生计就从修补鞋子的费用维持。虽然物质并不富裕，这却是一个十分温馨完整的家庭。妻子李英顺（李恩珠饰）和母亲开一间面馆，平日还要照顾下面的弟妹，弟弟李镇锡（元彬饰）是家中所有的希望，如今正在高中读书，考上大学是他的梦想，更是哥哥镇泰的寄托所在。然而随着1950年朝鲜战争突然爆发，平静的生活很快被打乱。战火蔓延汉城，镇泰决定带着家人到大邱去生活，躲避战乱。但在中途，他们却被强行征入军队，押上了开往洛东江前线的列车。从此一家人四分五裂。而在战场上的镇泰只剩下一个愿望：不顾一切用生命保护弟弟，当听说只要获得国家勋章就可以让弟弟免役时，他便疯子般冲在战斗最前沿。然而残酷的战场加上弄人的命运，他们的生命轨迹发生了无可避免的改变。
```

检索效率差不多，都很快。但是显然的是，压缩后检索效率肯定变低了，因为解压需要时间。

检索结果

PB21111715 李宁

请输入查询语句: (q 退出)
 音乐 and 儿童 and 乡村
 查询耗时: 0.0s
 查询结果:
 电影: 放牛班的春天 Les choristes
 简介: 1949年的法国乡村, 音乐家克莱蒙特(热拉尔·朱尼奥 饰)到了一间外号叫“塘低”的男子寄宿学校当助理教师。学校里的学生大部分都是难缠的问题儿童, 体罚在这里司空见惯, 学校的校长(弗朗索瓦·贝莱昂 饰)只顾自己的前途, 残暴高压。性格沉静的克莱蒙特尝试用自己的方法改善这种状况, 他重新创作音乐作品, 组织合唱团, 决定用音乐的方法来打开学生们封闭的心灵。然而, 事情并不顺利, 克莱蒙特发现学生皮埃尔·莫安琦(让-巴蒂斯特·莫尼耶 饰)拥有非同一般的音乐天赋, 但是单亲家庭长大的他, 性格异常敏感孤僻, 怎样释放皮埃尔的音乐才能, 让克莱蒙特头痛不已; 同时, 他与皮埃尔母亲的感情也渐渐微妙起来。

放牛班的春天的视频和图片: (预告片4 | 图片589 | 添加)

放牛班的春天的获奖情况: (全部)

PB21111716 李乐祺

请输入查询语句: (q 退出)
 科幻 and 动画 and 冒险
 查询耗时: 0.0s
 查询结果:
 电影: 机器人总动员 WALL·E
 简介: 公元2805年, 人类文明高度发展, 却因污染和生活垃圾大量增加使得地球不再适于人类居住。地球人被迫乘坐飞船离开故乡, 进行一次漫长无边的宇宙之旅。临行前他们委托Buynlarge的公司对地球垃圾进行清理, 该公司开发了名为WALL·E(Waste Allocation Load Lifters - Earth 地球废品分装员)的机器人担当此重任。这些机器人按照程序日复一日、年复一年辛勤工作, 但随着时间的流逝和恶劣环境的侵蚀, WALL·E们接连损坏、停止运动。最后只有一个仍在进行这项似乎永无止境的工作。经历了漫长的岁月, 它开始拥有了自己的意识, 它喜欢将收集来的宝贝小心翼翼藏起, 喜欢收工后看看几百年前的歌舞片, 此外还有一只蟑螂朋友作伴。直到有一天, 一艘来自宇宙的飞船打破了它一成不变的生活……本片荣获2009年第81届奥斯卡最佳动画长片奖。

安德鲁·斯坦顿 本·贝尔特 艾丽莎·奈特 杰夫·格林 布莱恩·科兰斯顿

PB21111738 周子语

请输入查询语句: (q 退出)
 剧情 and 人生观
 查询耗时: 0.0s
 查询结果:
 电影: 黑客帝国动画版 The Animatrix
 简介: 《黑客帝国动画版》由9段以《黑客帝国》系列电影世界观为基础生发出的短片组成, 各篇的角度与风格各异, 制作班底汇聚了日本、美国、韩国三地的动画精英。《机器复兴》讲述了MATRIX的历史。机器人不堪人类恶劣对待, 终于反抗并将人类化为能源。《少年故事》讲述一名少年突然收到来自网络的召唤, 而大批黑衣人开始对他展开围捕。《虚拟程序》展示了一对男女在虚拟训练程序中的片断。《世界纪录》描写一位短跑运动员在接近突破人类极限时终被系统牢牢控制。《超越极限》叙述因为系统错误, 平静的街区中出现了一片超常区域。《侦探故事》描写一位侦探被步步引领着试图摆脱系统的桎梏。《矩阵化》讲述一只机器人被人类捕获并驯化的故事。《终极战役》记录了一艘飞船毁灭的经过。

西德尼·吕美特 亨利·方达 马丁·鲍尔萨姆 约翰·菲德勒 李·厄米

电影: 十二怒汉 12 Angry Men
 简介: 一个在贫民窟长大的18岁少年因为涉嫌杀害自己的父亲被告上法庭, 证人言之凿凿, 各方面的证据都对他极为不利。十二个不同职业的人组成了这个案件的陪审团, 他们要在休息室达成一致的意见, 裁定少年是否有罪, 如果罪名成立, 少年将会被判处死刑。十二个陪审团成员各有不同, 除了8号陪审员(Henry Fonda 饰)之外, 其他人对这个犯罪事实如此清晰的案子不屑一顾, 还没有开始讨论就认定了少年有罪。8号陪审员提出了自己的“合理疑点”, 耐心地说服其他的陪审员, 在这个过程中, 他们每个人不同的人生观也在冲突和较量……

Part2

本部分提交的文件目录如下:

part2

```
├── data
│   ├── movie_score.csv      # 原始评分数据
│   └── selected_movie_top_1200_data_tag.csv  # 原始 Tag 数据
├── res
│   ├── selected_users.csv   # 筛选后的用户评分数据
│   ├── selected_tags.csv    # 筛选后的 Tag 数据
│   └── tag_embedding_dict.pkl # 保存标签嵌入
└── src
    └── MF_rec.ipynb          # 基于矩阵分解的推荐系统
```

“

源码路径: `./part2/src`

准备工作

环境配置

首先下载 `cuda`，网址：<https://developer.nvidia.com/cuda-toolkit-archive>，下载安装即可。

再下载 `cuDNN`，网址：<https://developer.nvidia.com/rdp/cudnn-archive>，下载下来是个压缩包，直接解压缩，能看到如下三个文件夹（bin、include、lib），这三个文件夹拷贝到 `cuda` 的目录中。

最后下载 `pytorch`，命令：`pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121`

然后就可以跑样例代码了，结果如下。

自选一：使用 `Bert + MF` 的算法：（耗时 10min）

```
Epoch 16, Train loss: 1.854992614395317, Test loss:, 5.449914652725746, Average NDCG: 0.6824123077175995
87it [00:14, 5.92it/s]
Epoch 17, Train loss: 1.8180054193255546, Test loss:, 5.5486618787392805, Average NDCG: 0.687534359723334
87it [00:14, 6.13it/s]
Epoch 18, Train loss: 1.8072285542542907, Test loss:, 5.492937460713003, Average NDCG: 0.692802859516942
87it [00:14, 5.96it/s]
Epoch 19, Train loss: 1.7793422279686764, Test loss:, 5.62400745523387, Average NDCG: 0.698653452685045
```

自选二：使用 `GraphRec` 的算法：（运行太慢了，且效果和 `MF` 差不多，弃用）

```
[13] 38m 50.8s
... 87it [30:22, 20.95s/it]
Epoch 0, Loss: 1.6551116814558533, MSE loss:, 4.922742734010193, Average NDCG: 0.6756584621210483
7it [03:19, 28.45s/it]
```

“

如果 `tqdm` 报错，执行命令：`pip install ipywidgets` (但是进度条还是不显示，不过无所谓)；如果 `ConnectTimeoutError`，需要科学上网。

学习Pytorch

60分钟快速入门 PyTorch - 知乎 (zhihu.com)

改代码

数据预处理

处理 Tags

保留用户评价，加入到 `Tag` 中，需要去重。

相关代码如下：



```

1 # Tag 数据集
2 # 读tag_data取保存的 CSV 文件
3 tag_data = pd.read_csv('../data/selected_movie_top_1200_data_tag.csv')
4 # 引入用户评价数据
5 rating_data = pd.read_csv('../data/movie_score.csv')
6 # 删除 NaN 的行
7 rating_data.dropna(inplace=True)
8 # 将用户打的 Tag 加入到 tag_data['Tags'] 中
9 tag_data['Tags'] = tag_data['Movie'].map(rating_data.groupby('Movie')
    ['Tags'].apply(list).to_dict())
10 # 对于每一行 Tags, 将其转换为一整个字符串
11 tag_data['Tags'] = tag_data['Tags'].apply(lambda x: ','.join(x))
12 # 拆分为列表, 去重, 去除空字符串, '|', '...' 等无意义的 Tag
13 tag_data['Tags'] = tag_data['Tags'].apply(lambda x:
    list(set(x.split(','))))
14 tag_data['Tags'] = tag_data['Tags'].apply(lambda x: list(filter(lambda x:
    x not in ['', '|', '...'], x)))
15 # 保存为 CSV 文件
16 tag_data.to_csv('../res/selected_tags.csv', index=False)
17 print(tag_data)

```

处理 Users

有些用户的评分数据大部分是 0, 参考价值不大 (一般人也不会打 0 分(可能)), 考虑将评分为 0 的数据删掉。注意还要去掉评分数据太少的用户 (因为有的用户基本评的都是 0)

相关代码如下:

```

1 # User 数据集
2 # 读user_data取保存的 CSV 文件
3 user_data = pd.read_csv('../data/movie_score.csv')
4 # 去除评分为 0 的行
5 user_data = user_data[user_data['Rate'] > 0]
6 # 去除评价数据过少的用户
7 user_data = user_data.groupby('User').filter(lambda x: len(x) > 10)
8 # 去除不必要的列
9 user_data = user_data[['User', 'Movie', 'Rate']]
10 # 保存为 CSV 文件
11 user_data.to_csv('../res/selected_users.csv', index=False)
12 print(user_data)

```

运行结果:

```
63it [00:07, 8.59it/s]
Epoch 17, Train loss: 0.8031248412435017, Test loss:, 1.550738559828864, Average NDCG: 0.8687
63it [00:07, 8.99it/s]
Epoch 18, Train loss: 0.7799021034013658, Test loss:, 1.5414088112967355, Average NDCG: 0.876
63it [00:08, 7.79it/s]
Epoch 19, Train loss: 0.7688872435736278, Test loss:, 1.5296603479082622, Average NDCG: 0.872
```

可以看到，数据量变少之后运行速度显著提升，预测效果也变好了很多。由此可见数据预处理的重要性，原始数据可能有很多无用且干扰判断的信息。

模型优化

加入bias

考虑到有的用户比较苛刻，打分偏低，有的用户比较宽容，打分偏高。为每个用户和每个电影加入一些偏置元素 bu 和 bi ，代表了他们自带的与其他事物无关的属性，融入了这些元素，才能区别且正确地对待每一个用户和每一个物品，才能在预测中显得更加个性化。

相关代码如下：

```
1 # 定义模型，引入 Item User 偏置提高效果
2 class MF(nn.Module):
3     def __init__(self, num_users, num_movies, embedding_dim, init_std =
4         0.1):
5         super(MF, self).__init__()
6         self.user_embedding = nn.Embedding(num_users, embedding_dim)
7         self.movie_embedding = nn.Embedding(num_movies, embedding_dim)
8         self.user_bias = nn.Embedding(num_users, 1)
9         self.movie_bias = nn.Embedding(num_movies, 1)
10        nn.init.normal_(self.user_embedding.weight, std = init_std)
11        nn.init.normal_(self.movie_embedding.weight, std = init_std)
12        nn.init.normal_(self.user_bias.weight, std = init_std)
13        nn.init.normal_(self.movie_bias.weight, std = init_std)
14
15    def forward(self, user, movie):
16        user_embedding = self.user_embedding(user)
17        movie_embedding = self.movie_embedding(movie)
18        user_bias = self.user_bias(user)
19        movie_bias = self.movie_bias(movie)
20        dot = (user_embedding * movie_embedding).sum(1)
21        return dot + user_bias.squeeze() + movie_bias.squeeze()
```

运行结果如下：

```
Epoch 7, Train loss: 0.24820947718052638, Test loss:, 1.3445476501707048, Average NDCG: 0.90230483  
63it [00:15, 4.07it/s]  
Epoch 8, Train loss: 0.24655695850886997, Test loss:, 1.3227644326194885, Average NDCG: 0.90335186  
63it [00:15, 4.06it/s]  
Epoch 9, Train loss: 0.24549385266644613, Test loss:, 1.305357953858754, Average NDCG: 0.904102576
```

可以看到，优化效果还是比较明显的，平均 **NDCG** 已经达到了 0.9