期末课程报告——推荐系统设计

研究背景和方法

随着互联网技术的发展，从抖音的视频，到淘宝的商品，推荐系统在我们的生活中无处不在，这也引起了我在课堂之外自行探索推荐系统的兴趣。借着期末大作业能自由选题的机会，我决定采用课上学过的几种方式，实现一个简单的电影推荐系统，并对比各种方法之间的效果和优缺点。

采用的具体方法如下，将在后续详细分析：

1. 基于电影内容
2. 基于矩阵分解的协同过滤
3. 基于电影间相似度的协同过滤

1. 基于电影内容

算法原理

1. 电影特征表示：

电影的特征通过 `genres`（电影的类型）来表示。每部电影可以有多个类别（例如，动作片、喜剧片等），这些类别被拼接成一个字符串，并通过 **TF-IDF** 向量化方法转化为特征向量。

- `TfidfVectorizer` 用于将电影类别转化为 TF-IDF 向量。这些向量的目的是捕捉电影的类型信息（即，哪些类型是电影的重要标志）。

计算公式如下（t 表示相应词，d 表示相应文档，D 表示所有文档）：

$$\mathrm{TF}(t,d) = \frac{\text{词 } t \text{ 在文档 } d \text{ 中的出现次数}}{\text{文档 } d \text{ 中所有词的总数}}$$

$$\mathrm{IDF}(t,D) = \log\left(\frac{|D|}{1+\mathrm{df}(t,D)}\right)$$

$$\mathrm{TF-IDF}(t,d,D) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t,D)$$

最终的计算结果反映了相应词 t 在文档 d 中的重要性

具体代码如下：

```
# 使用电影标题和类别创建 TF-IDF 特征
movies['genres'] = movies['genres'].str.split('|').apply(lambda x: ' '.join(x))  # 将多个类别合并成一个字符串

# 创建 TF-IDF 向量
tfidf = TfidfVectorizer(stop_words='english')
tfidf_matrix = tfidf.fit_transform(movies['genres'])
```

2. 计算电影之间的相似度：

使用 **余弦相似度** 来衡量电影之间的相似度。通过计算电影的 TF-IDF 向量之间的余弦相似度矩阵，能够得到每两部电影之间的相似度分数。余弦相似度的公式如下：

$$\mathrm{sim}(i,j) = \frac{\mathbf{v}_i \cdot \mathbf{v}_j}{\|\mathbf{v}_i\|\|\mathbf{v}_j\|}$$

其中，$\mathbf{v}_i$和 $\mathbf{v}_j$ 是电影 i 和电影 j 的 TF-IDF 特征向量。

具体代码如下：

```
# 计算电影之间的余弦相似度
cosine_sim = cosine_similarity(tfidf_matrix, tfidf_matrix)
```

3. 相似电影的推荐：

- 在 `get_similar_movies` 函数中，根据用户给定的电影ID（例如电影ID = 1），首先确定该电影的索引，然后通过余弦相似度矩阵获取与该电影最相似的电影。
- 获取与目标电影相似的电影是通过索引位置 `idx`（电影的行索引）在余弦相似度矩阵中查找相似度分数。
- 根据排序后的相似度，返回与目标电影最相似的前 `top_n` 部电影。

具体代码如下：

```
# 获取与某部电影相似的电影
def get_similar_movies(movie_id, cosine_sim=cosine_sim, top_n=5):
    # 获取电影的索引
    idx = movies[movies['movieId'] == movie_id].index[0]

    # 获取该电影与所有其他电影的相似度
    sim_scores = list(enumerate(cosine_sim[idx]))

    # 按照相似度进行排序
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)

    # 获取相似度最高的前top_n部电影
    sim_scores = sim_scores[1:top_n+1]

    # 获取推荐的电影ID
    movie_indices = [i[0] for i in sim_scores]

    # 返回电影标题
    recommended_movies = movies.iloc[movie_indices]

    return recommended_movies[['movieId', 'title']]
```

4. 用户-电影评分矩阵：

接下来，创建一个用户-电影评分矩阵 `user_movie_matrix`，表示每个用户对电影的评分。未评分的位置被填充为该电影的平均评分，以确保矩阵没有缺失值。

```
user_movie_matrix = user_movie_matrix.apply(lambda x:
x.fillna(movie_avg_ratings[x.name]), axis=0)
```

这种填充方法让系统能够对没有评分的电影进行估计，从而能够在推荐时考虑到所有电影。

5. 评分预测：

`predict_ratings` 函数使用 **加权相似度评分预测** 方法来预测用户对未评分电影的评分。对于每个未评分的电影，预测评分是基于与该电影最相似的电影的评分来计算的，权重是电影之间的相似度。

预测评分的公式如下：

$$\hat{r}_{ui} = \frac{\sum_{v \in N(i)} \text{sim}(i, v) \cdot r_{uv}}{\sum_{v \in N(i)} |\text{sim}(i, v)|}$$

其中：

- $\hat{r}_{ui}$ 是用户 u 对电影 i 的预测评分。
- $N(i)$ 是与电影 i 相似的电影集合。
- $\text{sim}(i, v)$ 是电影 i 和电影 v 之间的相似度。
- r_{uv} 是用户 u 对电影 v 的实际评分。

具体代码如下：

```
def predict_ratings(user_movie_matrix, cosine_sim):
    # 将pred_matrix改为pandas的DataFrame对象
    pred_matrix = pd.DataFrame(np.zeros(user_movie_matrix.shape), columns=user_movie_matrix.columns, index=user_movie_matrix.index)

    for user_id in user_movie_matrix.index:
        for movie_id in user_movie_matrix.columns:
            if user_movie_matrix.at[user_id, movie_id] == 0:  # 只对未评分的电影进行预测
                # 获取与当前电影最相似的电影的评分
                # 需要确保 movie_id - 1 是有效的索引
                movie_idx = movie_id - 1  # cosine_sim 是 0-based index
                if movie_idx >= 0 and movie_idx < cosine_sim.shape[0]:  # 确保索引在有效范围内
                    sim_scores = cosine_sim[movie_idx]  # 获取与当前电影的相似度
                    weighted_ratings = 0
                    total_sim = 0
                    for other_movie_id in user_movie_matrix.columns:
                        if user_movie_matrix.at[user_id, other_movie_id] > 0:  # 用户已经评分的电影
                            other_movie_idx = other_movie_id - 1
                            if other_movie_idx >= 0 and other_movie_idx < cosine_sim.shape[0]:  # 确保索引有效
                                weighted_ratings += sim_scores[other_movie_idx] * user_movie_matrix.at[user_id, other_movie_id]
                                total_sim += abs(sim_scores[other_movie_idx])

                    if total_sim != 0:
                        pred_matrix.loc[user_id, movie_id] = weighted_ratings / total_sim

    return pred_matrix
```

结果分析

1. 为用户推荐的电影

该模型为 **用户1** 推荐的5部电影如图所示：

```
Recommended Movies:
      movieId  ...                                        genres
1706     2294  ...  Adventure Animation Children Comedy Fantasy
2355     3114  ...  Adventure Animation Children Comedy Fantasy
2809     3754  ...  Adventure Animation Children Comedy Fantasy
3000     4016  ...  Adventure Animation Children Comedy Fantasy
3568     4886  ...  Adventure Animation Children Comedy Fantasy
```

可以看到，推荐的电影的种类相同，这也正好应证了根据电影内容来给用户推荐电影的模型特点。当然，在实际生产应用当中，电影的内容提取范围不仅仅局限于电影种类，还包括电影导演，演员，发行年代等等。由于我的设备有限，故选择了相对简单，特征较少的数据集来训练。

2. RMSE（根均方误差）

为了从整体上来测试模型的训练效果，计算训练模型在测试集上预测评分的根均方误差。测试结果如图所示：

```
RMSE: 0.943564087621956
```

坦白说，这是一个糟糕的结果。毕竟根据计算，所有样本评分的根均方差为：

```
根均方差 (RMSE): 1.042524069617957
```

也就是说，该模型的预测结果，仅仅比直接猜测评分为平均值的准确度高了一点点。

2. 基于矩阵分解的协同过滤

算法原理

基于矩阵分解的协同过滤（Matrix Factorization-based Collaborative Filtering）是一种常见的推荐算法，尤其在推荐系统中用于处理稀疏的用户-物品评分矩阵。矩阵分解方法通过将大规模的稀疏矩阵分解成低秩矩阵的乘积，捕捉到潜在的用户和物品的隐式特征，从而可以推测出用户对未评分物品的评分。

以下是算法的具体步骤：

1. 用户-物品评分矩阵

在推荐系统中，用户与物品之间的关系通常可以通过一个 **评分矩阵** 来表示。这个矩阵的行是用户，列是物品（例如电影、商品等），矩阵中的每个元素 R_{ui} 表示用户 u 对物品 i 的评分。对于没有评分的用户-物品对，这些位置为空，构成了稀疏矩阵。

假设我们有如下的用户-物品评分矩阵 R，其中行表示用户，列表示物品，元素 R_{ui} 表示用户 u 对物品 i 的评分。

$$R = \begin{bmatrix} R_{11} & R_{12} & R_{13} & \dots & R_{1n} \\ R_{21} & R_{22} & R_{23} & \dots & R_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ R_{m1} & R_{m2} & R_{m3} & \dots & R_{mn} \end{bmatrix}$$

2. 矩阵分解：SVD

在矩阵分解方法中，我们通过将评分矩阵 R 分解成几个低秩矩阵的乘积。**SVD**（奇异值分解）是最常见的矩阵分解方法之一，它将矩阵 R 分解为三个矩阵的乘积：

$$R \approx U\Sigma V^T$$

- U 是一个 $m \times k$ 的矩阵，表示用户的隐式特征矩阵，其中 k 是潜在因子的数量（通常选择比用户数量和物品数量少得多的值）。
- Σ 是一个 $k \times k$ 的对角矩阵，包含了奇异值，这些奇异值表示了不同潜在因子的权重。
- V^T 是一个 $k \times n$ 的矩阵，表示物品的隐式特征矩阵。

通过这个分解，我们能够得到低维的用户特征和物品特征，进而计算用户对未评分物品的预测评分。

3. 训练和预测：

在代码中使用了 `SVD` 类，这代表了 **奇异值分解** 方法，`SVD` 类本质上是在执行上述矩阵分解的步骤。其训练过程可以总结为：

1. **训练模型**：使用训练集数据，分解评分矩阵为用户特征矩阵 U、物品特征矩阵 V 和奇异值矩阵 Σ。

   ```
   algo.fit(trainset)
   ```

2. **在测试集上进行预测**：训练完成后，模型可以用来预测用户对未评分物品的评分。预测值由模型通过计算用户特征和物品特征的内积得到。

   ```
   predictions = algo.test(testset)
   ```

   每个预测值 $\hat{R}_{ui}$ 通过用户 u 的特征向量和物品 i 的特征向量的点积得到：

   $$\hat{R}_{ui} = \mathbf{U_u} \cdot \mathbf{V_i}$$

   其中，$\mathbf{U_u}$ 是用户 u 的特征向量，$\mathbf{V_i}$ 是物品 i 的特征向量。

结果分析

1. 为用户推荐的电影

该模型为 **用户1** 推荐的5部电影如图所示：

	movieId	...	genres
602	750	...	Comedy|War
659	858	...	Crime|Drama
680	898	...	Comedy|Drama|Romance
694	912	...	Drama|Romance
841	1104	...	Drama

可以看到，推荐的类别与先前基于内容的推荐系统所推荐的电影截然不同，并且在电影种类上也更加多元化

2. RMSE（根均方误差）

为了从整体上来测试模型的训练效果，计算训练模型在测试集上预测评分的根均方误差。测试结果如图所示：

```
RMSE: 0.8698236433527861
```

该结果要优于基于内容的推荐系统，准确率差强人意。

3. 基于电影间相似度的协同过滤

算法原理

基于电影间相似度的协同过滤（Item-based Collaborative Filtering）是一种推荐算法，其核心思想是通过计算电影之间的相似度来为用户推荐电影。也就是说，如果一个用户喜欢某部电影，那么他/她也可能会喜欢与这部电影相似的其他电影。通过计算电影之间的相似度，我们能够根据用户已评分的电影，推荐与这些电影相似的其他电影。

下面介绍算法的具体实现过程：

1. 构建用户-电影评分矩阵

首先，使用 `pivot` 函数构建了一个 **用户-电影评分矩阵**：

```
user_movie_ratings = ratings.pivot(index='userId', columns='movieId',
values='rating').fillna(0)
```

- `pivot` 函数将原始的评分数据（`ratings` 数据框）转换为一个新的矩阵，其中行表示用户（`userId`），列表示电影（`movieId`），矩阵的每个元素表示用户对电影的评分。对于没有评分的电影，使用 `0` 填充（表示用户未评分这些电影）。

2. 计算电影之间的相似度

接下来，使用 **余弦相似度**（Cosine Similarity）来计算电影之间的相似度：

```
movie_similarity = cosine_similarity(user_movie_ratings.T)
```

- 这里的 `user_movie_ratings.T` 表示对用户-电影评分矩阵进行转置。转置后的矩阵是 **电影-用户评分矩阵**，其中每一行表示一部电影，每一列表示一个用户对这部电影的评分。通过转置，我们可以在电影之间计算相似度，而不是在用户之间计算。
- **余弦相似度** 是计算两个向量相似度的一种方法，具体公式为：

$$\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{\|A\| \|B\|}$$

其中，A 和 B 是两个电影的评分向量，"·" 表示向量的点积，$\|A\|$ 和 $\|B\|$ 是向量的模。

余弦相似度值的范围在 [0, 1] 之间，值越接近 1，表示两部电影的相似度越高。

3. 将相似度矩阵转换为DataFrame

然后，将计算得到的电影相似度矩阵转换为一个 `DataFrame`，以便查看和分析：

```
movie_similarity_df = pd.DataFrame(movie_similarity,
index=user_movie_ratings.columns, columns=user_movie_ratings.columns)
```

- 这里使用 `movie_similarity_df` 将相似度矩阵转化为 DataFrame，并且将电影的 ID（`movieId`）作为行和列的索引。这样，你可以方便地查看任意两部电影之间的相似度。

4. 推荐系统的实现

一旦计算出了电影之间的相似度，你就可以基于用户的历史评分进行推荐：

1. **选择用户已经评分的电影**：对于一个用户，首先获取他们评分过的电影列表。
2. **找到与这些电影相似的电影**：通过电影相似度矩阵，查找与这些已评分电影相似度较高的其他电影。
3. **根据相似度加权推荐电影**：可以基于电影的相似度来加权预测用户对未评分电影的评分，类似于：

$$\hat{r}_{u,i} = \frac{\sum_{j \in \text{Rated Movies}} \text{Sim}(i, j) \cdot r_{u,j}}{\sum_{j \in \text{Rated Movies}} |\text{Sim}(i, j)|}$$

其中：

- $\hat{r}_{u,i}$ 是用户 u 对电影 i 的预测评分。
- $r_{u,j}$ 是用户 u 对电影 j 的实际评分。
- $\text{Sim}(i, j)$ 是电影 i 和电影 j 之间的相似度。

通过这种方式，基于电影相似度的协同过滤推荐系统可以为用户推荐与他们已经喜欢的电影相似的电影。

结果分析

1. 为用户推荐的电影

该模型为 **用户1** 推荐的5部电影如图所示：

	movieId	title	genres
793	1036	Die Hard (1988)	Action|Crime|Thriller
902	1200	Aliens (1986)	Action|Adventure|Horror|Sci-Fi
1158	1527	Fifth Element, The (1997)	Action|Adventure|Comedy|Sci-Fi
1445	1968	Breakfast Club, The (1985)	Comedy|Drama
2195	2918	Ferris Bueller's Day Off (1986)	Comedy

可以看到与基于矩阵分解的协同过滤方法得到的推荐电影的种类有相似之处，但依然是截然不同的电影。

2. RMSE（根均方误差）

此时的 RMSE 达到较优值，比先前的两种方法表现都要更好：

```
RMSE: 0.788020733061898
```

总结与思考

通过这次实验，我对于不同种类的推荐系统有了更深刻的理解。当然，本次实验的数据集以及测试方法都还有很大的局限性，例如我们常常更关心推荐系统对于高分电影预测的准确性，而不太关心低分电影预测的准确性，而 **RMSE** 赋予了两者相同的地位。所以本次实验的测试结果，并不能直接简单地反映出这三种方法的孰优孰劣。

结合本次实验结果，通过查阅相关资料，我得出了以下表格表示三种方法的适用场景以及优缺点：

方法	优点	缺点
基于内容的推荐系统	不需要其他用户数据，容易解释	推荐单一，难以发现用户潜在兴趣
基于矩阵分解的协同过滤	能够发现潜在兴趣，适应大规模数据	冷启动问题，计算复杂，缺乏可解释性
基于电影相似度的协同过滤	计算简单直观，推荐质量较好	稀疏性问题，冷启动问题，扩展性差

最后谈谈我的一些思考：
本次实验让我感受最深的一点是函数库的强大，在进行基于矩阵分解的协同过滤时，基本所有的过程，包括计算 **RMSE**，都是通过调用函数库实现的。而调库实现的算法，往往比我辛辛苦苦敲出来的更快更准。在本次实验中，我试图阅读函数库的源代码，但碍于规模太大，时间有限，并没有深入去理解。但这也给了我一个很好的启发：今后在学习相关的算法时，不妨仔细阅读一下一些函数库的源代码，体悟相应算法在目前的最高效率实现，相信会有很大的收获。