Assignment6：实现Latent factor model和Collaborative filtering model

作业要求

1. 实现实现Latent factor model和Collaborative filtering model
2. 找公开的MovieLens任何一个数据集(https://grouplens.org/datasets/movielens/)进行测试（随机划分80%训练、20%测试），对比两个算法的RMSE

作业过程

数据集划分：将数据集划分为80%训练，20%测试，为了避免随机划分时某一用户的数据全部划分到测试集，采用对每一用户的rate数据以4：1比例划分到训练集和测试集的方式。在实验过程中特别是在实现Collaborative filtering model的过程中，需要构造表存储中间数据以便更快速地计算出未知的rate值，但每个表的构造都会耗费一定时间，因此采用保存中间表数据再运行时读取的方式来加快实验效率。

```python
def get_TrainTest_data(mat):
    train_matrix = pd.DataFrame(np.nan, index=mat.index, columns=mat.columns)
    test_matrix = pd.DataFrame(np.nan, index=mat.index, columns=mat.columns)
    for user_id in mat.index:
        non_nan_indices = mat.loc[user_id].dropna().index.tolist()
        train_indices, test_indices = train_test_split(non_nan_indices,
test_size=0.2, random_state=43)
        train_matrix.loc[user_id, train_indices] = mat.loc[user_id,
train_indices]
        test_matrix.loc[user_id, test_indices] = mat.loc[user_id, test_indices]
    train_matrix.to_csv('tmp-matrix/train_matrix.csv')
    test_matrix.to_csv('tmp-matrix/test_matrix.csv')
    return train_matrix, test_matrix
```

一、Latent factor model

算法原理

将user-movie矩阵分解$R \approx UV^T$，其中矩阵 U 为 $m \times k$，矩阵 V 为 $n \times k$，训练结果可通过 $\hat{r}_{ij} = U_i \cdot V_j^T = \sum_{f=1}^{k} U_{if} \cdot V_{jf}$ 直接得到，任务为使用训练集已知的 rate，计算 $\hat{r}_{ij}$ 和实际训练集rate的误差，得到损失函数$L = \sum_{(i,j)\in\mathcal{R}} (r_{ij} - \hat{r}_{ij})^2 + \lambda \left(\sum_{i=1}^{m} \|U_i\|^2 + \sum_{j=1}^{n} \|V_j\|^2 \right)$，迭代优化过程如下$U_i = U_i + \alpha \left[(r_{ij} - \hat{r}_{ij})V_j - \lambda U_i \right]$，$V_j = V_j + \alpha \left[(r_{ij} - \hat{r}_{ij})U_i - \lambda V_j \right]$

代码实现

Latent factor model主要过程如下：

```python
def LFM(train_matrix,K,max_iter,lr,reg):
    M, N = train_matrix.shape
    U = np.random.normal(scale=1./K, size=(M, K))
    V = np.random.normal(scale=1./K, size=(N, K))
    for step in range(max_iter):
        cost = 0
        for u in range(M):
            for i in range(N):
                if train_matrix[u][i] > 0:
                    err = train_matrix[u][i] - np.dot(U[u,:],V[i,:])
                    cost +=  err ** 2
                    U[u, :] += lr * (err * V[i, :] - reg * U[u, :])
                    V[i, :] += lr * (err * U[u, :] - reg * V[i, :])
        if cost < 5:
            break
    return np.dot(U,V.T)
```

计算RMSE用于比对效果

```python
def RMSE(pred_R,test_matrix):
    M, N = pred_R.shape
    cost = 0
    for u in range(M):
        for i in range(N):
            if test_matrix[u][i] > 0:
                cost += (pred_R[u][i] - test_matrix[u][i]) ** 2
    return pow(cost/np.count_nonzero(test_matrix),0.5)
```

二、Collaborative filtering model

算法原理

1. 构造不同user_id之间的相似度表$sim(x,y) = cos(r_x, r_y) = \frac{r_x r_y}{||r_x|| ||r_y||}$
2. 计算r_{xi}的baseline预估表，$b_{xi} = u + b_x + b_i$
3. 根据相似度表，找到与x相似度最高的k个user_id，存在如下公式N中，可计算未知的rate：$r_{xi} = b_{xi} + \frac{\sum_{y=N} sim_{xy}(r_{yi} - b_{yi})}{\sum_{y=N} sim_{xy}}$

代码实现

构建用户的movie交集数量表，相当于计算$r_x r_y$部分

```python
def gen_cross_nums(mat):
    cross_nums = pd.DataFrame(0, index = mat.index, columns = mat.index)
    for column_name, column_data in mat.items():
        non_nan_column_data = column_data.dropna()
        for index_1, value_1 in non_nan_column_data.items():
            for index_2, value_2 in non_nan_column_data.items():
                if index_1 == index_2:
                    continue
                cross_nums.loc[index_1][index_2] += value_1 * value_2
    cross_nums.to_csv('tmp-matrix/cross_nums.csv')
    return cross_nums
```

构建用户的movie数量表，相当于计算$||r_x||$和$||r_y||$部分

```
def gen_movie_nums(mat):
    mat = mat.fillna(0)
    movie_nums = pd.DataFrame(0.0, index=mat.index, columns=['movie_nums'])
    for index, row_data in mat.iterrows():
        row_sum = row_data.sum(skipna=True)
        movie_nums.loc[index][0] = row_sum
    movie_nums.to_csv('tmp-matrix/movie_nums.csv')
    return movie_nums
```

构建用户的相似度表，即整合计算$\frac{r_x r_y}{\|r_x\|\|r_y\|}$部分得到$sim(x, y)$

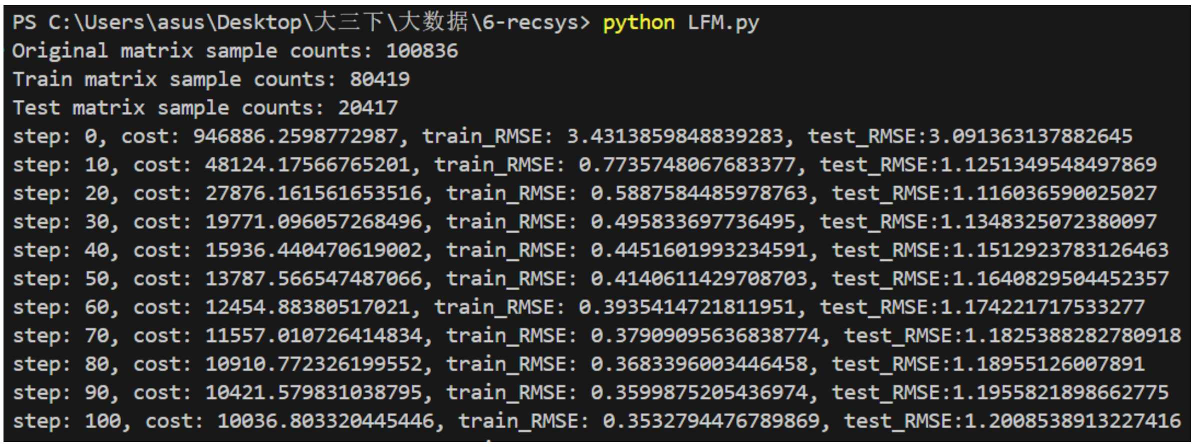
```
def gen_user_sim(mat,cross_nums,movie_nums):
    user_sim = pd.DataFrame(0.0, index=mat.index, columns=mat.index)
    for user1_id, users2 in cross_nums.items():
        for user2_id, internums in users2.items():
            user1_id = int(user1_id)
            user2_id = int(user2_id)
            user_sim.loc[user1_id][user2_id] = internums /
pow(movie_nums.loc[user1_id][0] * movie_nums.loc[user2_id][0],0.5)
    user_sim.to_csv('tmp-matrix/user_sim.csv')
    return user_sim
```

计算预测未知的rate值，对应$r_{xi} = \frac{\sum_{y=N} sim_{xy} r_{yi}}{\sum_{y=N} sim_{xy}}$部分

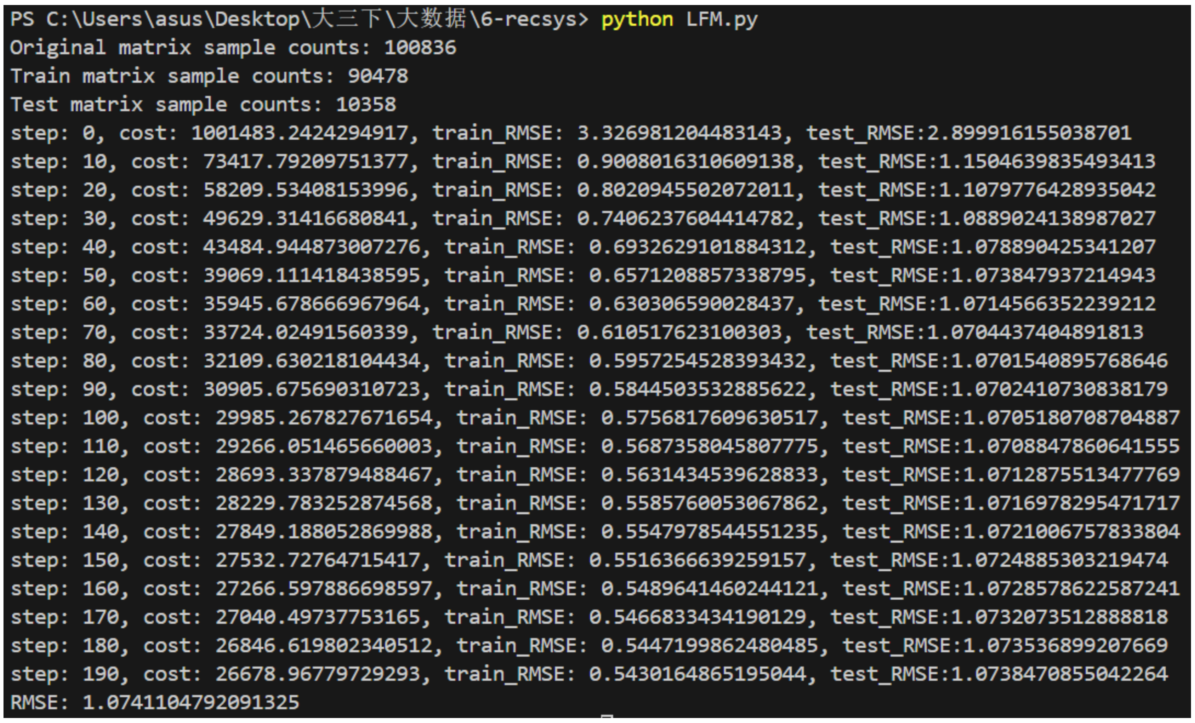
```
def calcu_rate(user_id,movie_id,k,train_matrix,user_sim):
    train_matrix_filled = train_matrix.fillna(0)
    if train_matrix_filled.loc[user_id][str(movie_id)] > 0:
        return train_matrix.loc[user_id][str(movie_id)]
    sims = user_sim.loc[user_id]
    top_k_sims = sims.sort_values(ascending=False).head(k)
    top_k_users = top_k_sims.index.tolist()
    pred_rate = 0
    sim_sum = 0
    for related_user in top_k_users:
        pred_rate += user_sim.loc[user_id][related_user] *
train_matrix_filled.loc[int(related_user)][str(movie_id)]
        sim_sum += user_sim.loc[user_id][related_user]
    pred_rate /= sim_sum
    return pred_rate
```

三、运行结果

1. Latent factor model：下面第一张图使用正则化系数0.01，可看到在epoch = 20 后，虽然训练集的RMSE还在下降，但测试集的RMSE已经出现反弹，推测是过拟合的问题，将正则化系数调为0.1，该现象得到缓和，测试集的RMSE损失进一步下降，效果更好。

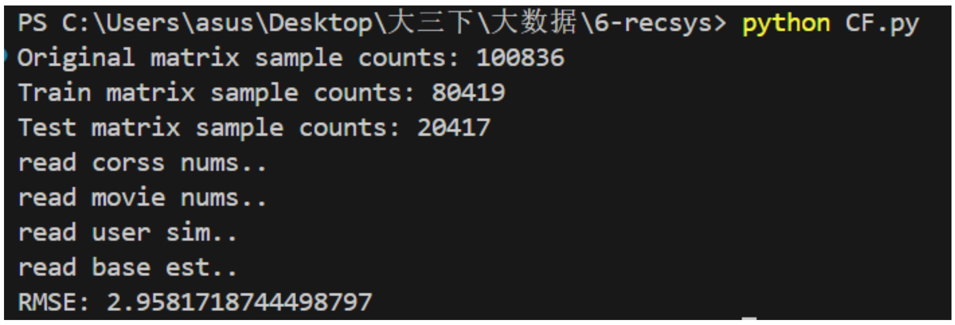


```
PS C:\Users\asus\Desktop\大三下\大数据\6-recsys> python LFM.py
Original matrix sample counts: 100836
Train matrix sample counts: 80419
Test matrix sample counts: 20417
step: 0, cost: 946886.2598772987, train_RMSE: 3.4313859848839283, test_RMSE:3.091363137882645
step: 10, cost: 48124.17566765201, train_RMSE: 0.7735748067683377, test_RMSE:1.1251349548497869
step: 20, cost: 27876.161561653516, train_RMSE: 0.5887584485978763, test_RMSE:1.116036590025027
step: 30, cost: 19771.096057268496, train_RMSE: 0.495833697736495, test_RMSE:1.1348325072380097
step: 40, cost: 15936.440470619002, train_RMSE: 0.4451601993234591, test_RMSE:1.1512923783126463
step: 50, cost: 13787.566547487066, train_RMSE: 0.4140611429708703, test_RMSE:1.1640829504452357
step: 60, cost: 12454.88380517021, train_RMSE: 0.3935414721811951, test_RMSE:1.174221717533277
step: 70, cost: 11557.010726414834, train_RMSE: 0.37909095636838774, test_RMSE:1.1825388282780918
step: 80, cost: 10910.772326199552, train_RMSE: 0.3683396003446458, test_RMSE:1.18955126007891
step: 90, cost: 10421.579831038795, train_RMSE: 0.3599875205436974, test_RMSE:1.1955821898662775
step: 100, cost: 10036.803320445446, train_RMSE: 0.3532794476789869, test_RMSE:1.2008538913227416
```



```
PS C:\Users\asus\Desktop\大三下\大数据\6-recsys> python LFM.py
Original matrix sample counts: 100836
Train matrix sample counts: 90478
Test matrix sample counts: 10358
step: 0, cost: 1001483.2424294917, train_RMSE: 3.326981204483143, test_RMSE:2.899916155038701
step: 10, cost: 73417.79209751377, train_RMSE: 0.9008016310609138, test_RMSE:1.1504639835493413
step: 20, cost: 58209.53408153996, train_RMSE: 0.802094550207201l, test_RMSE:1.1079776428935042
step: 30, cost: 49629.31416680841, train_RMSE: 0.7406237604414782, test_RMSE:1.0889024138987027
step: 40, cost: 43484.944873007276, train_RMSE: 0.6932629101884312, test_RMSE:1.078890425341207
step: 50, cost: 39069.111418438595, train_RMSE: 0.6571208857338795, test_RMSE:1.073847937214943
step: 60, cost: 35945.678666967964, train_RMSE: 0.630306590028437, test_RMSE:1.0714566352239212
step: 70, cost: 33724.02491560339, train_RMSE: 0.610517623100303, test_RMSE:1.0704437404891813
step: 80, cost: 32109.630218104434, train_RMSE: 0.5957254528393432, test_RMSE:1.0701540895768646
step: 90, cost: 30905.675690310723, train_RMSE: 0.5844503532885622, test_RMSE:1.0702410730838179
step: 100, cost: 29985.267827671654, train_RMSE: 0.5756817609630517, test_RMSE:1.0705180708704887
step: 110, cost: 29266.051465660003, train_RMSE: 0.5687358045807775, test_RMSE:1.0708847860641555
step: 120, cost: 28693.337879488467, train_RMSE: 0.5631434539628833, test_RMSE:1.0712875513477769
step: 130, cost: 28229.783252874568, train_RMSE: 0.5585760053067862, test_RMSE:1.0716978295471717
step: 140, cost: 27849.188052869988, train_RMSE: 0.5547978544551235, test_RMSE:1.0721006757833804
step: 150, cost: 27532.72764715417, train_RMSE: 0.5516366639259157, test_RMSE:1.0724885303219474
step: 160, cost: 27266.597886698597, train_RMSE: 0.5489641460244121, test_RMSE:1.0728578622587241
step: 170, cost: 27040.49737753165, train_RMSE: 0.5466833434190129, test_RMSE:1.0732073512888818
step: 180, cost: 26846.619802340512, train_RMSE: 0.5447199862480485, test_RMSE:1.073536899207669
step: 190, cost: 26678.96779729293, train_RMSE: 0.5430164865195044, test_RMSE:1.0738470855042264
RMSE: 1.0741104792091325
```

2. Collaborative filtering：可以看到得到的RMSE值是相对较大的，计算效果不如使用Latent factor model



```
PS C:\Users\asus\Desktop\大三下\大数据\6-recsys> python CF.py
Original matrix sample counts: 100836
Train matrix sample counts: 80419
Test matrix sample counts: 20417
read corss nums..
read movie nums..
read user sim..
read base est..
RMSE: 2.9581718744498797
```

总结

本次实验分别实现了Latent factor model和Collaborative filtering model，从两个不同角度熟悉了推荐系统的运行原理，但 Collaborative filtering 实现的效果并不是很理想，仅观察便能看出有很多实验结果不成规律，后续对Collaborative filtering通过划分数据集为9：1进行尝试，得到RMSE为2.8292，有所改善但效果也不是特别好，未找到改进之处，总体上还是对两种方法的区别以及优势有了更深了解。