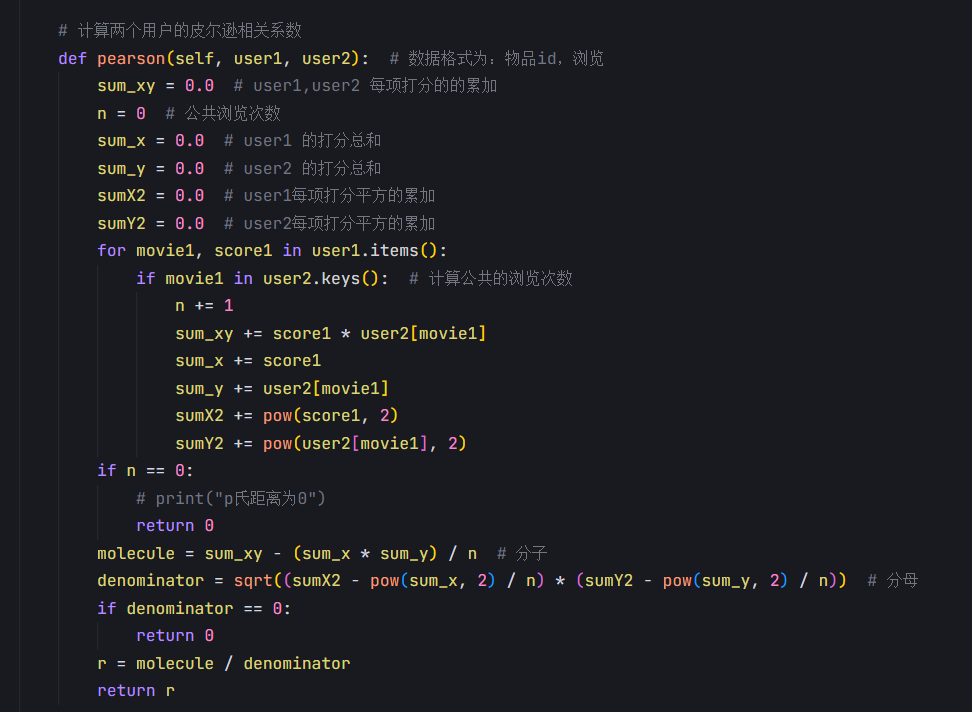
1. 基于用户的协同过滤 (User-Based Collaborative Filtering) ：

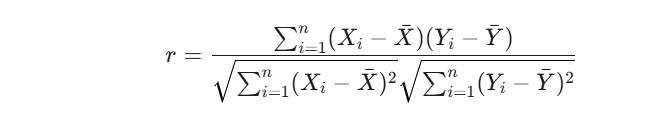
- 由 UserCf 类实现。

- pearson 方法计算两个用户之间的皮尔逊相关系数，用于衡量用户相似度。



### 皮尔逊相关系数的计算公式

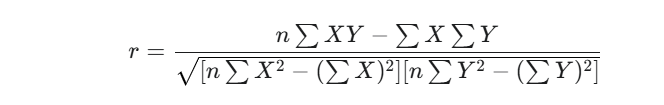
皮尔逊相关系数的公式为



其中：

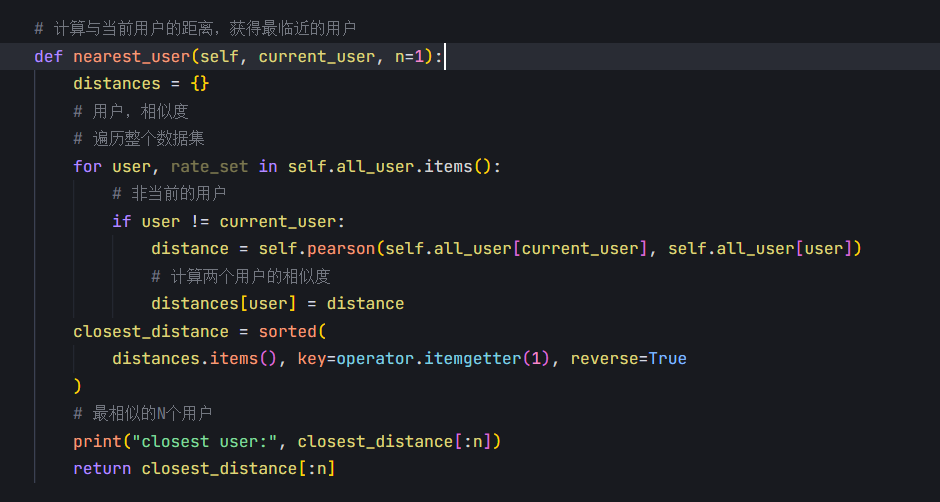
* n 是两个用户共同评分的物品数量。
* Xi​ 是用户1对第 i 个物品的评分。
* Yi​ 是用户2对第 i 个物品的评分。
* Xˉ 是用户1对所有共同评分物品的平均分。
* Yˉ 是用户2对所有共同评分物品的平均分。

公式通常可以简化为：



代码中使用的就是这个简化后的公式。

- nearest\_user 方法找到与当前用户最相似的 N 个用户。



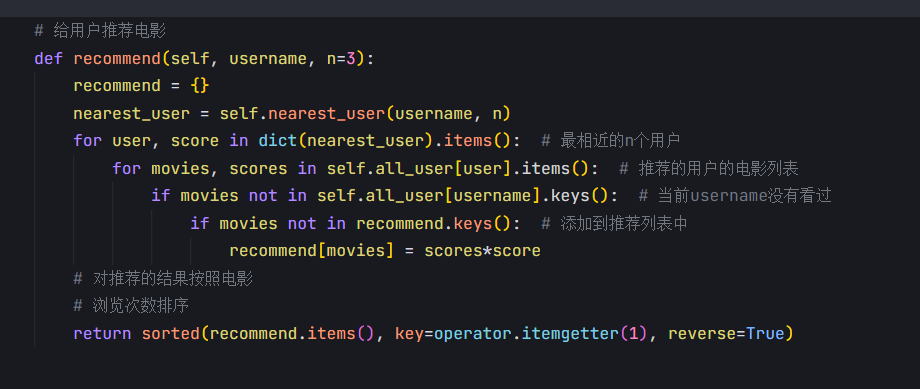
**计算相似度：**

* distance = self.pearson(self.all\_user[current\_user], self.all\_user[user])
  + 这是核心步骤。它调用了之前解释的 pearson 方法来计算 current\_user 和 user 之间的皮尔逊相关系数。
  + self.all\_user[current\_user] 传入了当前用户的评分数据。
  + self.all\_user[user] 传入了正在比较的另一个用户的评分数据。
  + 计算结果 distance（皮尔逊相关系数）会被赋值给 distance 变量。

**排序并选择最相似的 N 个用户：**

* closest\_distance = sorted(distances.items(), key=operator.itemgetter(1), reverse=True)：
  + distances.items() 将字典转换为一个包含 (键, 值) 对的列表，例如 [('userA', 0.8), ('userB', 0.5), ...]。
  + sorted() 函数对这个列表进行排序。
  + key=operator.itemgetter(1) 指定了排序的依据是每个元组的第二个元素，也就是相似度值。
  + reverse=True 表示进行降序排序，这样相似度最高的用户会排在列表的前面。
* 排序后的 closest\_distance 列表现在包含了所有其他用户及其与 current\_user 的相似度，并且是按相似度从高到低排列的。

### - recommend 方法根据相似用户的评分来推荐电影给当前用户。



· **找到相似用户（Find Nearest Neighbors）：**

* · nearest\_user = self.nearest\_user(username, n)：这是推荐的第一步。它利用之前解释的 nearest\_user 方法，找出与目标用户 username 口味最相似的 n 个用户。这些相似用户是后续推荐的依据。

· **遍历相似用户的电影列表：**

* · for user, score in dict(nearest\_user).items():：循环遍历这些找到的相似用户。
  + user：是相似用户的 ID。
  + score：是该相似用户与 username 之间的皮尔逊相似度（0 到 1 之间）。

· **筛选未观看的电影：**

* · if movies not in self.all\_user[username].keys():：对于相似用户评分过的每一部电影，系统会检查目标用户 username 是否已经观看过这部电影。
* **目的：** 推荐系统只应该推荐用户尚未接触过（未评分/未浏览）的物品。

· **计算加权评分：**

* · recommend[movies] = scores \* score 或 recommend[movies] += scores \* score：
* 这是用户协同过滤中预测评分的关键部分。它使用**加权平均**的思想：
  + scores 是相似用户对电影的原始评分。
  + score 是相似用户与目标用户之间的相似度。
  + 将这两者相乘 (scores \* score)，意味着：
    - **相似度越高的用户**对电影的评分，其对最终推荐列表的**贡献越大**。
    - **相似度越低的用户**对电影的评分，其对最终推荐列表的**贡献越小**。
  + 如果多于一个相似用户评分了同一部电影，这些加权评分会被累加起来，形成该电影的总加权评分。这反映了多位相似用户对该电影的集体偏好强度。

· **排序并返回推荐列表：**

* · return sorted(recommend.items(), key=operator.itemgetter(1), reverse=True)：
* 最后，将 recommend 字典中的电影按照它们的总加权评分进行降序排序。加权评分越高的电影，被认为用户 username 越可能喜欢，因此会排在推荐列表的前面。

- recommend\_by\_user\_id(user\_id) 函数是外部调用的接口，它会获取用户的评分数据，构建用户-电影评分矩阵，然后调用 UserCf 类进行推荐。如果用户没有评分，会根据用户偏好标签或电影热度进行推荐。



用户身份识别与初始数据准备：

函数首先获取当前用户 user\_id 的基本信息和其历史偏好标签。

这是一个很重要的冷启动（Cold Start）处理策略。

新用户/冷启动处理：

判断： if current\_user.rate\_set.count() == 0: 这一步是关键。它检查用户是否有任何评分记录。

策略一（基于标签）： 如果用户没有评分但设置了偏好标签 (len(user\_prefer) != 0)，系统会根据这些标签直接推荐相关电影。这是一种内容推荐（Content-Based Recommendation）的简单形式，用于解决新用户没有历史行为数据的问题。

策略二（基于热度）： 如果用户既没有评分也没有设置标签，那么系统会退化为推荐热门电影 (Movie.objects.order\_by("-num"))。这是最简单的冷启动策略，确保任何新用户都能获得推荐。

协同过滤数据准备（针对有评分的用户）：

数据稀疏性问题： 在实际应用中，很多用户可能只评分了少数电影，导致用户-电影评分矩阵非常稀疏。直接对所有用户计算相似度可能会效率低下且效果不佳。

优化：选择评分最多的用户： 代码通过 users\_rate = Rate.objects.values('user').annotate(mark\_num=Count('user')).order\_by('-mark\_num') 这一步，只选择评分数量最多的10个用户来构建协同过滤的上下文。这样可以减少计算量，并专注于那些提供更多信息的活跃用户。

包含当前用户： user\_ids.append(user\_id) 确保当前用户的数据也包含在内，即使他不在评分最多的10个用户之列，这样 UserCf 类才能计算他与“邻居”的相似度。

构建 all\_user 矩阵： 将这些用户的评分数据转换为 UserCf 类所需的 all\_user 字典格式（{用户名: {电影ID: 评分, ...}}）。这是将数据库数据转换为算法可处理格式的关键步骤。

执行用户协同过滤推荐：

user\_cf = UserCf(all\_user=all\_user)：实例化推荐引擎。

recommend\_list = [each[0] for each in user\_cf.recommend(current\_user.username, 15)]：调用 UserCf 实例的 recommend 方法，传入当前用户的用户名和期望的推荐数量（15部）。这一步正是基于用户相似度计算并生成推荐列表。

推荐结果处理与补充：

movie\_list = list(Movie.objects.filter(id\_\_in=recommend\_list).order\_by("-num")[:15])：根据协同过滤得到的电影ID，从数据库中获取完整的电影对象。这里再次根据 num（热度）进行排序和截取，可能是在协同过滤推荐结果数量多于15时的一种优化或兜底。

推荐数量不足补充： if other\_length > 0: 部分是又一个重要的兜底策略。如果协同过滤未能推荐出足够的（例如15部）电影，它会从用户未观看过的电影中，按照\*\*收藏量（或热度）\*\*补充剩余的数量，直到达到15部。这确保了推荐列表始终是完整和有用的。~Q(rate\_\_user\_id=user\_id) 确保补充的电影是用户没看过的。

## 基于物品的协同过滤 (Item-Based Collaborative Filtering) ：

基于物品的协同过滤与基于用户的协同过滤不同，它关注的是物品之间的相似性，而不是用户之间的相似性。它的核心思想是：**如果用户喜欢某部电影，那么他也可能喜欢与这部电影相似的其他电影。**



similarity 函数使用逻辑：

* **分子 (**common**)**：代表了同时给 movie1 和 movie2 评分过的用户数量。可以看作是两个电影“共同拥有”的评分用户的数量。
* **分母 (**sqrt(movie1\_sum \* movie2\_sum)**)**：movie1\_sum 是给 movie1 打分的用户总数，movie2\_sum 是给 movie2 打分的用户总数。分母是这两个数量乘积的平方根。

主要关注的是**有多少共同的用户对这两部电影感兴趣**

* similarity(movie1\_id, movie2\_id) 函数计算两部电影之间的相似度，这里使用的是余弦相似度，通过计算共同评分用户的数量来衡量。
* recommend\_by\_item\_id(user\_id, k=15) 函数是外部调用的接口，它会找到用户未看过的电影中，与用户已评分电影相似度最高的电影进行推荐。同样，如果用户没有评分，会根据用户偏好标签或电影热度进行推荐。

总的步骤是：

- 数据准备 ：从 Django 的 Rate 和 UserTagPrefer 模型中获取用户的评分数据和标签偏好数据。

- 用户协同过滤 ：

- 找到与目标用户有共同评分电影的其他用户。

- 使用皮尔逊相关系数计算目标用户与其他用户的相似度。

- 选择最相似的 N 个用户。

- 根据这些相似用户对目标用户未看过的电影的评分，进行加权平均，预测目标用户对这些电影的评分，并推荐评分最高的电影。

- 物品协同过滤 ：

- 计算电影之间的相似度（基于共同评分用户的余弦相似度）。

- 对于目标用户已评分的电影，找到与这些电影相似度高的、但用户尚未看过的电影。

- 结合用户对已看电影的评分和电影间的相似度，预测用户对未看电影的喜好程度，并推荐预测评分最高的电影。

- 冷启动处理 ：对于没有评分记录的用户，系统会根据其选择的标签偏好或者电影的浏览量（热度）来提供初始推荐。