web lab1

PB21051012 刘祥辉 PB20050990吕林峰 PB20020574 任呈祥

1.爬虫部分

一、爬取数据

采用的是 python 的 requests 库

```
    html_content = requests.get(url = url, headers = headers, proxies =
getip(),cookies = cookies)
    if html_content.status_code == 200:
        print("请求成功")
    else:
        print("请求失败，状态码：" + str(html_content.status_code))
    soup = BeautifulSoup(html_content.text, "html.parser")
```

针对反爬策略的应对策略：

1）设定user_agents：

每次爬取页面时随即返回一个user_agents

```
def get_ua():
    user_agents = [
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/118.0.0.0 Safari/537.36',
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/92.0.4515.107 Safari/537.36',
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:90.0) Gecko/20100101
Firefox/90.0',
        'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/91.0.4472.124 Safari/537.36',
    ]
    user_agent = random.choice(user_agents) # 随机抽取对象
    return user_agent
```

2)设定代理：

设定代理ip，每次爬取的时候随机设置代理ip:

```
iplist=[]
with open("verify_proxies.txt") as f:
    iplist = f.readlines()
def getip():
    proxy= iplist[random.randint(0,len(iplist)-1)]
    proxy = proxy.replace("\n","")
    proxies={
        'http':'http://'+str(proxy),
    }
    return proxies
```

可用代理存储在 `verify_proxies.txt` 文件中

3)使用cookie：

```
def get_cookie():
    cookie_sources = [
        'cookie1',
        'cookie2',
        'cookie3',
    ]
    cookie_source = random.choice(cookie_sources)
    return cookie_source
get_cook = get_cookie()
```

当使用代理和user_agents无法获取页面时再使用cookie访问即可。

经测试，使用这三条措施可以一次性完全爬取所需书籍，电影(部分给出待爬ID失效，直接忽略)。

二、解析数据

使用beautifulsoup库进行解析：

首次用python写的不太熟，我写的有点长不方便全放完，我就举个例子：

```
    # 提取导演信息
    directors = []
    director_tags = soup.find_all('span', {'class': 'attrs'})
    for director_tag in director_tags:
        directors.extend([a.get_text() for a in director_tag.find_all('a',
{'rel': 'v:directedBy'})])
    # 提取编剧信息
    writers = []
    writer_span = soup.find('span', class_='pl', string='编剧')
    if(writer_span):
        writer_span = writer_span.find_next('span', class_='attrs')
        for writer in writer_span:
            writers.append(writer.get_text())
    else:
        pass
```

对于书本解析有些不同，主要是书本html结构不太固定，尤其是作者部分

```
    #作者
    author = []
    author_span = soup.select("#info > span:nth-child(1)")
    others = soup.select("#info > a:nth-child(2)")
    if author_span:
        authors = author_span[0].find_all('a')
        if(authors):
            for a in authors:
                author.extend(a.get_text())
        elif others:
            author.extend(others[0].get_text().replace('\u3000',
'').replace('\n', '').replace(' ',''))
        else:
            pass
    else:
        pass
```

三、结果展示

结果存放在：`web_lab1/web_lab2_reptile`中

包含`movie.csv`和`book.csv`

2.索引表

结果保存在`web_lab1/web_lab2_chart/splitwordtest.ipynb`

数据处理生成tag

本阶段split.py对前面爬虫得到的信息进行处理,得到对应电影和书籍的tag.

数据预处理和分词

对电影简介句子进行处理,去掉标点符号和空格,将文本分为句子的list.然后使用jieba库对每个句子进行分词.分词后得到的词语集合为origin.将origin中的词与stoplist(停用词表)进行比较,去除在停用词表中的单词,在与alltag(豆瓣电影全部标签集合)进行比较,如果是出现在alltag,就将这个词加入对应电影的tag.

```
def intro_process(text):
    global stop_words
    global alltag
    text = re.sub(r'[^\w\s]', ' ', text)
    # 分割句子
    sentences = re.split(r'[.!?]', text)
    # 去除多余的空白
    sentences = [s.strip() for s in sentences if s.strip()]
    originaltag=[]
    origin=[]
    for sentence in sentences:
        originaltag=jieba.lcut(sentence)
        origin.append(originaltag)
    if len(origin)>0:
        origin=origin[0]
    kong=[' ']
    afterdelte=[]
    for word in origin:
        if word not in stop_words and word not in kong and word in alltag:
            afterdelte.append(word)
    return afterdelte
```

对电影名字,演员的前三位,类型,地区进行数据处理后也加入对应电影的tag中.

```
dicts=[]
for data in neededinfo:
    dic={}
    gettags=intro_process(data[13])
    tags=tag_process(data[5])
    for tag in tags:
        if tag not in gettags:
            gettags.append(tag)
    gettags.append(data[1].replace(" ",","))
    gettags.append(data[2])
    actors=actor_process(data[4])
    gettags.append(data[6].replace('/',','))
```

```
    if len(actors)>0:
        actors=actors[0].split()
    i=0
    j=len(actors)
    while i<3 and j>0:
        gettags.append(actors[i])
        i+=1
        j-=1
    dic['id']=data[0]
    dic['name']=data[1]
    string=''
    for tag in gettags:
        string=string+tag+","
    string=string.replace(' ','')
    dic['tag']=string
    dicts.append(dic)
```

分词库对比

splitwordtest.ipynb选用jieba库和snownlp库对电影简介处理,通过得到的tag总数和花费时间对两个库的性能进行比较.处理过程和前面相似,但加入snow_process(text)选用snowNLP来分词

```
jiebatag=[]
start=time.time()
for data in neededinfo:
    jiebatag+=intro_process(data[13])
end=time.time()
print(len(jiebatag))
print(end-start)
```

输出结果

```
206    1.6928455829620361
```

表明jieba库得到206个tag,用时1.69s

```
snowtag=[]
start=time.time()
for data in neededinfo:
    snowtag+=snow_process(data[13])
end=time.time()
print(len(snowtag))
print(end-start)
```

输出结果

```
238       64.91031289100647
```

表明snowNLP得到238个tag,用时64.91s可见jieba花费时间少,精度较低,snownlp花费时间远大于jieba,精度更高

构建倒排表

query.py文件实现了构建倒排表和布尔查询两个功能

代码的主要实现思路是先读入书籍、电影ID和关键词集合，将他们存入同一split_word中

split_word的每个元素是列表，列表的第一个元素是ID，后边的全是该ID对应的关键词，由于电影和书籍ID没有重复的，可以将他们存到一起

```
# 进行根据ID进行排序，对ID排序会方便后续的布尔运算以及二分查找的实现
split_word.sort(key=lambda x: x[0])

# 把ID对应的每个关键词拆开，构成[关键词，ID]的列表存入res_list中
res_list = []
for item in split_word:
    for tag in item[1:]:
        res_list.append([tag, [item[0]]])
# 再根据关键词进行排序，python会自行比较字符串的大小并进行排序
res_list.sort(key=lambda x: x[0])

###之后对关键词相同的ID合并，得到倒排表。
start = 0  # 开始遍历位置
pos = 0
while pos != len(res_list) - 1:
    if res_list[pos + 1][0] == res_list[pos][0]:
        if res_list[pos][1][-1] != res_list[pos + 1][1][0]:
            res_list[pos][1].append(res_list[pos + 1][1][0])
        del res_list[pos + 1]
    else:
        pos += 1

# 把词典和倒排表分开存储
# 由于之前已经排过序，顺序默认是正序的
res_list_key = [key[0] for key in res_list]  # 存储词典
res_list_value = [key[1] for key in res_list]  # 存储倒排表
```

倒排表优化

采用按块存储进行优化

讲所有的词项存入到一个字符串total_string中

由于关键词开头可能是数字，与存储关键词长度的数字相混，解决方法是长度后边加一个空格

```
# 压缩词项列表
total_string = ""
for i, item in enumerate(res_list_key):
    pos = len(total_string)
    # 考虑到首位为数字的情况，解决方法是加一个空格
    if item[0].isdigit():
        total_string += (str(len(item) + 1) + " " + item)
    else:
        total_string += (str(len(item)) + item)
    res_list_key[i] = pos

# 取块大小为4，将下标为4的倍数的词项位置存入res_list_key中
k = 4
temp = []
temp = [x for i, x in enumerate(res_list_key) if i % 4 == 0]
res_list_key = temp
```

二分查找

对关键词的查找使用find函数进行二分查找

不使用倒排表进行优化：

```
# 朴素的二分查找，找到位置后返回该位置在res_list_value中的值，即对应的ID
def find(a):
    right = key_num
    left = 0
    while left <= right:
        mid = (right + left) // 2
        if res_list_key[mid] == a:
            return res_list_value[mid]
        elif res_list_key[mid] < a:
            left = mid + 1
        else:
            right = mid - 1
    return []
```

对倒排表进行优化：

新增了两个函数

```
#该函数根据下标pos去total_string找，得到对应的词项长度word_length和连带长度的总长度
total_length
def get_length(pos):
    word_length = ""
    total_length = 0
    while pos <= len(total_string) and total_string[pos].isdigit():
        word_length = word_length + total_string[pos]
        pos += 1
    if word_length == "":
        return [0, 0]
    total_length = len(word_length) + int(word_length)
    word_length = int(word_length)
    return [word_length, total_length]


# 由于块的大小k=4，这个函数的作用是求第pos个块的第k个关键词
def get_list_value(pos, K):
    length = [0, 0]
    while K >= 0:
        pos += length[1]
        length = get_length(pos)
        K -= 1
    while total_string[pos].isdigit():
        pos += 1
    if total_string[pos] == " ":
        pos += 1
        length[0] -= 1
    value = ""
    while length[0] > 0:
        value += total_string[pos]
        pos += 1
        length[0] -= 1
    return value
```

```
# 二分查找也要进行修改，判断一个关键字是否在一个块中，需要与该块开头与下一个块开头字符进行比较。由于最后一个块没有下一块，要先讨论
def find(a):
    right = int(len(res_list_key) - 1)
    left = 0
    # 先讨论a在最后一个的情况
    if a > get_list_value(res_list_key[right], 0):
        if a == get_list_value(res_list_key[right], 0):
            return res_list_value[right * 4]
        if a == get_list_value(res_list_key[right], 1):
            return res_list_value[right * 4 + 1]
        if a == get_list_value(res_list_key[right], 2):
            return res_list_value[right * 4 + 2]
        if a == get_list_value(res_list_key[right], 3):
            return res_list_value[right * 4 + 3]
    # a不在最后一块，进行二分查找，与块中每一个元素进行比较
    while left <= right:
        mid = (right + left) // 2
        if get_list_value(res_list_key[mid], 0) <= a <= get_list_value(res_list_key[mid], 3):
            if a == get_list_value(res_list_key[mid], 0):
                return res_list_value[mid * 4]
            if a == get_list_value(res_list_key[mid], 1):
                return res_list_value[mid * 4 + 1]
            if a == get_list_value(res_list_key[mid], 2):
                return res_list_value[mid * 4 + 2]
            if a == get_list_value(res_list_key[mid], 3):
                return res_list_value[mid * 4 + 3]
        elif get_list_value(res_list_key[mid], 0) < a:
            left = mid + 1
        else:
            right = mid - 1
    return []
```

布尔查找

由于可能会有括号，先找查找对象query中的右括号")"，

具体做法是首先创建一个栈stack，把query中的元素从左到右入栈，直至找到")"，再把栈中元素出栈，直至"("出栈

这样得到一个布尔查询的子模块，对其进行计算，再递归去括号即可

在实现过程中布尔查询关键词，如and连接的元素 如 A and B，这里的的A和B既可以是要查询的关键词，也可以是列表，实际计算先把关键词转化为他们对应的ID的列表，再对两个列表进行布尔运算。

```
        # 由于运算优先级not > and > or，根据这个顺序进行计算
        while 'not' in cal_list:
            pos = cal_list.index('not')
            cal_list[pos] = cacu_not(cal_list[pos + 1])
            del cal_list[pos + 1]
        while 'and' in cal_list:
            pos = cal_list.index('and')
            cal_list[pos - 1] = cacu_and(cal_list[pos - 1], cal_list[pos + 1])
            del cal_list[pos]
            del cal_list[pos]
        while 'or' in cal_list:
```

```
            pos = cal_list.index('or')
            cal_list[pos - 1] = cacu_or(cal_list[pos - 1], cal_list[pos + 1])
            del cal_list[pos]
            del cal_list[pos]
        stack.append(cal_list[0])
```

这里使用的三个函数 cacu_not， cacu_and， cacu_or，分别实现了对not，and，or的计算

```
# 如过a或b不为列表，先转化为对应的列表，再进行计算
# and计算为两个队列从前往后遍历，找到相等元素插入result
def cacu_and(a, b):
    result = []
    if type(a) != list:
        a = find(a)
    if type(b) != list:
        b = find(b)
    i = 0
    i_max = len(a)
    j = 0
    j_max = len(b)
    result = []
    while i != i_max and j != j_max:
        if a[i] == b[j]:
            result.append(a[i])
            i += 1
            j += 1
        elif a[i] < b[j]:
            i += 1
        elif a[i] > b[j]:
            j += 1
    return result

# or计算为两个队列从前往后遍历，把元素插入result，如果任一队列到末尾，把另一队列剩余的元素全部插入result
def cacu_or(a, b):
    result = []
    if type(a) != list:
        a = find(a)
    if type(b) != list:
        b = find(b)
    i = 0
    i_max = len(a)
    j = 0
    j_max = len(b)
    result = []
    while 1:
        if a[i] == b[j]:
            result.append(a[i])
            i += 1
            j += 1
        elif a[i] < b[j]:
            result.append(a[i])
            i += 1
        elif a[i] > b[j]:
            result.append(b[j])
            j += 1
        if i == i_max:
```

```
            while j != j_max:
                result.append(b[j])
                j += 1
            return result
        elif j == j_max:
            while i != i_max:
                result.append(a[i])
                i += 1
            return result

# not计算ID_list存储的是所有的ID，计算时用key存储所有a中的ID的下标，再从后往前删除（为了不破坏下标）
def cacu_not(a):
    result = []
    if type(a) != list:
        a = find(a)
    result = ID_list[:]
    del_pos = []
    i = 0
    max_i = len(a)
    for key, item in enumerate(result):
        if item == a[i]:
            del_pos.append(key)
            i = i + 1
            if i == max_i:
                break
    for item in del_pos[::-1]:
        del result[item]
    return result
```

优化比较

```
print("词典空间大小为:{}".format(sys.getsizeof(res_list_key)))
print("词典串大小为:{}".format(sys.getsizeof(total_string)))
```

用getsizeof函数去求所占用空间大小

结果：

```
未优化
词典空间大小为:59736
优化后
词典空间大小为:14360
词典串大小为:81106
```

可以看到未优化所占用的空间其实是更少的

经过分析后，原因可能有以下两点

1：python列表本身就是变长存储的，未优化时，res_list_key是字符的列表，每个元素的大小根据所存储字符的长短会有所变化

将词典视作单一字符串，可以节省定长存储导致的额外空间，但python是变长存储的，效果不是特别好。

2：python本身采用unicode编码，实验中查询的关键词也包含中文英文韩文等，会占用额外的空间，定长存储导致的额外空间会进一步减少，压缩效果会更糟。加上需要存储字符本身的长度，会占用更多空间。

按块存储相比于单纯的将词典视作单一字符串以此来压缩表项的话，无疑占用空间是减少的。

查找时间比较：

进行比较

查询对象为

`query = "自由 and 自由 and 自由 and 自由 and 自由 and 自由 and " * 1000 + "自由"`

这么做是为了减少布尔运算使用时间，使得主要时间集中在布尔查找上，体现差异

未优化：

`查询成功，共找到25个结果,其中25部电影,0本书`
`查询所用时间：0.044880 秒`

优化后

`查询成功，共找到25个结果,其中25部电影,0本书`
`查询所用时间：0.329990 秒`

可以看到所消耗时间增加，和理论一致

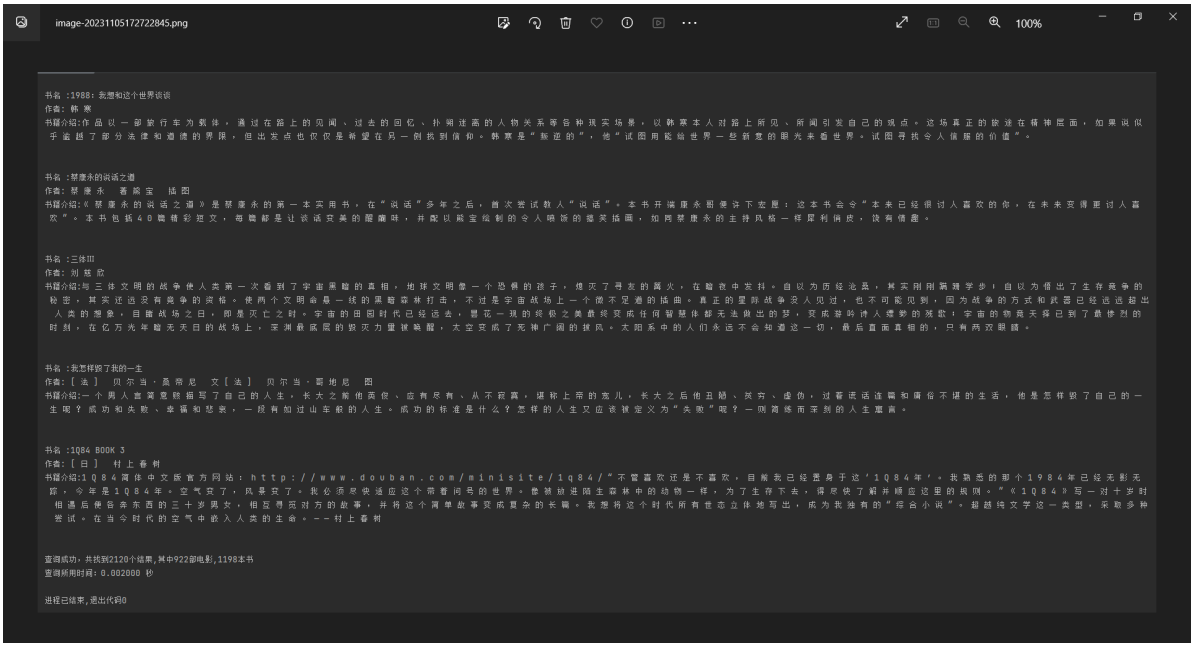
结果展示

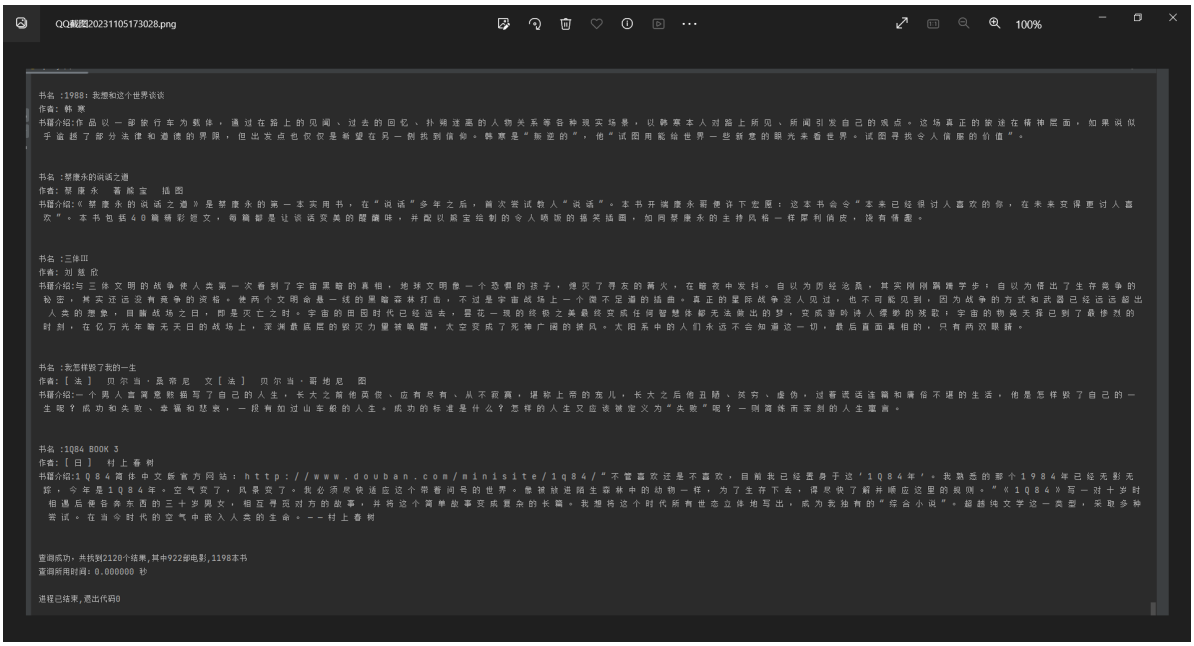
查询内容为

`query = "(not 剧情) or (not 爱情)"`
`query = "not (剧情 and 爱情)"`

这是简单的摩根定律，二者输出内容应该相同

输出结果分别为





二者运行时间不同，结果完全相同

3.推荐部分

相关文件保存在`web_lab1/web_lab2_recommend`

一、Funk-SVD 算法

1. 本此实验，使用了Funk-SVD算法分接矩阵来预测 user 对 item 的评分。
2. **模型说明**：
   $M_{m*n} = P^T_{m*k} Q_{k*n}$ 其中M矩阵为预测用户对每本书的得分矩阵，P矩阵为用户和书籍标签的关系矩阵，Q矩阵为书籍和书标签的关系矩阵
3. **原理说明**
   对于用户矩阵，每个用户对不同类型的书籍偏好不同，某个用户可能偏好历史、文化、政治类，而另一个用户可能偏好漫画类，所以可以根据这种关系构建两个矩阵，矩阵相乘结果即是用户对书籍的评分。
4. **矩阵更新*
   采用反向传播法更新着这两个矩阵，经过求导得出更新公式:

$$p_i = p_i + \alpha[(m_{ij} - q_j^T p_i)q_j - \lambda p_i]$$

$$q_i = q_i + \alpha[(m_{ij} - q_j^T p_i)p_j - \lambda q_i]$$

防止过拟合，还可以引入正则系数

二、实现过程

1.书籍标签的聚合

本次实验提供每本书的标签，但是过于杂乱，我们将书籍的标签聚合到21个大的标签里，并用这些标签对书籍矩阵进行初始化。聚合书记标签的目的是为了初始化用户矩阵和书记矩阵，以此得到更好的效果。

```
for index, content in data.iterrows():
```

```
        category_to_tags['Tags'] = []
        category_to_tags['Book'] = content['Book']
        lists = content['Tags'].split(',')
        tags = []
        for a in lists:
            res = re.findall(r"'([^']+)'", a)
            if(res):
                result_string = res[0]
                tags.append(result_string)
        for category, tag_lists in category_mapping.items():
            for tag in tags:
                for tag_list in tag_lists:
                    if tag_list in tag:
                        if(category) not in category_to_tags['Tags']:
                            category_to_tags['Tags'].append(category)
                        else:
                            pass
                    else:
                        pass
        tag_group  = tag_group._append(category_to_tags,ignore_index=True)
        category_to_tags.clear()
```

最终效果为：

```
Book,Tags
1046265,"['中国', '欧美', '日本', '童话', '小说', '漫画', '生活', '人文社科', '传记', '现代', '青春', '散文诗歌', '历史', '爱情']"
1084336,"['中国', '欧美', '日本', '童话', '小说', '漫画', '生活', '经管', '人文社科', '传记', '古典', '现代', '青春', '散文诗歌', '科幻玄幻']"
1008145,"['中国', '日本', '小说', '漫画', '生活', '人文社科', '传记', '古典', '现代', '散文诗歌', '历史']"
1017143,"['中国', '欧美', '日本', '童话', '小说', '生活', '人文社科', '现代', '青春', '悬疑', '散文诗歌', '历史']"
1040771,"['欧美', '日本', '小说', '漫画', '生活', '科技', '人文社科', '传记', '现代', '青春', '悬疑', '散文诗歌', '武侠', '科幻玄幻', '历史', '爱情']"
1016300,"['中国', '欧美', '小说', '漫画', '经管', '人文社科', '传记', '现代', '青春', '散文诗歌', '网文', '爱情']"
1529893,"['中国', '日本', '小说', '漫画', '生活', '科技', '人文社科', '传记', '现代', '青春', '散文诗歌', '历史']"
1082154,"['中国', '小说', '人文社科', '传记', '现代', '青春', '散文诗歌', '历史']"
1026425,"['中国', '欧美', '小说', '人文社科', '传记', '现代', '青春', '散文诗歌', '网文']"
1066462,"['中国', '日本', '童话', '小说', '漫画', '生活', '人文社科', '传记', '现代', '青春', '散文诗歌']"
1770782,"['中国', '欧美', '日本', '小说', '生活', '人文社科', '传记', '现代', '青春', '历史']"
1400705,"['中国', '欧美', '日本', '小说', '生活', '科技', '人文社科', '传记', '现代', '青春', '散文诗歌', '历史']"
1024217,"['中国', '小说', '科技', '人文社科', '传记', '现代', '青春', '散文诗歌', '武侠', '历史']"
2256039,"['中国', '小说', '漫画', '生活', '经管', '人文社科', '传记', '现代', '青春', '历史']"
4886245,"['中国', '小说', '人文社科', '传记', '现代', '青春', '散文诗歌', '科幻玄幻', '网文', '历史']"
1007305,"['中国', '童话', '小说', '漫画', '生活', '人文社科', '传记', '古典', '青春', '散文诗歌', '历史']"
1401425,"['中国', '小说', '人文社科', '现代', '青春', '历史']"
1065970,"['中国', '小说', '生活', '人文社科', '传记', '现代', '青春', '散文诗歌']"
1090043,"['中国', '童话', '小说', '生活', '人文社科', '现代', '青春', '散文诗歌']"
1029159,"['中国', '小说', '生活', '人文社科', '传记', '现代', '青春', '散文诗歌']"
1082518,"['中国', '欧美', '童话', '小说', '人文社科', '传记', '古典', '现代', '青春', '散文诗歌']"
1030052,"['中国', '小说', '生活', '人文社科', '传记', '现代', '青春', '散文诗歌', '历史']"
1141406,"['中国', '欧美', '日本', '童话', '小说', '漫画', '生活', '人文社科', '传记', '古典', '青春']"
2256438,"['中国', '日本', '童话', '小说', '生活', '人文社科', '传记', '现代', '青春', '散文诗歌']"
2250587,"['中国', '日本', '小说', '漫画', '生活', '人文社科', '现代', '青春', '散文诗歌']"
1465324,"['欧美', '日本', '小说', '生活', '经管', '人文社科', '现代', '青春', '散文诗歌']"
1059419,"['欧美', '日本', '小说', '生活', '经管', '人文社科', '传记', '现代', '青春', '散文诗歌', '科幻玄幻']"
1049219,"['中国', '小说', '漫画', '人文社科', '现代', '青春', '散文诗歌', '武侠', '网文', '爱情']"
1061118,"['中国', '小说', '生活', '人文社科', '现代', '历史']"
1016523,"['中国', '小说', '生活', '科技', '人文社科', '传记', '现代', '青春', '散文诗歌']"
```

根据此标签，如果用户对某本书打了分，便可以在用户矩阵上对相应的标签进行初始化。

```
def user_matrix_init(user_num,type_num,trainset,book_tag,tag_mapping):
    user_matrix = np.random.normal(0.5, .1, (user_num, type_num))
    user_matrix = np.clip(user_matrix, 0, 1)
    #初始化用户矩阵
    number = 1.0  #待调参数
    for row in range(1, user_num + 1):
        filtered_df = trainset[trainset['User'] == row] #该用户所有的打分记录
        for index,content in filtered_df.iterrows():
            book_id = content['Book']
            tags = book_tag.iloc[book_id-1]
            tag_list = eval(tags['Tags'])  #访问书具有的tag
            for tag in tag_list:
                value_found = tag_mapping.get(tag)
                pairs = list(data[value_found])
```

```
                pairs[0] += content['Rate'] * number
                pairs[1] += 1
                data[value_found] = tuple(pairs)
        for line in range(type_num):    #数值填充
            pairs = list(data[line+1])
            if pairs[1]!= 0:
                user_matrix[row-1,line] = pairs[0]/pairs[1]
            else:
                pass
        for key in data:
            data[key] = (0, 0)
    return user_matrix
```

2.模型训练

```
class MatrixFactorization():
    '''基于矩阵分解的推荐.'''

    def __init__(self, learning_rate, n_epochs, lmd, user_matrix, book_matrix):
 #转置后的矩阵

        self.lr = learning_rate  # 梯度下降法的学习率
        self.n_epochs = n_epochs  # 梯度下降法的迭代次数
        self.lmd = lmd # 防止过拟合的正则化的强度
        self.user_matrix = user_matrix
        self.book_matrix = book_matrix

    def fit(self, trainset):
        '''通过梯度下降法训练, 得到所有 u_i 和 p_j 的值'''
        self.trainset = trainset
        print('Fitting data with SGD...')
        for _ in range(self.n_epochs):
            print(f"epoches:{_}")
            for index,content in self.trainset.iterrows():
                i = content['User']
                j = content['Book']
                r_ij = content['Rate']
                err = r_ij -np.dot(self.user_matrix[i-1],self.book_matrix[j-1])
                self.user_matrix[i-1] -= -self.lr * err * self.book_matrix[j-1]
+ self.lr * self.lmd * self.user_matrix[i-1]
                self.book_matrix[j-1] -= -self.lr * err * self.user_matrix[i-1]
+ self.lr * self.lmd * self.book_matrix[j-1]
        return self.user_matrix,self.book_matrix
```

基于反向传播法对矩阵进行更新，并将更新后的矩阵返回用于评测。

3.评测

本次主要采用NDCG进行评测，辅助采用MAE评测

1）MAE方法(平均绝对误差)评测：

```
    for index, content in testset.iterrows():
        i = content['User']
        j = content['Book']
        Rate = content['Rate']
        prediect = np.dot(user_matrix[i-1], book_matrix[j-1])
        predictions.append(prediect)  # 将预测值添加到列表中
        total_err += abs(prediect - Rate)
    # 将预测值添加到 DataFrame
    testset['prediction'] = predictions
    # 计算 MAE
    MAE = total_err / num_rows
```

输出结果为`MAE = 1.047308224537279`

2)NDCG方法：

- 对于测试集中的每个 user, 我们先预测它在测试集中的 item 的评分，将其按照评分排序，得到 predict，用它的真实得分和预测的排序结果算出 DCG
- 再将该 user 在测试集中的 item 按照评分由高到低排序，用它的真实评分和真实排序计算出 iDCG
- 两者相除，即可算出该 item 的 ndcg
- 对于所有用户的 ndcg 求平均，得到NDCG

```
def evaluate(eval):
    #处理0分情况
    small_value = 0.00001  # 你可以根据需要选择一个很小的数
    eval['Rate'] = eval['Rate'].replace(0, small_value)
    ave_IDCG = 0
    user_number = 0
    for user, user_data in eval.groupby('User'):
        # 根据Rate列对用户的数据进行排序
        IDCG = 0
        DCG = 0
        #书名和评分的字典
        book_rate_dic = user_data.set_index('Book')['Rate'].to_dict()
        #真实排名
        ideal_sorted_data = user_data.sort_values(by='Rate', ascending=False)
        real_book_preferences = ideal_sorted_data['Book'].tolist()
        for i in range(len(book_rate_dic)):
            IDCG += book_rate_dic.get(real_book_preferences[i])/math.log(i+2, 2)
        #预测排名
        eval_sorted_data = user_data.sort_values(by='prediction',
ascending=False)
        eval_book_preferences = eval_sorted_data['Book'].tolist()
        for i in range(len(book_rate_dic)):
            DCG += book_rate_dic.get(eval_book_preferences[i])/math.log(i+2, 2)
        ave_IDCG += DCG/IDCG
        user_number += 1
    ave_IDCG = ave_IDCG/user_number
    return ave_IDCG
```

对于排序结果：我没有显示的输出出来。我是遍历每个用户的时候建立字典，进行排序的。

```
        #书名和评分的字典
        book_rate_dic = user_data.set_index('Book')['Rate'].to_dict()
        #真实排名
        ideal_sorted_data = user_data.sort_values(by='Rate', ascending=False)
        real_book_preferences = ideal_sorted_data['Book'].tolist()
        for i in range(len(book_rate_dic)):
            IDCG += book_rate_dic.get(real_book_preferences[i])/math.log(i+2, 2)
        #预测排名
        eval_sorted_data = user_data.sort_values(by='prediction',
ascending=False)
        eval_book_preferences = eval_sorted_data['Book'].tolist()
        for i in range(len(book_rate_dic)):
            DCG += book_rate_dic.get(eval_book_preferences[i])/math.log(i+2, 2)
        ave_IDCG += DCG/IDCG
```

在这个过程中算出来每个用户的DCG/IDCG的

三、结果分析

经过调参，选定参数为：

`learning_rate=.005, n_epochs=15, lmd = 0.2, number = 1.0`

结果为`NDCG = 0.862784079436799`

`MAE = 1.047308224537279`

测试集预测数据存到 `stage3/Data/predict.csv` 中。部分结果如下图所示：

```
5,959,3,2008-08-22T22:45:17+08:00,"九州,江南,缥缈录",4.823385182746016
5,661,4,2008-08-22T22:23:49+08:00,"可爱淘,狼的诱惑",3.1564125637454703
5,59,5,2008-08-22T21:48:39+08:00,"悟空传,今何在,网络小说,中国,西游记",4.793956644449522
5,11,5,2008-08-22T21:32:41+08:00,,5.097127087590592
5,227,4,2009-03-16T23:06:39+08:00,,4.191850087146673
5,1074,3,2009-12-06T20:33:53+08:00,,3.326478569248037
5,291,3,2008-08-22T22:21:24+08:00,"明晓溪,会有天使替我爱你",3.1704944109146345
5,586,3,2008-08-22T22:04:02+08:00,"饶雪漫,左耳",3.6286034384837613
5,1097,4,2008-08-22T22:48:54+08:00,"今何在,九州,海上牧云记",3.83550364722111
5,173,4,2008-12-04T00:00:43+08:00,,3.824493311196651
5,245,4,2008-08-22T22:37:09+08:00,"欧·亨利,麦琪的礼物",4.647383161923155
5,627,3,2008-08-22T22:05:40+08:00,"一光年的距离有多远,曾炜",2.6998276295184427
5,267,4,2008-08-22T22:03:09+08:00,,3.2637572575078893
5,277,4,2010-12-12T11:41:01+08:00,,5.087237787734475
5,234,5,2009-02-23T18:58:43+08:00,,4.169381648635175
5,39,5,2008-08-22T21:41:24+08:00,,4.882463622979623
5,703,4,2010-12-21T17:16:10+08:00,,4.055454584824927
5,670,5,2008-08-22T22:49:11+08:00,,3.8326238448350356
5,658,5,2008-08-22T21:42:18+08:00,,5.060396376407198
5,232,3,2009-10-27T07:55:39+08:00,,3.9314694244226174
5,568,4,2009-12-26T17:10:16+08:00,,4.155341061087643
5,640,4,2010-01-29T18:47:24+08:00,,4.160550917678474
```

大部分 item 的结果接近。

同时我们进行了对比实验，当用户矩阵和书籍矩阵为随机化生成时，得到的结果为

`NDCG = 0.8499662760436361`

`MAE = 1.06545886484845`

虽然有提升，但是不高，分析下原因：

聚合的tag并不是清晰分明，像中国、外国这类标签出现在了绝大部分书籍上，还有一部分标签如"科技"出现频率很低，如果能选择标签恰当，将这1000多本书接近平均的划分到各个标签上，对于提高准确度一定是有很大帮助的。

对于NDCG评估：

当用户评分全为0的时候(比如book_score.csv中第一个用户)，不管怎么排结果均是1，

类似，当用户只有1条记录时，不管怎么排，也全是1(影响小一些，我看了下用户评分记录还是挺稠密的)