HW7：混合推荐系统设计

21304219 刘文婧

MovieLens数据集

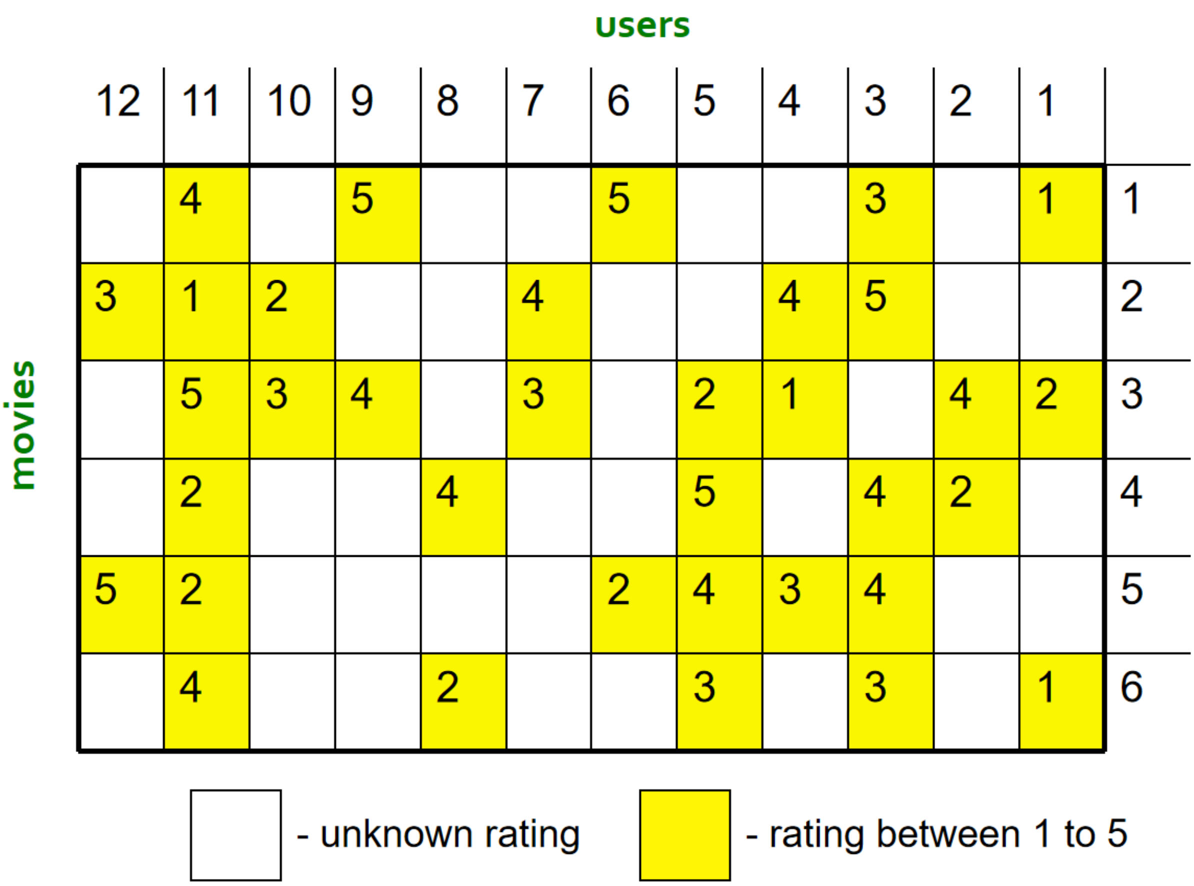
> https://grouplens.org/datasets/movielens/

- 使用官网提供的最小的`MovieLens`数据集进行实验：Small: 100,000 ratings and 3,600 tag applications applied to 9,000 movies by 600 users.
- 每位用户至少对20部电影进行了打分，每部电影至少有1个 rating 或 tag。具体包括如下文件：
- `links.csv` : not necessary in this task
- `movies.csv` : $movieId,\ title,\ genres$
- `ratings.csv` : $userId,\ movieId,\ rating,\ timestamp$
- `tags.csv` : not used here

Collaborative Filtering

Utility Matrix

- 构建一个类似于下面的效用矩阵，行、列分别是 $userId, movieId$



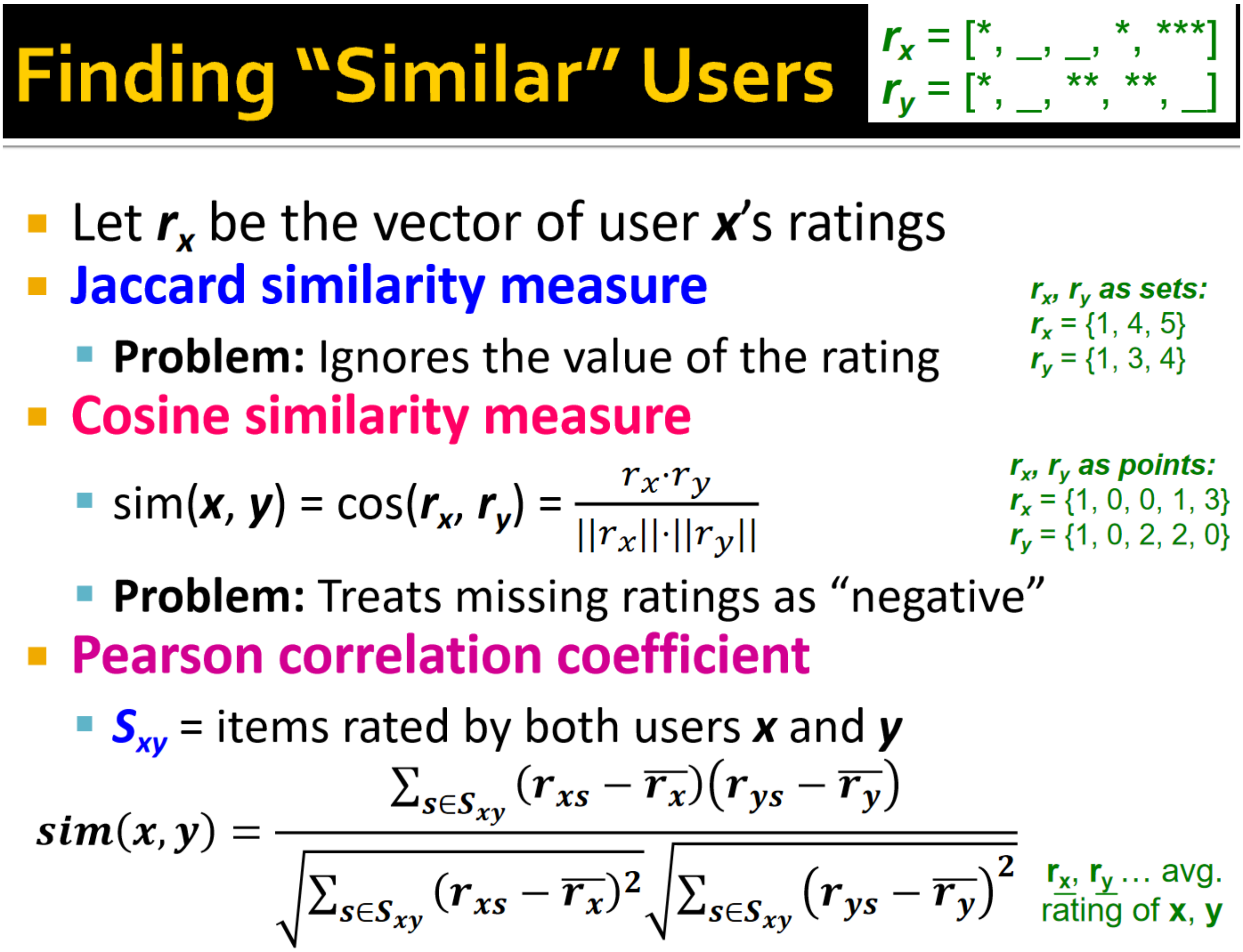
- 构建好后如下：

```
movieId  1       2       3       4       5       6       ...  193579  193581  193583  193585  193587  193609
userId                                                   ...
1            4.0     NaN     4.0     NaN     NaN     4.0 ...     NaN     NaN     NaN     NaN     NaN     NaN
2            NaN     NaN     NaN     NaN     NaN     NaN ...     NaN     NaN     NaN     NaN     NaN     NaN
3            NaN     NaN     NaN     NaN     NaN     NaN ...     NaN     NaN     NaN     NaN     NaN     NaN
4            NaN     NaN     NaN     NaN     NaN     NaN ...     NaN     NaN     NaN     NaN     NaN     NaN
5            4.0     NaN     NaN     NaN     NaN     NaN ...     NaN     NaN     NaN     NaN     NaN     NaN
...          ...     ...     ...     ...     ...     ... ...     ...     ...     ...     ...     ...     ...
606          2.5     NaN     NaN     NaN     NaN     NaN ...     NaN     NaN     NaN     NaN     NaN     NaN
607          4.0     NaN     NaN     NaN     NaN     NaN ...     NaN     NaN     NaN     NaN     NaN     NaN
608          2.5     2.0     2.0     NaN     NaN     NaN ...     NaN     NaN     NaN     NaN     NaN     NaN
609          3.0     NaN     NaN     NaN     NaN     NaN ...     NaN     NaN     NaN     NaN     NaN     NaN
610          5.0     NaN     NaN     NaN     NaN     5.0 ...     NaN     NaN     NaN     NaN     NaN     NaN

[610 rows x 9724 columns]
```

item-item CF

- 衡量 Similarity：



- 对 $item = movie$，实验中就选用 Pearson 相关系数去衡量

$$S_{xy} = \text{movies rated by both users } x \text{ and } y$$

$$sim(x,y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)(r_{ys} - \bar{r}_y)}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)^2} \cdot \sqrt{\sum_{s \in S_{xy}} (r_{ys} - \bar{r}_y)^2}}$$

$$\bar{r}_x, \bar{r}_y \ \ldots \text{ average rating of } x, y \ on \ S_{xy}$$

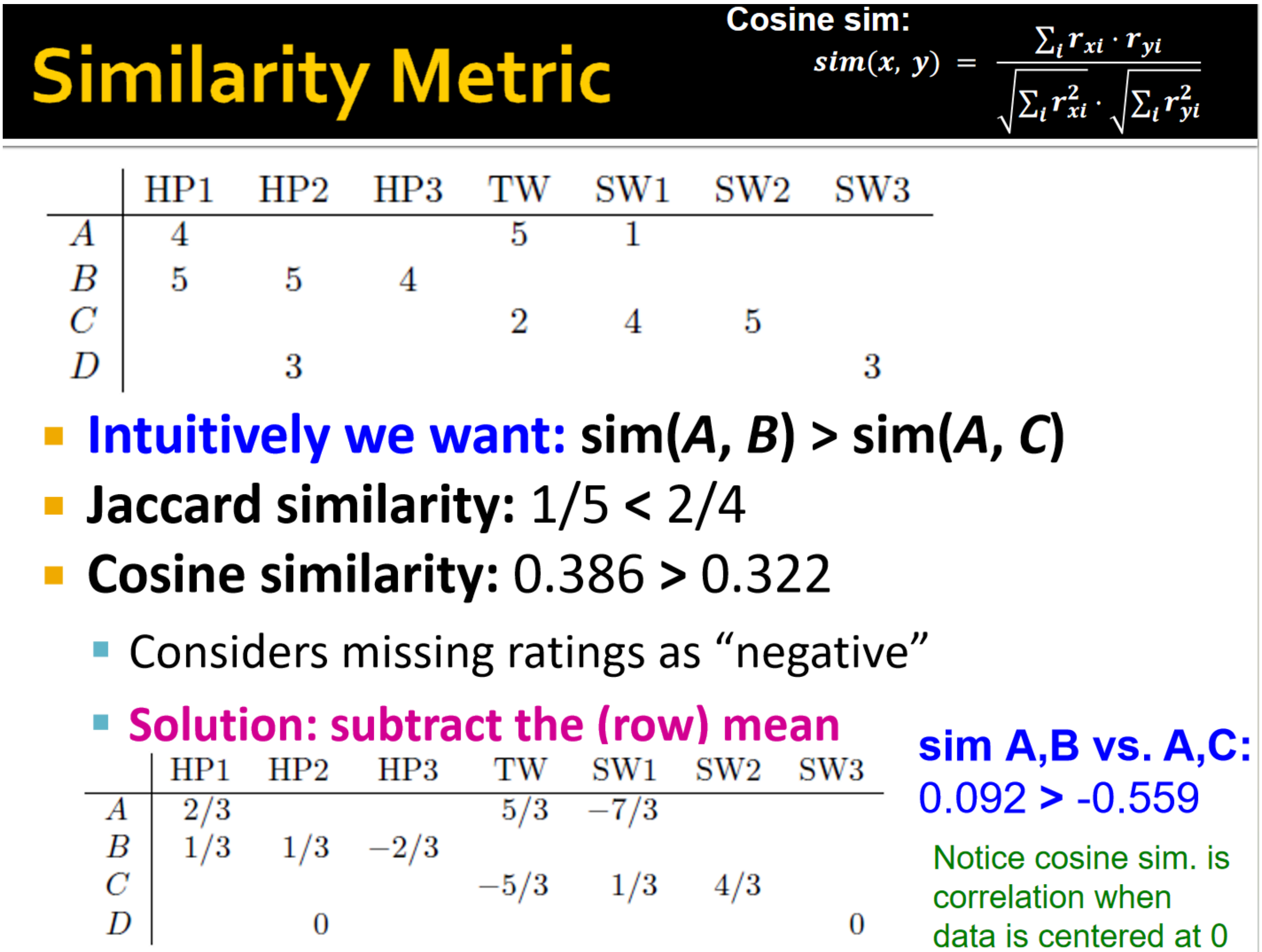
- 用 `pandas.DataFrame.corr(method='Pearson')`

```
userId        1         2         3         4         5    ...       606       607       608       609       610
userId                                                     ...
1      1.000000       NaN  0.079819  0.207983  0.268749  ...  0.066378  0.174557  0.268070 -0.175412 -0.032086
2           NaN  1.000000       NaN       NaN       NaN  ...  0.583333       NaN -0.125000       NaN  0.623288
3      0.079819       NaN  1.000000       NaN       NaN  ... -0.791334 -0.333333 -0.395092       NaN  0.569562
3      0.079819       NaN  1.000000       NaN       NaN  ... -0.791334 -0.333333 -0.395092       NaN  0.569562
4      0.207983       NaN       NaN  1.000000 -0.336525  ...  0.144603  0.116518 -0.170501 -0.277350 -0.043786
5      0.268749       NaN       NaN -0.336525  1.000000  ...  0.244321  0.231080 -0.020546  0.384111  0.040582
4      0.207983       NaN       NaN  1.000000 -0.336525  ...  0.144603  0.116518 -0.170501 -0.277350 -0.043786
5      0.268749       NaN       NaN -0.336525  1.000000  ...  0.244321  0.231080 -0.020546  0.384111  0.040582
...         ...       ...       ...       ...       ...  ...       ...       ...       ...       ...       ...
606    0.066378  0.583333 -0.791334  0.144603  0.244321  ...  1.000000  0.114191  0.240842  0.533002  0.389185
607    0.174557       NaN -0.333333  0.116518  0.231080  ...  0.114191  1.000000  0.200814  0.190117  0.106605
608    0.268070 -0.125000 -0.395092 -0.170501 -0.020546  ...  0.240842  0.200814  1.000000  0.488929  0.147606
609   -0.175412       NaN       NaN -0.277350  0.384111  ...  0.533002  0.190117  0.488929  1.000000 -0.521773
610   -0.032086  0.623288  0.569562 -0.043786  0.040582  ...  0.389185  0.106605  0.147606 -0.521773  1.000000

[610 rows x 610 columns]
```

user-user CF

- 实验中用改进后的 $CosineSimilarity$ 去衡量 $user$ 之间的相似度：



- 也就是先对每个用户的评分做中心化（减去该用户的平均评分）
- 再来算

$$sim(x,y) = \frac{\sum_i r_{xi} \cdot r_{yi}}{\sqrt{\sum_i r_{xi}^2} \cdot \sqrt{\sum_i r_{yi}^2}}$$

关于2个metrics

评估 user-user CF 时使用的 $metric$:

- Pearson correlation coefficient **可以看作是局部中心化** $user$ **的评分向量（仅对共同评分** $item$ **对象中心化），再来计算** $user$ **之间的** Cosine similarity **（仅基于共同评分** $item$ **计算）**；

评估 item-item CF 时使用的 $metric$:

- Adjusted Cosine similarity **可以看作是全局中心化** $user$ **的评分向量，再来计算** $item$ **之间的** Cosine similarity。

Pearson correlation coefficient 和 Cosine similarity 在数据经过零均值+单位方差标准化后，确实是一致的。

在实际实验中，发现会有一定数量的 $user$ **之间相似性为** 1**，而这确实是正确的：**

- 比如我发现计算出来 $user2$ 和 $user33$ 之间相似性就是 1，在表格中找到二者的评分重合部分如下：

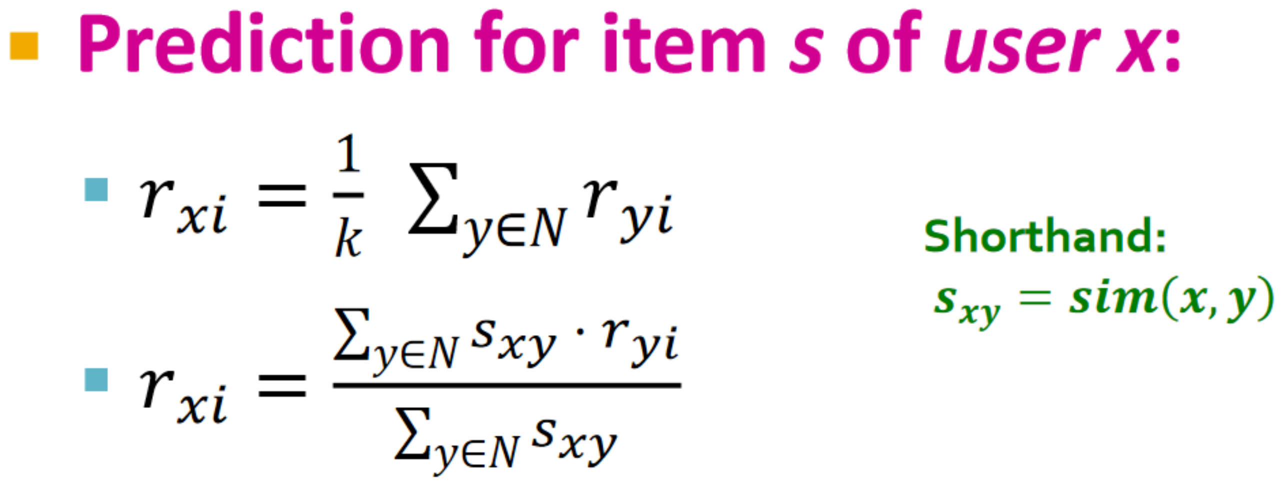
userId	movieId	rating
2	318	3
2	1704	4.5
33	318	4
33	1704	5

- 确实评分变化趋势完全一致，成比例线性关系，所以 Pearson Similarity 计算得到 1 是正确的结果：

$$
\begin{aligned}
\text{Pearson}(u_2, u_{33}) &= \frac{(3.0-3.75)(4.0-4.5)+(4.5-3.75)(5.0-4.5)}{\sqrt{(3.0-3.75)^2+(4.5-3.75)^2}\cdot\sqrt{(4.0-4.5)^2+(5.0-4.5)^2}} \\
&= \frac{(-0.75)\cdot(-0.5)+(0.75)\cdot(0.5)}{\sqrt{(-0.75)^2+(0.75)^2}\cdot\sqrt{(-0.5)^2+(0.5)^2}} = 1.0
\end{aligned}
$$

Prediction (Recommend n movies to a user)

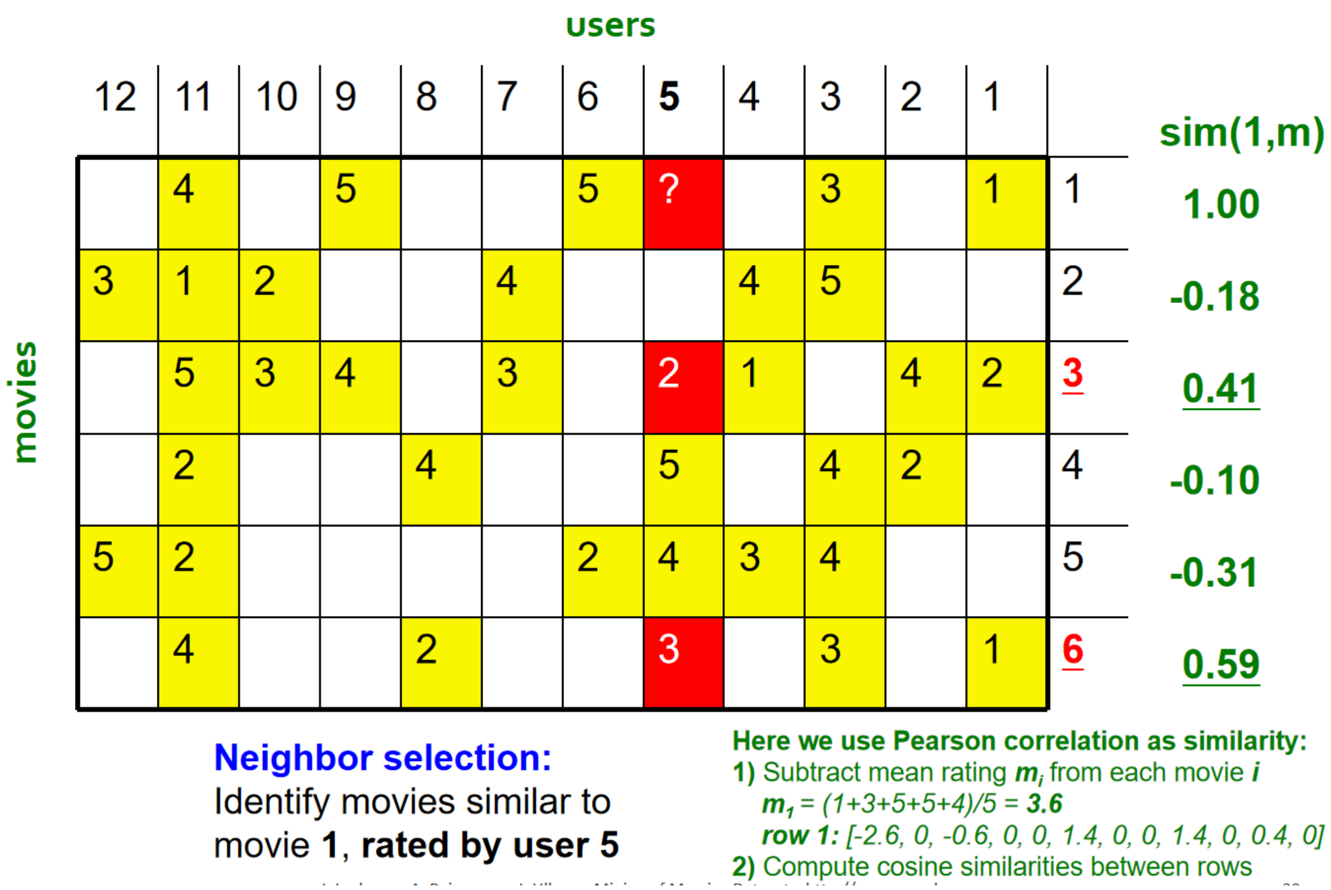
- user-user:



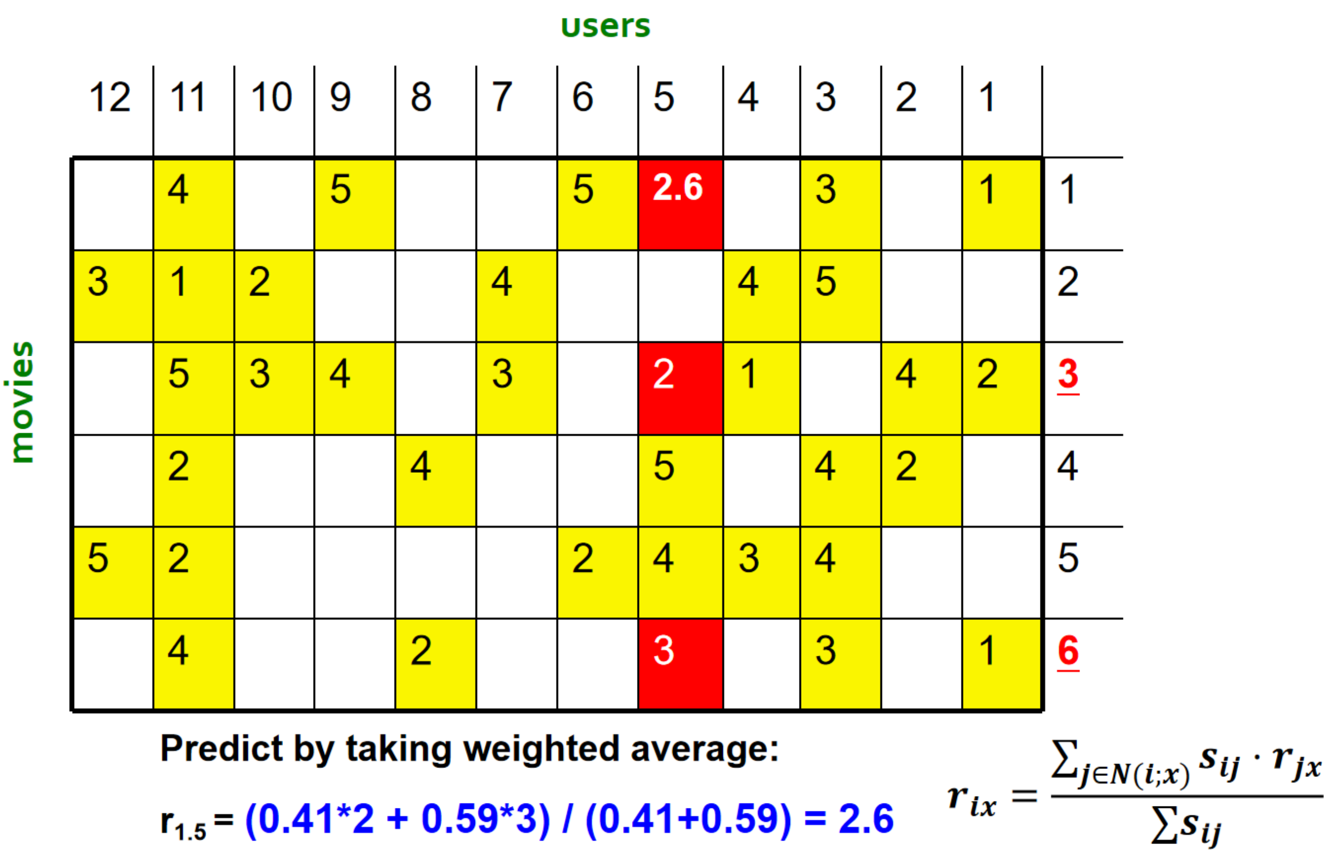
选择根据相似度加权来对评分做出预测：找和 $user$ 最像的 k 个，然后根据相似度加权平均这些其他 $user$ 的 $ratings$，作为对这个 $user$ 的预测

$$
r_{xi} = \frac{\sum_{y \in N} s_{xy} \cdot r_{yi}}{\sum_{y \in N} s_{xy}}
$$

- 一个形象的计算example：



12	11	10	9	8	7	6	5	4	3	2	1	movie	sim(1,m)
	4		5			5	?		3		1	1	1.00
3	1	2			4			4	5			2	-0.18
	5	3	4		3		2	1		4	2	3	0.41
	2			4			5		4	2		4	-0.10
5	2					2	4	3	4			5	-0.31
	4			2			3		3		1	6	0.59




```python
def recommend_top_n(user_id, utility_matrix, similarity_matrix, n, k):
    """
    Recommend top n items for the user.
    User-User CF.

    :param user_id: user id
    :param utility_matrix: utility matrix
    :param similarity_matrix: similarity matrix, Pearson similarity matrix here.
    :param n: number of items to recommend
    :param k: number of neighbors who is considered the most similar to the target user.
    :return: list of top n items to recommend
    """
    predicted_ratings = predict_for_target_id(user_id, 'user', utility_matrix, similarity_matrix, k)

    # deal with empty predicted ratings
    if predicted_ratings.empty:
        return []

    # remove items that the user has already seen
    user_seen = utility_matrix.loc[user_id].dropna().index
    predicted_ratings = predicted_ratings.drop(user_seen, errors='ignore')

    top_k_items = predicted_ratings.nlargest(n).index.tolist()
    return top_k_items
```

其中，`predict_for_target_id`如下：（写了`user_or_item='item'`的逻辑，但其实没用到，在下一个函数中实现了相似的部分。

```python
def predict_for_target_id(id, user_or_item, utility_matrix, similarity_matrix, k=200):
    """
    Predict the ratings for a item or for a user.
    User-User CF or Item-Item CF.

    :param user_id: user id or item id
    :param user_or_item: the id is for 'user' or 'item'
    :param utility_matrix: utility matrix
    :param similarity_matrix: similarity matrix.
    If "user", it is Pearson similarity matrix;
    if "item", it is adjusted cosine similarity matrix.
    :param k: number of neighbors who is considered the most similar to the target id.
    :return predicted ratings: a pandas.Series, predicted ratings for the target item or user
    """
    # drop the target itself
    similarities = similarity_matrix[id].drop(id)
    neighbors = similarities.nlargest(k).index
    neighbors_sim = similarities.loc[neighbors].values
    weights_sum = sum(neighbors_sim)

    # avoid division by zero
    if weights_sum == 0:
        return pd.Series(dtype='float64')  # return an empty Series: Series([], dtype: float64)

    if user_or_item == 'user':
        neighbors_ratings = utility_matrix.loc[neighbors]
        weighted_ratings = neighbors_ratings.mul(neighbors_sim, axis=0)
        predicted_ratings = weighted_ratings.sum(axis=0) / weights_sum
    elif user_or_item == 'item':
        neighbors_ratings = utility_matrix[neighbors]
        weighted_ratings = neighbors_ratings.mul(neighbors_sim, axis=1)
        predicted_ratings = weighted_ratings.sum(axis=1) / weights_sum
    return predicted_ratings
```

- item-item:

$$\boxed{r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}}$$

s_{ij}… similarity of items ***i*** and ***j***
r_{xj}…rating of user ***u*** on item ***j***
N(i;x)… set items rated by ***x*** similar to ***i***

  决定一个 $item\ i$ 是否要被推荐给 $user$，就去找被 $user$ 打过分的 $item$，找出其中和 $item\ i$ 最像的 k 个，也加权平均来做出预测。
- 只能给用户推荐到已经在 Utility Matrix 上出现过的电影。于是会拉低召回率。

```python
def item_based_recommend_top_n(user_id, utility_matrix, similarity_matrix, n, k):
    """
    Recommend top n items for the user.
    Item-Item CF.

    :param user_id: user id
    :param utility_matrix: utility matrix
    :param similarity_matrix: similarity matrix, adjusted cosine similarity matrix here.
    :param n: number of items to recommend
    :param k: number of neighbors who is considered the most similar to the target user.
    :return: list of top n items to recommend
    """
    # drop those movies rated by the user before (in the training set)
    user_rated = utility_matrix.loc[user_id].dropna().index
    candidate_items = utility_matrix.columns.difference(user_rated)

    predictions = {}

    for item_id in candidate_items:
        # Similar to part of 'predict_for_target_id' function
        similarities = similarity_matrix[item_id].drop(item_id, errors='ignore')

        # Tips: Only consider those items that the user has rated is enough.
        similarities = similarities.loc[similarities.index.intersection(user_rated)]

        if similarities.empty:
            continue

        neighbors = similarities.nlargest(k).index
        neighbors_sim = similarities.loc[neighbors].values
        weights_sum = sum(neighbors_sim)

        # avoid division by zero
        if weights_sum == 0:
            continue

        # neighbors_ratings_for_curr_user = utility_matrix.loc[user_id, neighbors].fillna(0)
        # since we use "intersection" above, no need for fillna(0) here
        neighbors_ratings_for_curr_user = utility_matrix.loc[user_id, neighbors]
        # weighted_ratings = neighbors_ratings_for_curr_user.mul(neighbors_sim, axis=1)
        weighted_ratings = neighbors_ratings_for_curr_user * neighbors_sim
        predicted_rating = weighted_ratings.sum() / weights_sum
        predictions[item_id] = predicted_rating

    if not predictions:
        return []

    pred_series = pd.Series(predictions)
    top_k_items = pred_series.nlargest(n).index.tolist()
    return top_k_items
```

More Details

- 用 `pandas.DataFrame` 比 `numpy.ndarray` 方便一点，因为能保留 Id 名
- 由于选用的数据集较小，而且在这样比较稀疏的评分数据上，衡量 movie-movie Similarity 更困难，所以实际表现中， user-user CF 的召回率比 item-item CF 更好。

Content-based

Profile

- 那么就需要分别构建 user 和 movie 的 profile

movie的profile：

用电影标签+ TFIDF 转换来构建。

- For each item, create an **item profile**
- **Profile is a set (vector) of features**
  - **Movies:** author, title, actor, director,…
  - **Text:** Set of "important" words in document
- **How to pick important features?**
  - Usual heuristic from text mining is **TF-IDF** (Term frequency * Inverse Doc Frequency)
    - **Term … Feature**
    - **Document … Item**

user的profile：

用评价过的 item profile 的加权平均来构建。

- **User profile possibilities:**
  - Weighted average of rated item profiles
  - **Variation:** weight by difference from average rating for item

代码实现

```python
def build_movie_profiles(movies_df):
    """
    Build movie profiles using TF-IDF.

    :param movies_df: pandas.DataFrame, movies data
    :return: movie profiles (TF-IDF matrix)
    """
    tfidf = TfidfVectorizer()
    tfidf_matrix = tfidf.fit_transform(movies_df['genres'].fillna(''))
    movies_profiles = pd.DataFrame(tfidf_matrix.toarray(), index=movies_df['movieId'])

    return movies_profiles

def content_based_top_n(user_id, utility_matrix, movie_profiles, n):
    """
    Content-based recommendation for top n items.

    :param user_id: user id
    :param utility_matrix: utility matrix
    :param movie_profiles: movie profiles (TF-IDF matrix)
    :param n: number of items to recommend
    :return: list of top n items to recommend
    """
    user_ratings = utility_matrix.loc[user_id].dropna()
    if user_ratings.empty:
        return []

    # build user profile
    rated_profiles = movie_profiles.loc[user_ratings.index]
    user_profile = np.dot(user_ratings.values, rated_profiles.values) / user_ratings.sum()  # numPy 1-D array

    # calculate cosine similarity between user profile and movie profiles
    # [user_profile]: to make it 2-D array
    # cosine_similarity(A, B) output: A.shape[0] x B.shape[0]
    # [0]: to get the only one row of the result
    similarities = cosine_similarity([user_profile], movie_profiles.values)[0]
    sim_series = pd.Series(similarities, index=movie_profiles.index)
    # remove items that the user has already seen
    sim_series = sim_series.drop(user_ratings.index, errors='ignore')
    top_k_items = sim_series.nlargest(n).index.tolist()
    return top_k_items
```

Prediction

- 直接用 Cosine Similarity 来衡量相似性：

  - **Prediction heuristic:**
    - Given user profile $\boldsymbol{x}$ and item profile $\boldsymbol{i}$, estimate

$$u(\boldsymbol{x}, \boldsymbol{i}) = \cos(\boldsymbol{x}, \boldsymbol{i}) = \frac{\boldsymbol{x} \cdot \boldsymbol{i}}{||\boldsymbol{x}|| \cdot ||\boldsymbol{i}||}$$

  代码如上。

召回率

- 单用户，Top-N 推荐召回率：

$$\mathrm{Recall@N}_u = \frac{|R_u^{(N)} \cap T_u|}{|T_u|}$$

- $R_u^{(N)}$：给用户 u 推荐的 Top-N 项物品；
- T_u：用户在测试集中的真实兴趣物品集合；
- $|R_u^{(N)} \cap T_u|$：推荐命中的物品个数；
- $|T_u|$：用户实际喜欢的物品数。

- 所有用户的平均 Recall@N

$$\mathrm{Recall@N} = \frac{1}{|U|}\sum_{u \in U} \mathrm{Recall@N}_u$$

实际代码实现直接加在一起。

```
def recall_n(test_df, recommendations):
    """
    Calculate recall@N.

    :param test_df: pandas.DataFrame, test data
    :param recommendations: dictionary, user_id -> recommended items (list of item_ids)
    :return: recall@N score
    """
    hit = 0
    total = 0
    # group by userId, get the set of movies watched by each user
    grouped = test_df.groupby('userId')['movieId'].apply(set)

    for user_id, watched_movies in grouped.items():
        recommended_movies = recommendations.get(user_id, [])
        hit += len(set(recommended_movies) & watched_movies)
        total += len(watched_movies)

    if total == 0:
        return 0
    else:
        return hit / total
```

设计加权混合策略推荐多样性

- 加权使用 user-user CF 和 content-based 得到的ratings来做预测（实验中就取0.5加权）：

```python
def hybrid_top_n(user_id, utility_matrix, user_similarity_matrix, movie_profiles, n, k, alpha=0.5):
    """
    Hybrid recommendation for top n items.

    :param user_id: user id
    :param utility_matrix: utility matrix
    :param user_similarity_matrix: user similarity matrix (Pearson similarity matrix)
    :param movie_profiles: movie profiles (TF-IDF matrix)
    :param n: number of items to recommend
    :param k: number of neighbors who is considered the most similar to the target user.
    :param alpha: weight for CF and content-based recommendation
    :return: list of top n items to recommend
    """
    cf_scores = predict_for_target_id(user_id, 'user', utility_matrix, user_similarity_matrix, k)

    # deal with empty predicted ratings
    if cf_scores.empty:
        return []

    # remove items that the user has already seen
    user_seen = utility_matrix.loc[user_id].dropna().index
    cf_scores = cf_scores.drop(user_seen, errors='ignore')

    # build user profile using content-based method
    user_ratings = utility_matrix.loc[user_id].dropna()
    rated_profiles = movie_profiles.loc[user_ratings.index]
    user_profile = np.dot(user_ratings.values, rated_profiles.values) / user_ratings.sum()

    # calculate cosine similarity between user profile and movie profiles
    similarities = cosine_similarity([user_profile], movie_profiles.values)[0]
    sim_series = pd.Series(similarities, index=movie_profiles.index)

    # remove items that the user has already seen
    sim_series = sim_series.drop(user_seen, errors='ignore')

    # combine CF and content-based scores using alpha parameter
    combined_scores = alpha * cf_scores.add((1 - alpha) * sim_series, fill_value=0)

    top_k_items = combined_scores.nlargest(n).index.tolist()

    return top_k_items
```

以上4种方法的召回率结果比较

- 考虑到 item-item CF 耗时较长，所以对 item-item CF 测试时，划分较小的测试集（`test_size=0.001`）
- 事实上，平均每个用户评分过的电影并不多，中位数在100左右，但也有评分了1000、2000部电影的用户。
- 召回率大致会随着 N 增大而增大，但当然 N 选太大并不合适，于是下面分别比较 N=200 和 N=1000（由于电影数据较稀疏，所以 item-item CF 和 Content-Based 在 N 不大时召回率表现并不好）。
- `test_size=0.001`，推荐 top-N 的 $N = 200$:

```
User-User CF Recall: 0.3564
100%|██████████████████████████████
Item-Item CF Recall: 0.0099
Content-based Recall: 0.0495
Hybrid Recall: 0.3861
```

- `test_size=0.001`，推荐 top-N 的 $N = 1000$:

```
User-User CF Recall: 0.6238
100%|██████████████████████████████
Item-Item CF Recall: 0.0792
Content-based Recall: 0.1881
Hybrid Recall: 0.5941
```

- `test_size=0.2`，推荐 top-N 的 $N = 200$：

  ```
  User-User CF Recall: 0.3101
  Content-based Recall: 0.0661
  Hybrid Recall: 0.3201
  ```

结果分析

如上，发现：

- User-User CF 平均地表现最好：
  - 在较真实的数据划分（$\text{test\_size} = 0.2$，且就只从 9000 个电影中推荐 200 个）中 Recall 有 0.31 。
  - 而如果推荐数量很大、或者是有更大比例的训练数据集，那么召回率表现能更好（0.35、0.62）。
- 可见**对于 `MovieLens` 数据集，User-User CF 提取出的相似性信息比其他方法更有效**。
- Item-Item CF 表现不好（仅 0.0099 和 0.0792）
  - 用户评分行为较分散，电影间共同评分用户很稀疏，导致相似度不可靠。
- Content-Based 表现一般般，但也是合理的：
  - 只能根据构建的特征profile来进行相似性推荐，无法发现用户的潜在兴趣，所以确实不如 User-User CF
  - 但是利用电影标签进行了特征profile构建，所以也能达到一定的效果。
- Hybrid Method 综合了 User-User CF 和 Content-Based：
  - 能有较好的召回率
  - 因为结合了 Content-Based，所以能在冷启动、数据不完整（比Item-Item CF表现更好）、推荐多样性方面具有优势
  - 较稳定