南開大學
Nankai University

计算机学院

大数据计算与应用实验报告

推荐算法的实现及优化

姓名：齐明杰 李佳豪 纪潇洋

专业：信息安全

2023 年 6 月 8 日

1 实验目的

- 基于深度学习，借助 SVD 算法实现协同过滤的推荐算法
- 利用训练得到的权重预测 Test.txt 文件中 (u, i) 对的评分分数

2 数据集的处理

2.1 数据集介绍

- train.txt，包含不同用户对不同物品的打分数据，每个用户所评价的物品数各不相同，用于后续训练模型。
- test.txt，包含待评分的用户序号以及其需要评价的 6 个物品编号，用于测试所训练的模型。
- ItemAttribute.txt，包含了每个物品的两个属性值 (不是所有的物品都具有两个属性值，因此需要预处理)，可以通过这两个属性值计算物品之间的相似度，更加科学地进行基于物品的协同过滤，更加精准地训练模型。
- ResultForm.txt，保存预测结果的文档，将 Test.txt 文件中待测试物品评分输出到该文档中提交。

2.2 数据预处理

2.2.1 建立物品下标的有序映射

由于物品的序号不连续，需要建立有序的映射。

首先定义一个空集合 all_nodes，用于存储所有的节点; 打开 train_path 文件，并循环读取每一行内容 (使用 while ... readline() 实现)。每行内容包括两个部分：节点数量和节点关联信息; 对于每一行，首先使用 strip().split('|') 方法切分出节点数量和节点关联信息，并将节点数量转化为整数类型。并提取其中的节点 ID，将其加入到 all_nodes 集合中; 完成所有行的读取后，使用 sort 方法对 all_nodes 中的节点从小到大排序，为每个节点指定一个唯一的索引。

```python
def get_idx(train_path):
    all_nodes = set()
    with open(train_path, 'r') as f:
        while (line := f.readline()) != '':
            _, num = map(int, line.strip().split('|'))
            for _ in range(num):
                line = f.readline()
                item_id, _ = map(int, line.strip().split())
                all_nodes.add(item_id)
    node_idx = {node: idx for idx, node in enumerate(sorted(all_nodes))}
    return node_idx
```

2.2.2 处理训练集和测试集

具体分为以下几个步骤：

1. get_train_data 函数：读取训练集文件 train.txt，并将数据按照用户为键和物品为键分别存储到字典 data_user 和 data_item 中。在处理每条记录时，先读取用户 ID 和该用户评价的物品数量，再读取该用户评价的物品及其评分，对应地将物品 ID 和评分存储到 data_user 和 data_item 中。由于原始数据中物品 ID 不是连续的数字，因此需要一个 node_idx 字典来映射原始 ID 与连续 ID 之间的关系。

2. get_attribute_data 函数：读取物品属性文件 itemAttribute.txt，并将数据按照物品 ID 存储到字典 attrs 中。在处理每条记录时，先读取物品 ID 及其两个属性值 attr1 和 attr2，将它们转换成 0/1 格式后，将物品 ID 和属性列表存储到 attrs 中。只有在 node idx 中出现的物品才会被存储，也就是说只有这些物品有相关的属性信息。

3. get_data_by_attr 函数：根据物品属性来划分数据。这个函数接受两个参数，第一个是之前得到的 data user 字典，第二个是之前得到的 attrs 字典。返回两份数据，第一份是有属性 1 的，第二份是有属性 2 的。这里将属性值为 1 和 2 的物品对应的评价数据分别提取出来，存储到 data1 和 data2 字典中。

4. get_test_data 函数：读取测试集文件 test.txt，并按照用户 ID 存储到字典 data 中。在处理每条记录时，先读取用户 ID 及其评价的物品数量，再读取该用户评价的物品 ID，将这些物品 ID 存储到 data 中。

5. split_data 函数：将训练集数据划分为训练集和验证集。这个函数接受两个参数，第一个是之前得到的 data user 字典，第二个是可选参数 ratio，表示训练集所占的比例，默认是 0.85。返回两份数据，第一份是训练集，第二份是验证集。

上述几个函数代码如下所示：

```
def get_train_data(train_path, node_idx):
    data_user, data_item = defaultdict(list), defaultdict(list)
    with open(train_path, 'r') as f:
        while (line := f.readline()) != '':
            user_id, num = map(int, line.strip().split('|'))
            for _ in range(num):
                line = f.readline()
                item_id, score = line.strip().split()
                item_id, score = int(item_id), float(score)
                # 把0-100的得分映射到0-10
                score = score / 10
                data_user[user_id].append([node_idx[item_id], score])
                data_item[node_idx[item_id]].append([user_id, score])
    return data_user, data_item


```

```python
def get_attribute_data(attribute_path, node_idx):
    attrs = defaultdict(list)
    with open(attribute_path, 'r') as f:
        while (line := f.readline()) != '':
            item_id, attr1, attr2 = line.strip().split('|')
            attr1 = 0 if attr1 == 'None' else 1
            attr2 = 0 if attr2 == 'None' else 1
            item_id = int(item_id)
            if item_id in node_idx:
                attrs[node_idx[item_id]].extend([attr1, attr2])
    return attrs
def get_data_by_attr(data_user, attrs):
    # 返回两份数据，第一份是有属性1的，第二份是有属性2的
    data1, data2 = defaultdict(list), defaultdict(list)
    for user_id, items in data_user.items():
        for item_id, score in items:
            if attrs[item_id]:
                if attrs[item_id][0] == 1:
                    data1[user_id].append([item_id, score])
                if attrs[item_id][1] == 1:
                    data2[user_id].append([item_id, score])
    return data1, data2
def get_test_data(test_path):
    data = defaultdict(list)
    with open(test_path, 'r') as f:
        while (line := f.readline()) != '':
            user_id, num = map(int, line.strip().split('|'))
            for _ in range(num):
                line = f.readline()
                item_id = int(line.strip())
                data[user_id].append(item_id)
    return data

def split_data(data_user, ratio=0.85, shuffle=True):
    train_data, valid_data = defaultdict(list), defaultdict(list)
    for user_id, items in data_user.items():
        if shuffle:
            np.random.shuffle(items)
        train_data[user_id] = items[:int(len(items) * ratio)]
        valid_data[user_id] = items[int(len(items) * ratio):]
    return train_data, valid_data
```

2.2.3 统计数据特征

由于训练过程中需要用到全评分均值，用户偏差，物品偏差，因此需要对处理好的数据进行数据统计，方便后续进行偏差的计算。我们通过以下代码读取 train.txt 中的用户 ID，物品 ID，统计出 ID 的数量，最大 ID 等信息。

```
with open(file_path, 'r') as f:
      for line in f:
          if '|' in line:
              user_id, num_ratings = line.strip().split('|')
              user_id = int(user_id)
              user_set.add(user_id)
              rating_count += int(num_ratings)

              if user_id > max_user_id:
                  max_user_id = user_id
          else:
              item_id, _ = line.strip().split()
              item_id = int(item_id)
              item_set.add(item_id)

              if item_id > max_item_id:
                  max_item_id = item_id
```

最终得到如下结果：

- Number of users: 19835
- Number of rated items: 455705
- Number of ratings: 5001507
- Max user ID: 19834
- Max item ID: 624960

2.2.4 获取均值与偏差

- 计算全局评分均值 miu: 遍历 train_data_user，对每个用户的评分进行累加 sum，将 sum 加到全局评分均值 miu 中，将 miu 除以评分总数 ratings_num，得到全局评分均值 miu
- 计算用户偏差 b_x: 遍历 train_data_user，对每个用户的评分进行累加 sum，将 sum 除以该用户的评分总数 len(train_data_user[user_id])，得到该用户的平均评分，将该用户的平均评分 b_x[user_id] 减去全局评分均值 miu，得到该用户的偏差 b_x
- 计算物品偏差 b_i: 遍历 train_data_item，对每个物品的评分进行累加 sum，将 sum 除以该物品的评分总数 len(train_data_item[item_id])，得到该物品的平均评分，将该物品的平均评分

bi[item_id] 减去全局评分均值 miu，得到该物品的偏差 b_i 返回三个变量 miu、b_x 和 b_i，用于后续的评分预测

```python
def get_bias(train_data_user, train_data_item):
    """
    :param train_data_user: 用户-[物品，评分]字典
    :param train_data_item: 物品-[用户，评分]字典
    :return: 全评分均值，用户偏差，物品偏差
    """
    miu = 0.0
    bx = np.zeros(user_num, dtype=np.float64)
    bi = np.zeros(item_num, dtype=np.float64)
    for user_id in train_data_user:
        sum = 0.0
        for item_id, score in train_data_user[user_id]:
            miu += score
            sum += score
        bx[user_id] = sum / len(train_data_user[user_id])
    miu /= ratings_num
    for item_id in train_data_item:
        sum = 0.0
        for user_id, score in train_data_item[item_id]:
            sum += score
        bi[item_id] = sum / len(train_data_item[item_id])
    bx -= miu
    bi -= miu
    return miu, bx, bi
```

3 实验原理

3.1 基于物品（item）的协同过滤算法

该方法的基本思想是使用物品之间的相似度来进行预测，即对于物品 i 找到其他相似的对象，然后基于对相似对象们的评分 (Ratings)，估计用户对 i 的评分。这里的相似度计算和预测函数可以沿用 User-user 中的方法。

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

这里的 s_ij 表示对象 i 和 j 的相似度，r_xj 表示用户 x 对对象 j 的评分，N(i; x) 一系列与 i 相似且被用户 x 评过分的对象。

具体到应用场景上，User-user 因为是基于相似用户，所以对社交性有更高的要求，适用于新闻推荐场景。因为新闻本身的兴趣点往往是分散的，相比用户对不同新闻的兴趣偏好，新闻的及时性、热点性

往往是其更重要的属性，而 User-user 正适用于发现热点，以及跟踪热点的趋势。另一方面，Item-item 更适用于兴趣变化较为稳定的应用。比如在 Amazon 的电商场景中，用户在一个时间段内更倾向于寻找一类商品，这时利用物品相似度为其推荐相关物品是契合用户动机的。

在实际情况中，Item-item 协同推荐比 User-user 协同推荐效果更好，因为用户的口味可能是多样的，对象相对简单。

3.2 局部和全局效应建模

在之前的模型中，存在一些问题

- 相似度衡量标准是“任意的”
- 成对相似性忽略用户之间的相互依赖关系
- 采用加权平均可能会有限制

因此我们提出解决方案：可以直接用机器学习的方法从数据中估计出 w_{ij}，而不是使用 s_ij 来衡量相似度。

$$r_{xi} = b_{xi} + \frac{\sum_{j \in \mathcal{N}(i;x)} s_{ij} \cdot (x_{xj} - b_{xj})}{\sum_{j \in \mathcal{N}(i;x)} s_{ij}}$$

其中:

$$b_{xi} = \mu + b_x + b_i$$

整体平均评分为 μ，用户 x 的评分偏差为 b_x = 用户 x 的平均评分 μ，电影 i 的评分偏差为 b_i = 电影 i 的平均评分 μ。

所以这里我们用 w_{ij} 计算加权总和，从数据中进行评估，而不直接使用相似度计算，公式如下：
$\widehat{r_{xi}} = b_{xi} + \sum_{j \in N(i;x)} w_{ij} \left(r_{xj} - b_{xj}\right)$

3.3 矩阵分解

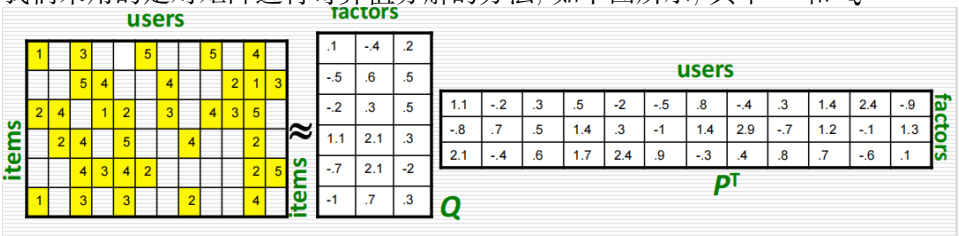
我们之前使用权值 w_{ij} 得出了一个预测公式，权重 w_{ij} 根据其作用得出，明确说明相邻电影之间的相互关系。接下来，要关注潜在因子模型（Latent factor model），以提取出“区域”相关性（“regional” correlations）。

由于 CF 本身存在泛化能力较弱的缺点，无法将两个物品相似这一信息推广到其他物品的相似性计算之上。这会造成一个比较严重的后果，那就是很强的“头部效应”，容易与大量对象有高相似性；而尾部对象（冷门对象）因为特征向量比较稀疏，与其他对象的相似性就会比较低，因而很少被推荐。如下所示由 4 个对象组成的共现矩阵.

$$\begin{array}{l} A[0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 0\ \ 1] \\ B[0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0] \\ C[0\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0] \\ D[1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1] \end{array} \Rightarrow \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} - & 0.00 & 0.00 & 0.71 \\ 0.00 & - & 0.00 & 0.18 \\ 0.00 & 0.00 & - & 0.18 \\ 0.71 & 0.18 & 0.18 & - \end{bmatrix} \end{array}$$

可以看到 A, B, C 之间毫无相似性，但 D 却与 A, B, C 都有相似性。然而 D 的这些相似性仅仅因为它比较热门，这就是 CF 最大的缺陷: 头部效应明显，无法处理稀疏矩阵。为解决上述问题，同时增加模型的泛化能力，矩阵分解技术被提出。该方法在协同过滤共现矩阵的基础上，使用更稠密的隐向量表示用户和物品，挖掘用户和物品的隐含兴趣和隐含特征，在一定程度上弥补了协同过滤模型处理稀疏矩阵能力不足的问题。

我们采用的是对矩阵进行奇异值分解的方法, 如下图所示, 其中 P 和 Q



这里的 factors 是一个超参数, 即隐向量的维度, 由用户决定，factor 数量越大，隐向量表达能力更强，效果会越好，但也会增大计算量，降低泛化性，因此这需要权衡。在该方法中，我们希望重构出的矩阵中“已知”评分的误差尽可能小，而未知评分的部分我们暂时忽略。我们想知道用户 i 对对象 j 的评分，只需要用对应的行列相乘即可得到结果。

我们采用最小化误差的方法来求 P 和 Q:

$$min_{P,Q} \sum_{(i,x)\in R} (r_{xi} - q_i \cdot p_x)^2$$

该目标函数的目的是让原始评分 r_{ui} 与用户向量和物品向量之积 $q_i^T p_u$ 的差尽量小，这样才能最大限度地保存共现矩阵的原始信息。这里的 K 是所有用户评分样本的集合。为了减少过拟合现象，加入正则化项后的目标函数如下所示:

$$\min_{\boldsymbol{q}^*,\boldsymbol{p}^*} \sum_{(\mathrm{u,i})\in K} \left(\boldsymbol{r}_{\mathrm{u,i}} - \boldsymbol{q}_i^\top \boldsymbol{p}_{\mathrm{u}}\right)^2 + \lambda(\|\boldsymbol{q}_i\| + \|\boldsymbol{p}_{\mathrm{u}}\|)^2$$

4 核心算法解析

4.1 基于物品属性的协同过滤

4.1.1 余弦相似度

d1, d2 是两个由用户评分组成的两个物品属性的向量,self.matrix_item 存储着 item_id:[(user_),], 的键值对。

```
def CosineCorrelation(self, id1, id2):
    dic1 = {id1:self.matrix_item.get(id1)}
    dic2 = {id2:self.matrix_item.get(id2)}
    n1 = np.zeros(self.NumOfItem)
    n2 = np.zeros(self.NumOfItem)
    //对numpy进行初始化，因为使用numpy进行数据计算比较快
    for id, v1 in dic1.items():
        for it in v1:
            n1[it[0]] = it[1] - self.itemBia[id]
    for id, v2 in dic2.items():
        for it in v2:
            n2[it[0]] = it[1] - self.itemBia[id]
    return np.multiply(n1,n2).sum() / (np.linalg.norm(n1)* np.linalg.norm(n2))
```

注：一开始考虑到计算量，先使用计算量较小的余弦相似度而不是皮尔逊卡方进行计算相似度。

4.1.2 协同过滤核心代码

- self.matrix_user 存储了 user: [(item_id, score),] 的键值对，key 为用户 id，value 是包含了改用户打分的列表。
- 遍历用户，_score 为将来该用户对所有电影评分的数组，要是用户已经打过分的电影，则无需使用协同过滤的方式进行预测；若是用户未评分的电影，对所有电影两两使用余弦相似度函数求相似度 Sij，然后取其前 neighbor 个值进行进一步计算
- self.itemBia 和 self.userBia 存储了 item 和 user 的偏置，self.GlobalMean 则是分数全局平均，根据前 neighbor 个相似物品的相似度求加权和，以及加上物品偏置和用户偏置、全局平均。

```
def fulfill_table(self, neighbor=2):
    # 协同过滤
    for user, score_lis in self.matrix_user.items():
        _score = np.zeros(self.NumOfItem) # 分数
        _temp = [] # 已经打分的movie_id
        for it in score_lis:
            _score[it[0]] = it[1]
            _temp.append(it[0])
        for movie in trange(self.NumOfItem):
            # 遍历所有电影计算相似度
            if _score[movie] == 0:
                # 取两个计算
                _neighbor = [(0,-1) for _ in range(neighbor)]
                for v in _temp:
                    Corr = self.CosineCorrelation(movie, v)
                    if v != movie and Corr>_neighbor[0][1]:
                        _neighbor.pop(0)
                        _neighbor.append((v,Corr))
                        _neighbor.sort(key=lambda x:x[1])
                _tsum=0.0
                for ne in _neighbor:
                    _tsum += ne[1]
                for ne in _neighbor:
                    _score[movie] += _score[movie]*ne[1]/_tsum
                _score[movie] +=
                    self.GlobalMean+self.itemBia[movie]+self.userBia[user]
```

注：由于物品数量 50w 个，使用上述方法直接计算物品两两之间相似度实际上会花费大量时间，就算将原本物品之间的相似度提前计算存入本地，共 $\binom{500000}{2}$ 条目，若使用 numpy 进行存储，将会产生近

千 GB 的空间。因此直接使用上算法进行 $\mathcal{O}(n^3)$ 的计算，大约要耗费 7 天的时间，而解决该问题则希望采取并行的方式加速，而且计算两两之间相似度时可以考虑使用随机算法寻找 neighbor。

不过，SVD 实际解决了协同过滤空间、时间的复杂度的难题，于是下面直接讲解我们小组的主要实验。

4.2 基于全局与局部效应和 SVD 的协同过滤

下面介绍进行分数预测的三个模型；

- Base 模型: 求完整训练集的 GloabalMean，用户 Bia 和物品偏置
- SVD 模型: 利用矩阵分解求出 P 和 Q 矩阵, 结合 base 模型进行评分
- attr_SVD 模型: 只使用对应属性的值做 svd, 再结合 base、svd 共三个模型进行评分

4.2.1 Base 模型

模型参数如下:

表 1: 模型输入输出

模型输入	模型输出
完整数据集	GlobalMean; user_bia:vector with shape (number_of_users,) item_bia: vector with shape (number_of_items)

- 初始化函数：定义了一些必要的变量和数据结构，包括用户偏置、物品偏置、训练集数据、测试集数据和全局平均分等。
- 计算全局平均分函数：遍历训练集数据，计算所有评分的平均值，作为全局平均分。
- 预测评分函数：根据用户偏置、物品偏置和全局平均分，计算出给定用户对给定物品的预测评分。
- 测试函数：遍历测试集数据，对每个用户和物品组合进行预测评分，并将结果保存在一个字典中。如果物品不在训练集数据中，则将预测评分设置为全局平均分的 10 倍，这是由于，在读取训练分数时已经将分数除 10 以避免训练时出现数字过大溢出的情况。最后将预测结果写入到指定的文件中，并返回预测结果字典。

Baseline 代码如下:

```
class Baseline:
    def __init__(self, data_path='./data/train_user.pkl'):
        self.bx = load_pkl(bx_pkl) # 用户偏置
        self.bi = load_pkl(bi_pkl) # 物品偏置
        self.idx = load_pkl(idx_pkl)
        self.train_user_data = load_pkl(data_path)
        self.test_data = load_pkl(test_pkl)
```

```
        self.globalmean = self.get_globalmean() # 全局平均分
    def get_globalmean(self):
        score_sum, count = 0.0, 0
        for user_id, items in self.train_user_data.items():
            for item_id, score in items:
                score_sum += score
                count += 1
        return score_sum / count
    def predict(self, user_id, item_id):
        pre_score = self.globalmean + \
            self.bx[user_id] + \
            self.bi[item_id]
        return pre_score
    def test(self, write_path='./result/result.txt'):
        print('Start testing...')
        predict_score = defaultdict(list)
        for user_id, item_list in self.test_data.items():
            for item_id in item_list:
                if item_id not in self.idx:
                    pre_score = self.globalmean * 10
                else:
                    new_id = self.idx[item_id]
                    pre_score = self.predict(user_id, new_id) * 10
                    if pre_score > 100.0:
                        pre_score = 100.0
                    elif pre_score < 0.0:
                        pre_score = 0.0
                predict_score[user_id].append((item_id, pre_score))
        print('Testing finished.')
        def write_result(predict_score, write_path):
            print('Start writing...')
            with open(write_path, 'w') as f:
                for user_id, items in predict_score.items():
                    f.write(f'{user_id}|6\n')
                    for item_id, score in items:
                        f.write(f'{item_id} {score}\n')
            print('Writing finished.')
        if write_path:
            write_result(predict_score, write_path)
        return predict_score
```

4.2.2 SVD 模型

Input:

UserBiasMatrix: User bias matrix; ItemBiasMatrix: Item bias matrix ; P: Matrix P with shape (item, factor); Q: Matrix Q with shape (user, factor)

11.2 表 2 **模型输入输出**

模型输入	模型输出
UserBiasMatrix: User bias matrix ItemBiasMatrix: Item bias matrix P: Matrix P with shape (item, factor) Q: Matrix Q with shape (user, factor)	scores

我们将模型四个矩阵分别使用 np.random.normal(0, 0.1, (matrix)) 进行初始化，模型参数如下：

SVD 模型参数

参数	值
学习率	5e-3
P 正则化系数	0.05
Q 正则化系数	0.05
b_x 正则化系数	0.05
b_i 正则化系数	0.05
隐向量维度 $factor$	50

4.2.2.1 模型训练

SVD 模型代码如下：

```
# 有偏置，有正则化(四个正则化项)的矩阵分解
class SVD:
    def __init__(self, model_path='./model',
                 data_path='./data/train_user.pkl', lr=5e-3,
                 lamda1=1e-2, lamda2=1e-2, lamda3=1e-2, lamda4=1e-2,
                 factor=50):
        self.bx = load_pkl(bx_pkl) # 用户偏置
        self.bi = load_pkl(bi_pkl) # 物品偏置
        self.lr = lr # 学习率
        self.lamda1 = lamda1 # 正则化系数1，乘以P
```

```
        self.lamda2 = lamda2 # 正则化系数2，乘以Q
        self.lamda3 = lamda3 # 正则化系数3，乘以bx
        self.lamda4 = lamda4 # 正则化系数4，乘以bi
        self.factor = factor # 隐向量维度
        self.idx = load_pkl(idx_pkl)
        self.train_user_data = load_pkl(data_path)
        self.train_data, self.valid_data = split_data(self.train_user_data)
        self.test_data = load_pkl(test_pkl)
        self.globalmean = self.get_globalmean() # 全局平均分
        # 随机初始化矩阵Q(物品)和P(用户)
        self.Q = np.random.normal(0, 0.1, (self.factor, len(self.bi)))
        self.P = np.random.normal(0, 0.1, (self.factor, len(self.bx)))
        self.model_path = model_path


    def get_globalmean(self):
        score_sum, count = 0.0, 0
        for user_id, items in self.train_user_data.items():
            for item_id, score in items:
                score_sum += score
                count += 1
        return score_sum / count


    def predict(self, user_id, item_id):
        pre_score = self.globalmean + \
            self.bx[user_id] + \
            self.bi[item_id] + \
            np.dot(self.P[:, user_id], self.Q[:, item_id])
        return pre_score


    def loss(self, is_valid=False):
        loss, count = 0.0, 0
        data = self.valid_data if is_valid else self.train_data
        for user_id, items in data.items():
            for item_id, score in items:
                loss += (score - self.predict(user_id, item_id)) ** 2
                count += 1
        # 如果是训练集，计算正则化项
        if not is_valid:
            loss += self.lamda1 * np.sum(self.P ** 2)
```

```
            loss += self.lamda2 * np.sum(self.Q ** 2)
            loss += self.lamda3 * np.sum(self.bx ** 2)
            loss += self.lamda4 * np.sum(self.bi ** 2)
        loss /= count
        return loss


    def rmse(self):
        rmse, count = 0.0, 0
        for user_id, items in self.train_user_data.items():
            for item_id, score in items:
                rmse += (score - self.predict(user_id, item_id)) ** 2
                count += 1
        rmse /= count
        rmse = np.sqrt(rmse)
        return rmse

    def train(self, epochs=10, save=False, load=False):
        if load:
            self.load_weight()
        print('Start training...')
        for epoch in range(epochs):
            for user_id, items in tqdm(self.train_data.items(), desc=f'Epoch {epoch +
                1}'):
                for item_id, score in items:
                    error = score - self.predict(user_id, item_id)
                    self.bx[user_id] += self.lr * (error - self.lamda3 *
                        self.bx[user_id])
                    self.bi[item_id] += self.lr * (error - self.lamda4 *
                        self.bi[item_id])
                    self.P[:, user_id] += self.lr * (error * self.Q[:, item_id] -
                        self.lamda1 * self.P[:, user_id])
                    self.Q[:, item_id] += self.lr * (error * self.P[:, user_id] -
                        self.lamda2 * self.Q[:, item_id])
            print(f'Epoch {epoch + 1} train loss: {self.loss():.6f} valid loss:
                {self.loss(is_valid=True):.6f}')
        print('Training finished.')

        if save:
            self.save_weight()


```

```
    def test(self, write_path='./result/result.txt', load=True):
        if load:
            self.load_weight()
        print('Start testing...')
        predict_score = defaultdict(list)
        for user_id, item_list in self.test_data.items():
            for item_id in item_list:
                if item_id not in self.idx:
                    pre_score = self.globalmean * 10
                else:
                    new_id = self.idx[item_id]
                    pre_score = self.predict(user_id, new_id) * 10
                    if pre_score > 100.0:
                        pre_score = 100.0
                    elif pre_score < 0.0:
                        pre_score = 0.0

                predict_score[user_id].append((item_id, pre_score))
        print('Testing finished.')
```

训练思路

- 首先将数据集打乱后划分为训练集和测试集，比例设置为 0.85
- loss 计算：对于每个打分元组 (user_id, item_id, score)，通过 predict 函数计算预测值 _score，score 减去预测值，计算它们的平方和。
- 梯度传导，根据求导公式，对四个输入进行更新。
- 输出每一个 epoch 在训练集的 loss
- 值得注意的是，模型两个偏置量 b_x 和 b_i 是需要在训练过程中优化的，而其初值我们将其设置为 Baseline 模型的偏置值，以达到”起跑线领先”的效果。

4.2.3 SVD_attr 模型

这个模型在上述 SVD 模型的基础上，结合了 ItemAttribute.txt 文件的使用，力求充分利用已知数据来做出更精准的预测。

- 首先，通过继承 Solution 类创建 Solution_Attr 类，需要包括模型路径、数据路径、学习率、正则化权重等参数.
- loss 函数的实现: 如果 is_valid 参数为 False，表示当前处理的是训练集数据，那么就对每个用户和物品评分计算预测结果与真实结果的平方误差，并对所有误差求和并除以总数得到平均误差; 如果 is_valid 参数为 True，则表示当前处理的是验证集数据。在这种情况下，不需要进行正则化，而是通过结合当前模型的预测结果和另外两个属性模型的预测结果来计算损失. 这里采用了

简单的加权平均方法，其中权重分别是 0.8、0.1 和 0.1，分别表示当前模型、属性 1 模型和属性 2 模型的贡献比例。

- 将属性值运用到推荐系统 SVD 中: 首先需要根据属性值构建相应的模型，接下来就是如何将不同模型的预测结果结合起来。常用的方法有加权平均、矩阵分解等。在这段代码中，我们使用了加权平均的方法，将当前模型、属性 1 模型和属性 2 模型的预测结果进行加权平均得到最终的预测结果，从而提高推荐系统的精度和覆盖率。

SVD_attr 模型代码如下：

```
class SVD_Attr(SVD):
    def __init__(self, model_path='./model',
                 data_path='./data/train_user.pkl',
                 lr=5e-3,
                 lamda1=1e-2, lamda2=1e-2, lamda3=1e-2, lamda4=1e-3,
                 factor=50):
        super().__init__(model_path, data_path, lr, lamda1, lamda2, lamda3, lamda4,
            factor)
        self.attr1 = SVD(model_path='./model_attr1')
        self.attr2 = SVD(model_path='./model_attr2')
        self.attr1.load_weight()
        self.attr2.load_weight()

    def loss(self, is_valid=False):
        loss, count = 0.0, 0
        data = self.valid_data if is_valid else self.train_data
        # 如果是训练集
        if not is_valid:
            for user_id, items in data.items():
                for item_id, score in items:
                    loss += (score - self.predict(user_id, item_id)) ** 2
                    count += 1
            loss += self.lamda1 * np.sum(self.P ** 2)
            loss += self.lamda2 * np.sum(self.Q ** 2)
            loss += self.lamda3 * np.sum(self.bx ** 2)
            loss += self.lamda4 * np.sum(self.bi ** 2)
            loss /= count
        # 如果是验证集，不需要正则化，同时结合另外两个属性模型的预测结果
        # 按0.8*本模型+0.1*属性1模型+0.1*属性2模型的比例计算loss
        else:
            for user_id, items in data.items():
                for item_id, score in items:
                    loss += (score - self.predict(user_id, item_id)) ** 2
                    count += 1
```

```
            loss *= 0.8
            loss += 0.1 * self.attr1.loss(is_valid=True)
            loss += 0.1 * self.attr2.loss(is_valid=True)
            loss /= count
        return loss

    def predict(self, user_id, item_id):
        return 0.8 * super().predict(user_id, item_id) + \
            0.1 * self.attr1.predict(user_id, item_id) + 0.1 *
                self.attr2.predict(user_id, item_id)
```

5 推荐算法实验结果

5.1 Baseline 计算结果

表 4: Baseline 计算结果

用户 ID	物品 ID	评分值
0	208031	89.0074548913919
0	193714	99.3189492922157
0	393064	93.41849680352792
0	207030	100.0
0	112040	73.495419880451
0	464229	85.39440977944089
1	55971	100.0
1	583090	92.74407389873193
1	180171	100.0
1	617646	100.0
1	175835	100.0
1	553890	100.0

5.2 SVD 计算结果

表 5: SVD 计算结果

用户 ID	物品 ID	评分值
0	208031	91.56452132477179
0	193714	65.18324435919928
0	393064	37.28245349410778
0	207030	79.5223011677439
0	112040	51.451792136036715
0	464229	77.17008874354663
1	55971	89.14414602588442
1	583090	87.30347518892893
1	180171	87.8170572411186
1	617646	91.58699804481857
1	175835	89.11096494711528
1	553890	89.7784090003257

5.3 带属性的 SVD 计算结果

表 6: 带属性的 SVD 计算结果

用户 ID	物品 ID	评分值
0	208031	98.86593463013403
0	193714	68.80643768416341
0	393064	84.07892229673476
0	207030	91.38121457103834
0	112040	59.83807203259182
0	464229	78.60939763535667
1	55971	89.88532431945927
1	583090	83.57154890059475
1	180171	90.84206666471499
1	617646	91.65473111878568
1	175835	89.76072049399745
1	553890	90.0442265908409

6 属性的利用

题目的 itemAttribute.txt 文件中给出物品的两个特征，直观上看，这两个属性一定程度决定了物品之间的相似度，即将两个属性组成的元组 (Attr1, Attr2)，看作物品投影到二维空间的向量，那么使用余弦相似度 (皮尔逊卡方相似度) 可以得到其二者之间的相似度。而我们在使用协同过滤计算物品之间相似度时，每一个物品的维数都是 user 的数量（本实验中为 19w），维数过高导致计算量巨大。那么使用这个二维向量计算其相似度可以减少计算量。

同时第二个问题是物品数量规模在 50w，计算两两之间相似度依旧是耗费时间；我们小组的解决方案是利用 SVD，对**单独含有属性 1 或者属性 2 的三元组** (user, item, score) 进行训练，最终将导致同时含有属性 1 或者含有属性 2 的物品有了更高的相似度，为其增加了权重。

但缺点便是我们没有利用属性的具体大小，使得物品属性的条件减弱。

7 SVD 过程中的问题和思考

7.1 时间与空间

- **时间：** svd 模型每一个 epoch 大概需要 50s 的训练时间。属性提取后的两个 svd 属性模型训练每个 epoch 大概 25s 的时间，带两个 svd 属性模型的主模型训练仍需 50s/epoch。
- **磁盘：** 共 500 万条数据，我们以稀疏的形式存储两次 (item_matrix 和 user_matrix)，每个 37mb，PQ 矩阵大小分别为 [item_number:factor],[user_number:factor]，P 矩阵在磁盘存储占 7mb，Q 占 170mb
- **内存：** 训练可选择读取权重继续训练 or 不读取权重开始训练，我们需要读取已经存储为 pkl 文件的训练集数据，同时若选择加载权重，则需要读取磁盘存储的 PQ 矩阵，读入后训练过程内存大约占用 3GB。测试过程占用内存较少。

7.2 梯度爆炸

一开始做 SVD 时，将原始分数输入模型，进行训练集上的训练、梯度，但由于计算 score 值介于 (0-100), 导致在开始训练的 epoch 中 loss 过大导致梯度爆炸，直接程序崩溃。

解决方法

- 调整学习率大小, 继续减少其值, 但是训练出效果差, 在测试集上的 loss 只能减低到原来的 85%。
- 将分数映射，一开始我们直接映射到（0，1），但随着训练的进行，由于每个 epoch 要计算平均 loss 时要做大整数的除法，使得其精度出现问题，loss 下降并不明显，且难以观察是否收敛。
- 最终将 score 映射至（0，10），lr 调整至 1e-3 级别，基本保证 loss 曲线正常。

7.3 参数估计

- **factor** 使用 SVD 分解，旨在于使用 P、Q 矩阵近似原本的 [item, user] 的打分矩阵，factor 过大 (上万)，导致内存无法承载；过小又会导致难以对原矩阵进行预测。但多大算大以及多小算小成为问题。
- **正则惩罚项系数**惩罚项旨在于防止参数在训练集上拟合过度，但惩罚项又不能过大，否则使得训练的梯度过多来自于惩罚项而非真实预测值和真实值的偏差；过小则导致过拟合现象严重，使其在验证集上表现极差。而我们在后期的实验中就发生很多这样的现象，验证集上的 loss 在下降到一个度后开始反弹、上升。

  思考

  对于 factor 的估计，我们最终使用 50 的区间，过多会使得训练时间延长，但训练效果并没有明显的提升 (这里效果观察训练集 loss 降低的快慢)，惩罚项系数定位 0.01，我们优先防止其过大导致干扰训练。

7.4 收敛判断

实验过程中，发现训练集上的 loss 曲线一直达不到收敛，而测试集的 loss 曲线同样有这样的问题，要么出现一直降低，但降低速度很慢却没有下降的趋势或者出现反弹。我们认为主要原因在于 factor 的选择不足以很好的拟合原矩阵，其次是我们参数的选取也存在偏差。

7.5 结果估计

下面选一个由 factor=50，lr=0.005，惩罚项系数 0.05 的结果：

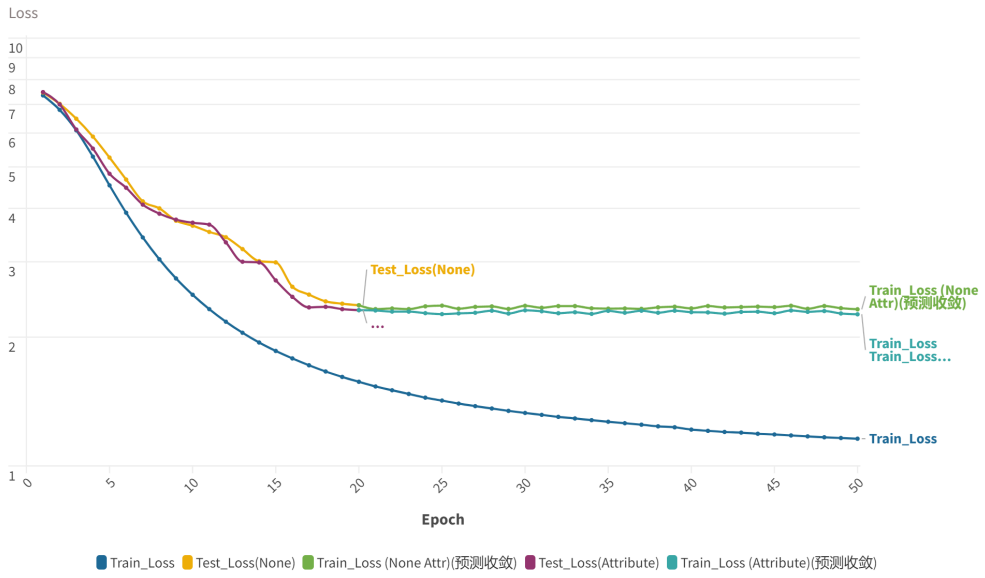


图 7.1: 结果展示

注: 绿色后缀的是根据曲线最后几个结果拟合出来的收敛的猜测值，因为在实际实验过程中是否收敛难以判断。**分析**：可以看到，含有 Attribute 的 loss 曲线下降还是比没有 Attribute 开始下降快一点，最后预测出来的收敛值也更低，因此可以认为我们使用 Attribute 标签，一定程度上可以优化我们原有的 SVD 模型。

致 谢

感谢杨老师耐心细致的讲解，您深入浅出的分析让我们对推荐算法有了清晰的认知并对大数据这个领域产生了浓厚的兴趣。

感谢助教李学长的耐心协调和答疑，为我们排忧解难，更加严谨地从事学术活动。

作为大二的学生凭借着兴趣的驱使在本学期选了这门大三的选修课, 感谢老师们的耐心指导, 收获颇丰! 谨以此报告代表我对大数据计算与应用这门课的喜爱以及对学问的敬重!