面试让你手写ItemCF/UserCF代码，你会吗?

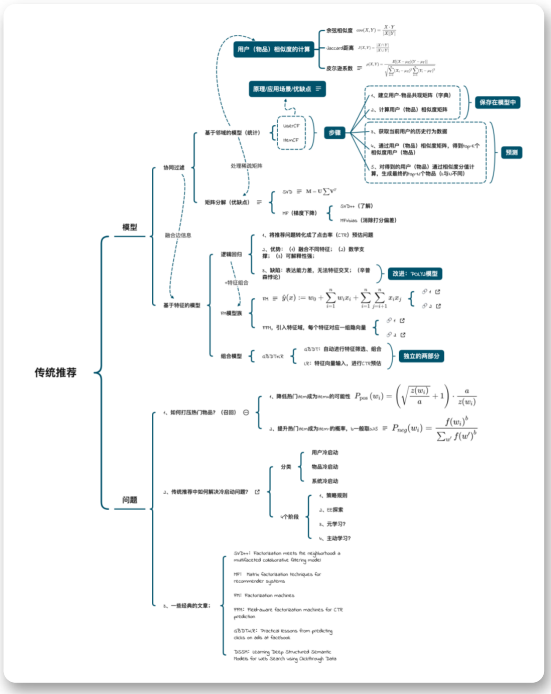
原创 潜心 推荐算法的小齿轮 1周前

收录于话题
#推荐系统

5个

前言

之前朋友说有同学在面字节算法实习时让复现DeepFM算法（包括训练），然后就懵了。因此最近在整理传统推荐算法的一些内容时，大概是这样的：



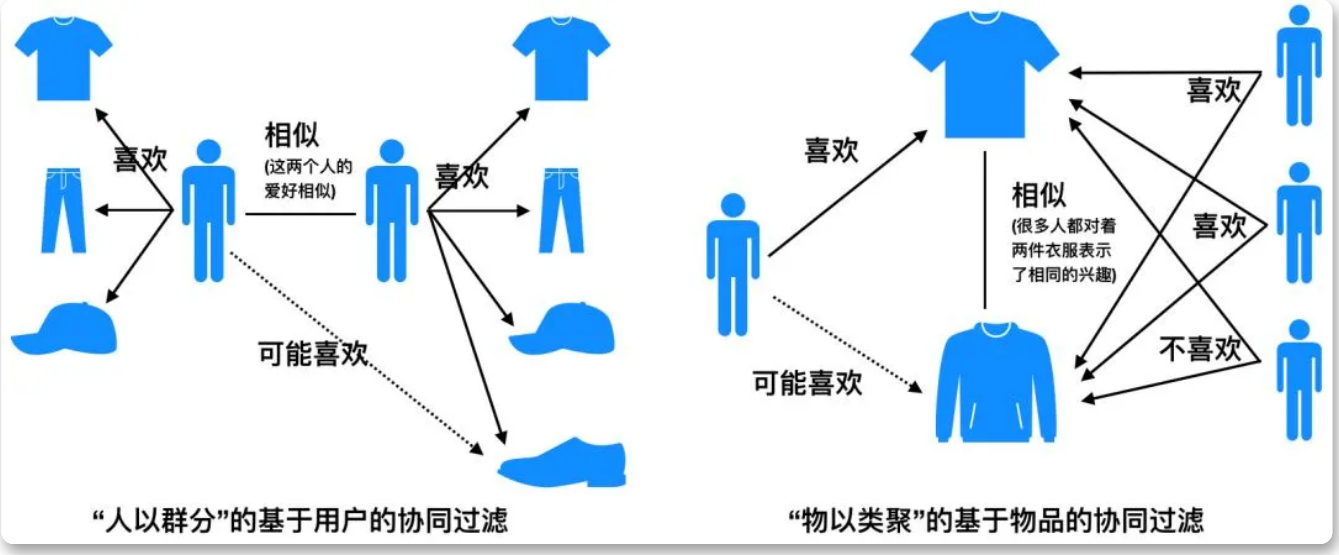
就想到**「基于邻域的协同过滤（UserCF与ItemCF），除了了解原理、应用场景的区别外，如果现场写实现伪代码你会么？」** 有很多文章在讲协同过滤的原理，很少具体到代码，所以这次把以前写的CF代码优化下进行分享。

本文约1.7k字，预计阅读5分钟.

概要

协同过滤是**「基于用户行为」**设计的推荐算法，具体来说，是「通过群体的行为来找到某种相似性」（用户之间的相似性或者物品之间的相似性），通过相似性来为用户做决策和推荐。当

然，协同过滤有基于邻域的、隐语义模型等，这里我们主要指的是基于邻域的ItemCF和UserCF。具体的原理就不多做解释了，见下图：



ItemCF与UserCF的代码其实很相近，因此以ItemCF为例，两个算法的具体代码可以在github：https://github.com/ZiyaoGeng/SimpleCF（阅读原文）中找到。

ItemCF

对于ItemCF算法的实践，我采用类的方式进行建立，主要内容包括：

1. 模型的输入；
2. 初始化【计算相似度矩阵】；
3. 对单个用户进行推荐；
4. 对测试集的所有内容进行推荐；

因此大体框架为：

```
class ItemCF:
    def __init__(self, user_item_dict, item_hot_list, sim_item_topK, topN, i2i_sim=None):
        """
        Item-based collaborative filtering
        :param user_item_dict: A dict. {user1: [(item1, score),...], user2: ...}
        :param item_hot_list: A list. The popular movies list.
        :param sim_item_topK: A scalar. Choose topK items for calculate.
        :param topN: A scalar. The number of recommender list.
        :param i2i_sim: dict. If None, the model should calculate similarity matrix.
        """

    def __get_item_sim(self):
        """
        calculate item similarity weight matrix
        :return: i2i_sim
        """
```

```
def recommend(self, user_id):

def recommend_all(self, test):
```

输入

- `user_item_dict`：首先最重要的是建立 `user-item` 共现矩阵作为**「输入」**，当然在实际应用中，我们采用字典的形式减少不必要的空间。对于ItemCF，需要 `user-item-dict` 字典，而对于UserCF，需要 `user-item-dict` 和 `item-user-dict` 两个字典：

$$\text{item-user-dict: } \{user1: [(item1, score), \dots], user2: [\dots], \dots\}$$
$$\text{user-item-dict: } \{item1: [(user1, score), \dots], item2: [\dots], \dots\}$$

【注】元组部分score可以去除或者替换为其他内容；

- `item_hot_list`：热门物品的列表，主要是当推荐物品数量不够时（冷启动等），用热门物品进行填充，计算也比较简单；

- `sim_item_topK`：选取某个物品最相似的TopK个物品，不然选择所有物品会产生很大的计算量；

- `topN`：推荐列表的大小；

- `i2i_sim`：物品相似度矩阵。一般计算相似度矩阵后会在本地进行保存，因此如果之前计算过，则只需读取，不用重复计算；

物品相似度矩阵

`ItemCF` 算法认为**「物品A和物品B具有很大的相似度是因为喜欢物品A的用户也大多喜欢物品B」**，因此需要计算物品相似度矩阵，主要分为两步：

1. 统计两两物品之间的共现次数，即**「用户同时喜欢两个物品」**；
2. 通过Jaccard距离、余弦相似度等方式计算两个物品的相似性；

当然对于1来说，需要对于活跃的用户进行惩罚，通过增加IUF（Inverse User Frequence），用户活跃度对数倒数的参数，对应代码中：

```
i2i_sim[i][j] += 1 / math.log(len(items) + 1)
```

在2中，通过余弦相似度的计算可以降低热门物品会和很多物品相似的可能性，因为基于物品的推荐主要是挖掘长尾信息。

```
i2i_sim[i][j] = wij / math.sqrt(item_cnt[i] * item_cnt[j])
```

具体代码如下：

```
def __get_item_sim(self):
    """
    calculate item similarity weight matrix
    :return: i2i_sim
    """
    i2i_sim = dict()
    item_cnt = defaultdict(int)  # Count the number of visits to the item
    for user, items in tqdm(self.user_item_dict.items()):
        for i, score_i in items:
            item_cnt[i] += 1
            i2i_sim.setdefault(i, {})
            for j, score_j in items:
                if i == j:
                    continue
                i2i_sim[i].setdefault(j, 0)
                i2i_sim[i][j] += 1 / math.log(len(items) + 1)
    for i, related_items in i2i_sim.items():
        for j, wij in related_items.items():
            i2i_sim[i][j] = wij / math.sqrt(item_cnt[i] * item_cnt[j])
    return i2i_sim
```

推荐

使用之前计算的相似度矩阵对每个用户进行推荐。主要分为两步：

1. 获取推荐用户的历史行为，在相似度矩阵中选取每个历史物品（遍历）最相似的 `topk` 个物品来计算每个物品（未出现在历史行为中）的**「累积权重」**；
2. 若1中所有物品数量小于推荐列表，则采用其他策略进行填充，如选择热门物品，但权重分数是最小的，可以为负数。然后对权重列表进行排序，选取权重分数最高的 `TopN个物品` ；

具体代码如下：

```
def recommend(self, user_id):
    """
    recommend one user
    :param user_id: user's ID
    :return:
    """
    item_rank = dict()
    user_hist_items = self.user_item_dict[user_id]
    for i, score_i in user_hist_items:
        for j, wij in sorted(self.i2i_sim[i].items(), key=lambda x: x[1], reverse=True)[:self.s
            if j in user_hist_items:
                continue

            item_rank.setdefault(j, 0)
            item_rank[j] += 1 * wij

    if len(item_rank) < self.topN:
        for i, item in enumerate(self.item_hot_list):
            if item in item_rank:
                continue
            item_rank[item] = - i - 1  # rank score < 0
            if len(item_rank) == self.topN:
                break
    item_rank = sorted(item_rank.items(), key=lambda x: x[1], reverse=True)[:self.topN]

    return [i for i, score in item_rank]
```

【注】在计算每个物品的权重时，选择了 `1 * wij`，并没有用到分数和其他信息。

评估

「数据集」：ml-1m，当然也可以选择其它。预处理主要包括：

1. 选择评分大于x的作为正样本（当然可以直接将所有交互过的物品都作为正样本，将评分融入推荐时权重分数的计算）；
2. 选取每个用户的最后一次发生交互的物品作为测试集（所以评估的结果会比项亮《推荐系统实战》的结果差很多）；

3. 建立 user-item 和 item-user 字典；

「评估指标」：

HR（点击率）与NDCG，当每个用户真实的交互物品只有一个时，HR等价于Recall；

评估代码如下：

```
def evaluate_model(model, test):
    """
    evaluate model
    :param model: model of CF
    :param test: dict.
    :return: hit rate, ndcg
    """
    hit, ndcg = 0, 0
    for user_id, item_id in tqdm(test.items()):
        item_rank = model.recommend(user_id)
        if item_id in item_rank:
            hit += 1
            ndcg += 1 / np.log2(item_rank.index(item_id) + 2)
    return hit / len(test), ndcg / len(test)
```

实验

对于ItemCF算法，超参数：

- sim_item_topK = 20；
- topN = 20；

使用自己的笔记本，结果如下：

数据读取：2s；

模型建立（计算相似度矩阵）：3min52s；

模型评估：19min47s，HR = 0.065232，NDCG = 0.024265；

总结

之前参考《推荐算法实践》进行过复现，然后这次结合了Datawhale中新闻推荐的Baseline进行优化，确实对算法中如何惩罚热门物品与活跃用户有了更深的认识。所有代码在github“https://github.com/ZiyaoGeng/SimpleCF”中，如果存在一些Bug或者更好的优化（评估时间太长），欢迎提出Issues。

阅读原文