实验1实验报告

PB21000057 顾荣建　PB21000069 林润锋(组长)　PB21000074 田钦

第一阶段.豆瓣数据的爬取与检索

1.0 实验要求:

对于指定ID的电影、书籍，爬取其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，对于给定的查询，能够以精确查询或模糊语义匹配的方法返回最相关的书籍或者电影集。

1.1 爬虫

1.1.1 爬虫要求:

针对给定的电影、书籍ID，爬取其基本信息、简介、演员/作者等，选取网页爬取或API爬取方式，解析网页内容并提交所获数据。

1.1.2 所爬取信息:

电影:电影标题、电影评分、导演、演员、电影类型、上映日期、电影简介、"喜欢这部电影的人也喜欢"。
书籍:书籍标题、副标题、评分、作者、出版社、原名（对于外文书籍）、ISBN号、译者、出版时间、页码、价格、简介

1.1.3 爬虫实现:

1)采用python语言的requests库爬取网页，用bs4.BeautifulSoup库对爬取信息进行整理

2)爬虫方式:网页爬取

3)平台反爬措施:如果同一个IP频繁对豆瓣发出请求，那么豆瓣会封禁该IP

4)应对措施:一开始采用了隔几十秒(sleep(randit(40,60)))爬取一部电影的方式来避免被封禁IP，后面发现一个IP要频繁访问几百次才会被封禁，于是就取消了sleep，改为每爬取几百部电影就换一个IP继续爬的方式

1.1.4 爬虫代码:

`BeautifulSoup`库可以向网页发送访问请求从而得到网页等返回的html信息，并且其中提供的库函数可以对html源码进行解析，如`soup.select()`函数，可以在html源码中寻找特定的元素，可以根据类型、内容等进行筛选，具体实现如下(以爬取电影为例):

1)获取网页:

```
    url = f'https://movie.douban.com/subject/{movie_id}/'
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
         AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
    }
    response = requests.get(url, headers=headers)
    soup = BeautifulSoup(response.text, 'html.parser')
```

2)信息提取:

```
    # 获取电影标题
    chinese_title = soup.title.get_text().strip()[:-5]
    title_element = soup.select_one('h1 span[property="v:itemreviewed"]')
    title = title_element.text.strip() if title_element else ""

    # 获取电影评分
    rating_element = soup.select_one('strong.ll.rating_num')
    rating = rating_element.text.strip() if rating_element else "无"
    if(rating!="无"):
        rating = float(rating)

    # 获取导演
    director_elements = soup.select('a[rel="v:directedBy"]')
    directors = [director.text for director in director_elements]
```

3)csv文件读ID并调用爬虫函数:

```
    # 调用函数并输入豆瓣电影的ID和文件路径
    name = "Movie_details.csv"
    cnt = 0
    with open(name, 'r', encoding='utf-8') as f:   #跳过已经爬下的部分
        line = f.readline()
        while(line):
            cnt+= 1
            line = f.readline()

    with open('Movie_id.csv', 'r') as f:         #根据ID爬取对应电影
        line = f.readline()
        while(cnt>0):
            cnt-=1
            line = f.readline()
        while line:
            print(line)
            save_movie_details(line[:-1], name)
            line = f.readline()
            #sleep(random.randint(20, 40))
```

4)信息存入文件:

```
    with open(file_path, 'a', encoding='utf-8',newline="") as file:
        csv_writer = csv.writer(file)
        z = [
            movie_id,
            chinese_title,
            title,
            rating,
            directors,
            actors,
            genres,
```

```
            release_date,
            summary,
            recommendation_list
        ]
        csv_writer.writerow(z)
        print("写入数据成功")
        file.close()
```

1.2 信息检索

1.2.1 检索要求:

实现电影、书籍的bool检索，对内容简介进行分词、去停用词处理，表征为一系列关键词集合，建立倒排表来实现检索。

1.2.2 检索关键词提取:

电影: 电影关键词提取整合自导演、演员、电影类型以及剧情简介的分词
书籍: 书籍关键词提取整合自作者、标题以及书籍简介等的分词

1.2.3 所用分词工具及其差异:

我们组所采用的分词工具为jieba和snownlp，结果发现jieba要远远优于snownlp.

以电影为例，在分词用时上，jieba对1200部电影的剧情简介完成分词用时仅为3s，而snownlp则用时为100s，从准确度看，jieba分词也要远优于snownlp，例如在肖申克的救赎剧情简介开头中，两者分词分别为：
jieba:'一场', '谋杀案', '使', '银行家', '安迪'
snownlp:'一', '场', '谋杀', '案使', '银行家', '安迪'
通过观察整个分词结果，也可以明显发现jieba分词的准确率比snownlp准确率高很多

1.2.4 索引压缩优化

这里采用间隔存储+可变字节长度的方式对于倒排表索引进行压缩优化以减小存储需求

基本实现步骤如下（对于倒排表中某一项的列表a[]而言）：

1. 首先将每一项，将它改写为它和上一项的差
2. 改写为可变字长编码，由于我们只有1200个数据项，最多只需要两个字节即可存储，可以进行简化。

   若id<128，则用一个字节b=128+id存储；

   若id>=128，则用两个字节b1=128+(id>>7)，b0=id%128存储

压缩前后的对比（以电影运行10k次查询操作为例）

- 压缩前：
  - 倒排表的大小 533kb
  - 查询时间 136.012s
- 压缩后
  - 倒排表的大小 226kb
  - 查询时间 133.133s

可见压缩索引后在基本不影响查询时间的情况下，数据占用的空间大幅减少了（减少了近一半）

1.2.5 查询

对于任意合法的查询表达式（包含关键词以及三种逻辑运算符：`AND` / `OR` / `NOT`），我们可以先将它转化为对应的主析取范式，对于析取范式中的每一项，得到对应目标的得分，取一个目标在析取范式中的各项取得的最高分作为它自身的得分，最后进行比较，得到得分最高（也就是和查询条件匹配度）最高的前若干项

对于析取范式中的项，它一定由若干项合取得到，那么我们用匹配项/总项数得到这一合取式的得分，特别的，若某项存在`NOT`，则出现关键词不得分，不出现关键词则得分

1.3实验结果:

我们正确爬取了实验要求中的1200条豆瓣电影以及书籍的信息，同时对电影/书籍内容进行了分词处理，建立倒排表，并且能够根据用户输入的关键词布尔表达式检索与关键词匹配度最高的书籍/电影。

第二阶段.使用豆瓣数据进行推荐

2.0 实验要求:

基于第一阶段爬取的信息、提供的豆瓣电影书籍评分数据、用户间的社交关系等判断用户的偏好，预测用户对新的电影/书籍的评价，并进行偏好排序。

2.1实验实现:

采用协同过滤的方法来估计未知的评分，实现方法大致为：先将每一个用户看成一个向量，计算用户之间的相似性矩阵，然后通过用户之间的相似性来预测未知评分，采用预测分数与实际分数之间的方差以及对用户评分排序后的NDCG进行计算来评估实验结果。

2.1.1数据读取及划分:

采用panda.csv_reader读取数据，用一个embedding数组来划分训练集和测试集，具体实现方法为往embedding数组中存入等量0和1，然后随机打乱数组，最后1对应的数据项标记为测试集(未知评分)，0对应的数据项标记为训练集(已知评分)，如此刚好实现对数据集的对半划分。将划分好的数据缓存起来，后面读取数据直接从缓存中读取，代码如下:

```
def load_data_from_path(data_path, embedding, cache_dir):
    cache_path = os.path.join(cache_dir, "ratings_matrix_cache")

    # 加载数据集
    if os.path.exists(cache_path):
        ratings_matrix = pd.read_pickle(cache_path)
    else:
        ratings = pd.read_csv(data_path, dtype={
            "User": np.int32, "Book": np.int32, "Rate": np.float32
        }, usecols=range(3))
        length = len(ratings)
        for row in range(0, length):
            if embedding[row] == 1:
                ratings.at[row, "Rate"] = None
        ratings_matrix = pd.pivot_table(
            data=ratings, index=["User"], columns=["Book"], values="Rate")
```

```
        ratings_matrix.to_pickle(cache_path)

    return ratings_matrix
```

2.1.2预测分数:

调用python库的corr函数计算两个向量的相似性求得user的相似矩阵，对于每一个待预测的评分，先确定其所对应的User，然后在相似性矩阵中寻找评分跟该User相似的其他用户，通过向量相似计算的方法预测评分。如果找不到相似的用户，那么该评分则是无法预测的，此时将评分预测为均值2.5，预测评分代码如下:

```
numerator = 0
denominator = 0
for uid in relevant_users.index:
    similarity = relevant_users[uid]
    item_rating = rating_matrix.loc[uid].dropna()[item_id]
    numerator += similarity * item_rating
    denominator += similarity
```

2.1.3效果评估:

调用预测评分的函数预测embedding为1的数据项(即假装不知道评分的测试项)，采用两种方法来评估推荐的结果，由于数据集的项数过多，如果要对所有数据的预测进行评估耗时过久，因此只预测前几万项embedding数据评分来评估结果。
方法1:计算预测的分数与实际分数之间的平均平方差，平均平方差越小则代表预测越接近实际评分，另外为了反映推荐的效果，另外设置一个在0-5评分区间内瞎猜的平均平方差进行对比，代码如下:

```
for row in range(0, len(rating)):
    if embedding[row] == 1:
        user_id = rating.at[row, "User"]
        book_id = rating.at[row, "Movie"]
        real_rating = rating.at[row, "Rate"]
        predict_rating = functions.predict_score(user_id, book_id, rating_matrix,
user_similarity)
        random_guess_rating = random.randint(0, 6)
        cnt += 1
        user_ids.append(user_id)
        pred_ratings.append(predict_rating)
        real_ratings.append(real_rating)
        random_guess_ratings.append(random_guess_rating)
        var_predict += (predict_rating - real_rating) ** 2
        var_random_guess += (random_guess_rating - real_rating) ** 2
        if cnt % 10 == 0:
            print(cnt, var_predict / cnt, var_random_guess / cnt)
        if cnt == 10000:
            break
```

方法2:将评分进行排序，然后调用python的库计算预测结果总体的NDCG值，同样计算一个在0-5评分区间随机瞎猜的NDCG值，对比反映预测的效果，代码如下:

```
results.append(predict_result)
results = np.vstack(results)
results_df = pd.DataFrame(results, columns=['user', 'pred', 'true'])
results_df['user'] = results_df['user'].astype(int)
ndcg_scores = results_df.groupby("user").apply(functions.compute_ndcg)
avg_ndcg = ndcg_scores.mean()
print(f"Predict average NDCG: {avg_ndcg}")
```

2.2实验结果:

采用如2.1.3的方法，评估推荐效果如下:
电影的推荐效果:
3)7)$$SX825T~T3BP{3.jpg>)
对20000个测试数据进行预测，我们采用的算法预测分数与实际得分之间的平均平方差为2.012,NDCG值为0.8193,而随机猜测得到的平均平方差为6.623,NDCG值为0.6271，可以看出我们实现的推荐算法能够根据用户的喜好给出合理的评分预测。
书籍的推荐效果:
![Alt text](K7E~(ZFU~BK{@5{W5S5~RYS.jpg)
对20000个测试数据进行预测，我们采用的算法预测分数与实际得分之间的平均平方差为3.721,NDCG值为0.7582,而随机猜测得到的平均平方差为8.407,NDCG值为0.6739，可以看出书籍的预测效果比电影要差一些，不过从随机猜测的方差也比电影大可以看出书籍的评分两极分化可能更大一些，这也是书籍评分预测结果较差的客观原因，不过与随机猜测相对比，还是可以看出我们的预测算法可以根据用户喜好给出合理的预测。

3.0实验总结:

通过小组组员的分工合作不懈努力下，我们成功完成了实验一两个阶段的内容，在第一阶段，我们学习了爬虫的应用、数据的处理，了解了分词工具的应用，初步接触搜索引擎的一种简单实现，在第二阶段，我们学习了如何编写协同过滤算法来实现推荐。