Web实验报告

组长：方驰正PB21000163

组员：来泽远PB21000164、周匡翔PB21000265

一、实验目的

豆瓣（www.douban.com）是一个中国知名的社区网站，以书影音起家，用户可以在豆瓣上查看感兴趣的电影、书籍、音乐等内容，还可以关注自己感兴趣的豆友。

本实验的目的为爬取指定的电影、书籍的主页，并解析其基本信息，然后结合给定的标签信息，实现电影和书籍的检索并评估其效果；在此基础上，结合用户的评价信息及用户间社交关系，进行个性化电影、书籍推荐。

二、实验环境

- 操作系统：Windows 11
- 开发环境：jupyter notebook
- 软件平台：visual studio code

三、实验内容

1.爬虫

爬取数据：

下面以电影为例，说明具体爬取过程

```
for line in fread.readlines():
    time.sleep(0.1) # 反爬

    id = line.strip('\n')

    #防止重爬 && 跳过下架电影的id
    if id in id_map or id in skip_set:
        continue

    print(id)

    url = 'https://movie.douban.com/subject/' + id + '/'
    try:
        content = get_content(url)
        id_map[id] = content
    except:
        with open(error_path, "a+") as ferror:
            ferror.write(id + "\n")
```

我们使用 `request` 库进行爬虫，通过维护 `id_map` 与 `skip_set` 两个集合，实现了断点续爬与跳过下架页面的功能。其中，id_map用于存储爬取到的电影信息，skip_set用于存储下架电影的id，避免重复爬取。 `get_content` 函数用于获取电影主页的html内容。

解析数据：

以获取info信息为例，具体解析过程如下：

```python
def get_info(soup):
    info = soup.find('div', id='info')
    info = info.get_text().strip('\n').split('\n')
    info = [i.split(': ') for i in info]
    info_dict = {}
    for i in info:
        #check for i[1] exsit
        if (len(i) == 1):
            continue
        info_dict[i[0]] = i[1].split(' / ')
        if (len(info_dict[i[0]]) == 1):
            info_dict[i[0]] = info_dict[i[0]][0]
    return info_dict
```

我们使用 `BeautifulSoup` 库进行解析，通过 `find` 函数找到html中对应的标签，然后使用 `get_text` 函数获取标签内的文本内容，再通过 `strip` 函数去除多余的换行符，最后使用 `split` 函数进行分割，得到一个列表。我们将列表中的每一项再次使用 `split` 函数进行分割，得到一个二维列表。最后，我们将二维列表转化为字典，得到电影的info信息。

反爬策略：

我们通过给 `request` 函数添加 `headers` 参数，模拟浏览器访问，避免被反爬。同时，为了避免同一ip在一定时间内访问次数过多，我们在每次爬取后都使用 `time` 库的 `sleep` 函数，使爬虫休眠0.1s。

实验结果：

最终爬取的数据如 `data/Book_info.json` 和 `data/Movie_info.json` 所示。这里仅展示部分

```json
{
    "1046265": {
        "name": "挪威的森林",
        "info": {
            "原作名": "ノルウェイの森",
            "出版年": "2001-2",
            "页数": "350",
            "定价": "18.80元",
            "装帧": "平装",
            "ISBN": "9787532725694"
        },
        "rating": {
            "评分": " 8.1 ",
            "评分人数": "346906",
            "1星": "34.1%",
            "2星": "34.1%",
            "3星": "34.1%",
            "4星": "34.1%",
            "5星": "34.1%"
        },
        "intro": "这是一部动人心弦的、平缓舒雅的、略带感伤的恋爱小说。..."
    },
    ...
}
```

2.检索

预处理

这里，我们采用结巴分词库，同时加入简介中的类型字段，并合并了近义词，删除了停用词，以帮助接下来的查询。过程如下所示：

```
for id in content:
    if self.type == "Movie":# 书籍没有类型
        Type = content[id]['info']['类型']
    intro = content[id]['intro']
    seg_list = jieba.cut(intro)

    # 合并同义词 && 去除停用词

    seg_set = set()
    for seg in seg_list:
        if seg in synonym:
            seg_set.add(synonym[seg])
        elif seg not in stop_word:
            seg_set.add(seg)

    # merge seg_set and type
    if self.type == "Movie":
        for t in Type:
            seg_set.add(t)

    if id in tag_map:
        for tag in tag_map[id]:
            seg_set.add(tag)

    # output[id] = "/".join(seg_set)
    content[id]['tags'] = '/'.join(seg_set)
```

这里，我们将使用 `jieba` 库与 `thuac` 库分词的结果进行比较，最终选择使用 `jieba` 库。以对《肖申克的救赎》电影的分词结果为例， `jieba` 库分词结果如下：

```
"tags": "第一/使用/消失/一把/虚伪/金球奖/总帐/合法/TimRobbins/妻/囚禁/知识/击退/革命家/到期/愿望/风浪/谋杀罪/误杀/出/正义/现身/注意/几十年/相等/阴险/做/监狱/暗中/提名/血案/人数/洗雪/女明星/偷税/其才/弗里/大显/狱长/帮助/广告/没/越狱/一幅/犯罪/经济/石锤/脱离/躲藏/渐成/自己/窃贼/夜间/重视/遭受/翻案/多项/探悉/和睦/能够/年成/控告/彻底/罗宾斯/以/曼/担负/地奔/一度/释放/任意/导致/灰心/连同/一场/取得/扮演/难友/蒂姆/旧交/水星/救赎/计划/肖申克/燃起/向来/1995/答/假装/入狱/跟/初/针对/瑞德/受冤/情人/吗/下级/10/正在/偶然/证明/如果/派遣/杀死/引起/异/MorganFreeman/快/激发/确实/被/道格拉斯/和/电闪雷鸣/摩根/领队/一次/唯一/安迪/公/重获/件/找出/厘/奖励/长兄/就让/两人/一生/书籍/典狱长/一名/勇敢/本片/连/剧情/喽/少"
```

`thuac` 库分词结果如下：

```
"tags": "厘/以/件/确实/宾斯TimRobbins/夜间/任意/跟/一度/派遣/虚伪/受冤/和/向来/勇敢/提名/窃贼/年成/出/救赎/狱长/释放/情人/蒂姆/偶然/即使/妻/被/吗/下级/敝/官避税/肖申克/旧交/土星奖/广告/引起/宽度/燃起/导致/道格拉斯/长兄/典狱/本片/做/注意/扮演/计划/风浪/相同/电闪雷鸣/担负/入狱/彻底/没/喽/针对/越狱/10/找出/女人/书籍/长洗/洗雪/重视/领队/激发/连同/探悉/答/石锤/叫做/使用/紧靠/剧情/囚禁/几十/摩根·/境地/集市/如果/帮助/躲藏/相等/异/犯罪/增长/能够/消失/第一/初/正在/一生/非常/大多/总帐/正义/安迪/翻案/脱谋/合法/人数/1995年/赶快/经济/杀死/银行/连/取得/现身/弗里曼MorganFreeman饰/证明/难友/少/逐渐/自己/假装/唯一/球体/杀罪/击退/线/暗中/再/血案/瑞德/和睦/向/钱财/成为/监狱/坏/明星/控告/知识/奖励/误杀/遭受/到期/大显其才/灰心/阴险/愿望"
```

可以看到， `jieba` 库对外国人名的分词更加自然，同时更好地呈现了年份的分词结果，所以我们选择使用 `jieba` 库。

倒排表与布尔查询

这里，我们实现了多层跳表，以及对应的插入、删除、查询、合并等操作，详情参见 `src/skip_list.py`

我们还通过实现带符号栈的表达式求值模块，实现了布尔查询的功能，详情参见 `src/expression.py`

通过调节多层跳表的 `level` 值，我们测试了它与链表、一层跳表的性能差异，具体如下表所示。其中， `setup time` 代表建立跳表所需的时间， `query time` 代表跳表查询布尔表达式 `爱情 and 剧情` 所需的时间。

	链表	一层跳表	多层跳表
$setup\ time/s$	1.674	1.803	2.482
$query\ time/ms$	103	96	60

可以看到，随着跳表的层数增多，建立跳表所需的时间越来越长，但是单词查询的时间越来越短。且多层跳表相较于链表的性能提升接近一倍。

索引压缩

我们实现了按块存储和前缀压缩两种方式，详情参见 `src/index_compress.py` 。针对id进行压缩，按块压缩默认一块包含五个条目。我们以查询书籍中包含"爱情"标签的用例进行比较。其中，$dict$代表使用python内置的平衡树实现，$block$代表使用按块压缩，$trie_1$代表仅对id进行前缀压缩，$trie_2$代表对id进行前缀压缩，并且对条目内的标签也进行前缀压缩。

	$dict$	$block$	$trie_1$	$tire_2$
$query\ time/ms$	140	130	120	109
$memory/kB$	10968	11936	22732	35560

对于时间分析。按块压缩可以将总条目数n减小5倍，块内查询即为遍历所有的块，但时间开销较小，所以有一定的优化提升。前缀压缩，由于trie树查询的时间复杂度为$O(n)$，优于dict查询的时间复杂度$O(nlog(n))$，提升较为明显。将id条目中标签进一步压缩为trie则可以进一步提升。

对于空间分析。按块压缩中，我们需要将原本的id作为信息添加到每一个条目中，所以需要更多的内存空间。同时，按块压缩要求每一个词项占内存空间较小，可以从bit位节省空间；但是每一项书籍信息占用空间很大，对于id的压缩无法做到在bit位上节省空间。前缀压缩中，由于trie树的实现较为复杂，包含更多指针，所以需要用到更大的内存空间，并且对于id压缩使用的是数字，对于标签压缩使用的是汉字，相对英文字母有一定的劣势。

两种索引压缩在内存上表现得不理想，主要因为书籍信息的存储结构与词典略有不同，难以发挥两者的优势。

检索结果

三位同学学号后两位分别为63、64、65，对应电影分别为《本杰明·巴顿奇事》、《美丽心灵》、《穿条纹睡衣的男孩》。

输入剧情，检索结果如下：

```
1291545 大鱼 Big Fish ['剧情', '家庭', '奇幻', '冒险']
1291546 霸王别姬 ['剧情', '爱情', '同性']
1291549 放牛班的春天 Les choristes ['剧情', '音乐']
...
1306029 美丽心灵 A Beautiful Mind ['剧情', '传记']
...
1485260 本杰明·巴顿奇事 The Curious Case of Benjamin Button ['剧情', '爱情', '奇幻']
...
3008247 穿条纹睡衣的男孩 The Boy in the Striped Pajamas ['剧情', '战争']
...
```

面向《美丽心灵》检索，输入 `剧情 and 传记` ，结果如下：

```
1296736 钢琴家 The Pianist ['剧情', '音乐', '传记', '战争']
1305487 猫鼠游戏 Catch Me If You Can ['剧情', '传记', '犯罪']
1306029 美丽心灵 A Beautiful Mind ['剧情', '传记']
1308831 寻找梦幻岛 Finding Neverland ['剧情', '家庭', '传记']
1309015 摩托日记 Diarios de motocicleta ['剧情', '传记', '冒险']
1441602 霍元甲 ['剧情', '动作', '传记']
1476023 绝代艳后 Marie Antoinette ['剧情', '传记', '历史']
...
4164444 127小时 127 Hours ['剧情', '传记', '冒险']
```

输入 `福布斯 or 纳什 or 艾丽西亚` ，结果如下：

```
1306029 美丽心灵 A Beautiful Mind ['剧情', '传记']
3541415 盗梦空间 Inception ['剧情', '科幻', '悬疑', '冒险']
```

面向《本杰明·巴顿奇事》检索，输入 `剧情 and 爱情 and 奇幻` ，结果如下：

```
2135981 画皮 畫皮 ['剧情', '爱情', '惊悚', '奇幻']
2268359 暮光之城 Twilight ['剧情', '爱情', '惊悚', '奇幻']
3148027 暮光之城2：新月 The Twilight Saga: New Moon ['剧情', '爱情', '惊悚', '奇幻']
3289086 暮光之城3：月食 The Twilight Saga: Eclipse ['剧情', '爱情', '惊悚', '奇幻']
```

输入 `戴茜 or 本杰明 or 巴顿 or 凯若琳` ，结果如下：

```
1303173 甲方乙方 喜剧 ['喜剧']
1485260 本杰明·巴顿奇事 The Curious Case of Benjamin Button ['剧情', '爱情', '奇幻']
```

面向《穿条纹睡衣的男孩》检索，输入 `剧情 and 战争` ，结果如下：

```
1292849 拯救大兵瑞恩 Saving Private Ryan ['剧情', '战争']
1293318 萤火虫之墓 火垂るの墓 ['剧情', '动画', '战争']
1293764 与狼共舞 Dances with Wolves ['剧情', '西部', '冒险']
...
3008247 穿条纹睡衣的男孩 The Boy in the Striped Pajamas ['剧情', '战争']
...
```

输入 `布鲁诺 or 纳粹 or 条纹睡衣` ，结果如下：

```
1291565 疯狂约会美丽都 Les triplettes de Belleville ['喜剧', '动画']
1292063 美丽人生 La vita è bella ['剧情', '喜剧', '爱情', '战争']
1292272 大象 Elephant ['剧情', '犯罪']
1292504 柏林苍穹下 Der Himmel über Berlin ['剧情', '爱情', '奇幻']
1294408 音乐之声 The Sound of Music ['剧情', '爱情', '歌舞', '传记']
1295873 偷自行车的人 Ladri di biciclette ['剧情', '犯罪']
1296717 夺宝奇兵 Raiders of the Lost Ark ['动作', '冒险']
1296736 钢琴家 The Pianist ['剧情', '音乐', '传记', '战争']
1296753 卡萨布兰卡 Casablanca ['剧情', '爱情', '战争']
1307847 兄弟连 Band of Brothers ['剧情', '动作', '历史', '战争']
2028645 拆弹部队 The Hurt Locker ['剧情', '惊悚', '战争']
2028647 行动目标希特勒 Valkyrie ['剧情', '历史', '战争']
2213597 朗读者 The Reader ['剧情', '爱情']
2297265 浪潮 Die Welle ['剧情', '惊悚']
3008247 穿条纹睡衣的男孩 The Boy in the Striped Pajamas ['剧情', '战争']
```

3.推荐

实验要求

> *在这次实验中，你们需要自行划分训练集与测试集，在测试集上为用户对书籍或电影的评分进行排序，并用 NDCG 对自己的预测结果进行评分和进一步分析。*

协同过滤(collaborative filtering)

本次实验中，使用了协同过滤的方法进行推荐，具体来说，使用了基于物品的协同过滤算法(item-based collaborative filtering)和基于用户的协同过滤算法(user-based collaborative filtering)。由于本次数据集的规模很小，大约只有千余的用户和物品，所以我并没有使用矩阵分解的方法，而是直接使用了基于物品的协同过滤算法，因为矩阵分解的实质是对信息进行降维和压缩，而本次数据集的规模很小，可以很轻松地将表格放进内存里，所以没有必要使用矩阵分解的方法。

以基于用户的协同过滤为例，就是通过计算用户之间的相似度，然后根据相似度进行推荐。具体来说，对于用户u，我们可以找出与他相似度最高的k个用户，然后根据这k个用户对物品的评分，来预测用户u对物品i的评分。同样由于本次实验的数据规模很小，我直接采用了暴力的方法，即对于每一个用户，计算他与其他所有用户的相似度，然后取出相似度最高的k个用户，再根据这k个用户对物品的评分，来预测用户u对物品i的评分。

数据集

本次实验中使用的数据集来自于豆瓣，包含了用户对书籍和电影的评分和标签，在这次实验中，我只使用了评分数据。每个用户对每个物品的评分是一个整数，范围是1到5，0表示数据缺省。数据集中的每一行代表了一个用户对一个物品的评分。我们按照助教的要求对数据集进行了划分，将数据集划分为训练集和测试集，其中训练集测试集各占50%。

实验结果

我们使用了item-based和user-based两种方式来进行推荐，具体来说，对于每一个用户，我们都计算了他对所有物品的评分，然后根据评分进行排序，取出评分最高的前k个物品，作为推荐结果。我们使用NDCG来评价推荐结果的好坏，具体来说，对于每一个用户，我们都计算了他的NDCG值，然后对所有用户的NDCG值取平均，作为最终的NDCG值。由于部分用户的评分数据缺省，而且只有一条评分记录没有办法计算NDCG值，我们只对大于等于2条评分记录的用户进行了NDCG值的计算(助教给出的代码中没有遇到这个问题，可能是因为没有把0数据当成缺省数据)。

最终经过大致的调参处理得到结果如下：

```
book-item-based: k=50, NDCG=0.9600708303257246
book-user-based: k=50, NDCG=0.9595920774270024
movie-item-based: k=50, NDCG=0.9194036798029388
movie-user-based: k=50, NDCG=0.9180497961365143
```

而将助教使用的text_embedding作为baseline

```
book-item-based: k=50, NDCG=0.9374121103798688
```

明显效果要更好一些

从上面可以观察到，item-based相比user-based效果更好一些，但并没有好太多，这可能是因为物品之间的相似度相比较用户而言更加稳定，毕竟一千个人心中有一千个哈姆雷特。但同时也因为数据足够稠密，所以user-based的效果也不错，基本上与item-based相差无几。

同时我们还可以看到对书籍的推荐效果要比对电影的推荐效果好一些，这可能是因为用户对书籍的评价更加稳定，而对电影的评价更加主观，所以推荐效果也更好一些。

下面截取了一段对用户1000152的预测结果

```
user,true_rating,predict_rating
1000152,5,4.566170001649244
1000152,4,4.516068253175259
1000152,5,4.482258715879236
1000152,4,4.481394002177283
1000152,4,4.458774088735441
1000152,5,4.3947717089697615
1000152,5,4.389041677465942
1000152,4,4.3856065466102345
1000152,4,4.378603940114248
1000152,4,4.370014240495313
1000152,4,4.367251352559569
1000152,4,4.364222362532171
1000152,4,4.33770709963167
1000152,4,4.3194915531481035
1000152,5,4.302762048054981
1000152,4,4.3013200013038935
1000152,4,4.2986888542944035
1000152,5,4.2958791108620895
1000152,5,4.2856058553521725
1000152,5,4.282518394205787
1000152,5,4.265354779820212
1000152,5,4.2494654503074845
1000152,4,4.222667778353488
1000152,3,4.2129735482145465
1000152,4,4.198676395460678
1000152,4,4.196660097377075
1000152,4,4.1489869230596685
1000152,5,4.143826766156278
1000152,4,4.14081591737458
1000152,5,4.132485108368975
1000152,4,4.130899316551944
1000152,4,4.130836395607929
1000152,5,4.116643348904948
1000152,3,4.109268438474971
1000152,4,4.106396086816048
```

```
1000152,3,4.059257086667785
1000152,4,4.059227907599719
1000152,4,4.055981107145631
1000152,4,4.046904025740255
1000152,4,4.042455114679798
1000152,4,3.9973590540975628
1000152,5,3.9713286208233183
1000152,5,3.968893890935384
1000152,5,3.9281466737540316
1000152,3,3.9238573720592496
1000152,4,3.8916614619823338
1000152,4,3.886394678192446
1000152,5,3.877335867131819
1000152,4,3.8594183419647075
1000152,5,3.849155081664981
1000152,3,3.8451383921618225
1000152,4,3.8235571244924427
1000152,4,3.7491465980400247
1000152,4,3.746740823310879
1000152,4,3.738065969891922
1000152,3,3.732886915061242
1000152,2,3.7252631103302085
1000152,4,3.7139914007033212
1000152,4,3.692313435904507
1000152,3,3.652012403363075
1000152,4,3.64390042613042
1000152,4,3.5788682160725545
1000152,3,3.5663019785123082
1000152,5,3.5639008120505626
1000152,2,3.5269894091651564
1000152,4,3.5253247307238333
1000152,3,3.5134376055465335
1000152,4,3.5120247091530463
1000152,3,3.434923382975853
1000152,2,3.392627310147986
1000152,2,3.3709652398330845
1000152,4,3.304412880421148
1000152,3,3.2585945181348968
1000152,2,3.255823527717265
1000152,4,3.0686506568309624
1000152,3,3.0053974883409245
```

可以看到，我们给出的预测喜好顺序与真实的喜好顺序大体上还是比较接近的，但是也有一些差异。当然，因为真实评分有大量的相同值，可能也造成了NDCG值的偏高，但是最终0.96的NDCG值也说明了我们的预测结果还是比较准确的。

四、实验总结

通过本次实验，我们学习了爬虫的基本原理和实现方法，学习了检索的基本原理和实现方法，还学习了推荐的基本原理和实现方法。同时，我们也学习了如何使用python进行数据处理和分析，学习了如何使用python进行数据可视化。

通过本次实验，我们学习了倒排表的原理，学习并实现了多层跳表，并以此实现了简单的布尔查询系统。

通过本次实验，我们对信息检索及其应用有了更深入的了解，对推荐系统也有了更深入的了解。同时，我们也对python的使用更加熟练，也学会了如何使用github进行团队协作。