南　　开　　大　　学

计　算　机　学　院

大数据计算与应用实验报告

推荐算法的实现与优化

姓名：艾明旭　刘璞睿　常昊

年级：2021 级

指导教师：杨征路

2024 年 6 月 15 日

目录

一、 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

(一) 数据集介绍

- **train.txt**：该文件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量不尽相同，该数据集将用于后续的模型训练。
- **test.txt**：该文件包含待评分的用户序号以及其需要评价的 6 个物品编号。此数据集用于测试训练好的模型。
- **ItemAttribute.txt**：该文件包含每个物品的两个属性值（并非所有物品都具有两个属性值，因此需要进行预处理）。通过这两个属性值可以计算物品之间的相似度，从而更加科学地进行基于物品的协同过滤，提升模型的训练精度。
- **ResultForm.txt**：该文件用于保存预测结果。将 **test.txt** 文件中待测试物品的评分结果输出到此文件中，以便提交和评估。

(二) 数据预处理

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。具体步骤如下：

1. 定义一个空集合 `all_nodes`，用于存储所有的节点。
2. 打开 `train.txt` 文件，并循环读取每一行内容（使用 `while ... readline()` 实现）。
3. 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。
4. 提取其中的节点 ID，将其加入到 `all_nodes` 集合中。
5. 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。

以下是实现该过程的示例代码：

```
def create_index(train_path):
    unique_items = set()
    with open(train_path, 'r') as file:
        while (line := file.readline()) != '':
            _, num = map(int, line.strip().split('|'))
            for _ in range(num):
                line = file.readline()
                item_id, _ = map(int, line.strip().split())
                unique_items.add(item_id)
    # 创建物品ID到索引的映射字典
    item_to_index = {item: idx for idx, item in enumerate(sorted(unique_items
        ))}
```

```
    return item_to_index
```

上述代码片段展示了如何从 **train.txt** 文件中读取数据，并建立一个有序的节点映射。这样可以确保在后续的分析和计算中，每个节点都有唯一且连续的索引。

2. 处理训练集和测试集

具体分为以下几个步骤:

1. **load_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 `user_data` 和 `item_data` 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 `user_data` 和 `item_data` 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. **load_attributes_ 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. **filter_data_by_attributes 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 `user_data` 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 `data_with_attr1` 和 `data_with_attr2` 字典中。

4. **load_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 `test_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 `user_data` 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。

以下是上述几个函数的代码实现:

```
def load_training_data(path, index_map):
    user_data, item_data = defaultdict(list), defaultdict(list)
    with open(path, 'r') as file:
        while True:
            line = file.readline()
            if not line:
                break
            user_id, count = map(int, line.strip().split('|'))
            for _ in range(count):
                item_id, score = map(float, file.readline().strip().split())
                user_data[user_id].append([index_map[item_id], score / 10])
                item_data[index_map[item_id]].append([user_id, score / 10])
    return user_data, item_data

def load_attributes(path, index_map):
```

```
    attributes = defaultdict(list)
    with open(path, 'r') as file:
        for line in file:
            item_id, attr1, attr2 = line.strip().split('|')
            attr1 = 1 if attr1 != 'None' else 0
            attr2 = 1 if attr2 != 'None' else 0
            if int(item_id) in index_map:
                attributes[index_map[int(item_id)]].extend([attr1, attr2])
    return attributes

def filter_data_by_attributes(user_data, attributes):
    data_with_attr1, data_with_attr2 = defaultdict(list), defaultdict(list)
    for user_id, items in user_data.items():
        for item_id, score in items:
            if item_id in attributes and len(attributes[item_id]) == 2:
                if attributes[item_id][0]:
                    data_with_attr1[user_id].append([item_id, score])
                if attributes[item_id][1]:
                    data_with_attr2[user_id].append([item_id, score])
    return data_with_attr1, data_with_attr2

def load_test_data(path):
    test_data = defaultdict(list)
    with open(path, 'r') as file:
        for line in file:
            user_id, count = map(int, line.strip().split('|'))
            for _ in range(count):
                item_id = int(file.readline().strip())
                test_data[user_id].append(item_id)
    return test_data

def split_data(data, train_ratio=0.85, shuffle=True):
    train_set, validation_set = defaultdict(list), defaultdict(list)
    for user_id, items in data.items():
        if shuffle:
            np.random.shuffle(items)
        split_point = int(len(items) * train_ratio)
        train_set[user_id], validation_set[user_id] = items[:split_point],
            items[split_point:]
    return train_set, validation_set

def load_pkl(pkl_path):
    with open(pkl_path, 'rb') as f:
        return pickle.load(f)

def save_data_to_pickle(path, data):
    with open(path, 'wb') as file:
        pickle.dump(data, file)
```

通过上述函数，我们可以高效地处理训练集和测试集数据，进行模型训练和验证。

3. 统计数据特征

我们首先通过以下代码读取 `train.txt` 中的用户 ID 和物品 ID，并统计出相关信息：

```
def extract_statistics(file_path):
    data = []
    with open(file_path, 'r') as file:
        while (line := file.readline()) != '':
            user_id, num_ratings = map(int, line.strip().split('|'))
            for _ in range(num_ratings):
                line = file.readline()
                item_id, score = map(float, line.strip().split())
                data.append([user_id, item_id, score])
    df = pd.DataFrame(data, columns=['user_id', 'item_id', 'score'])
    user_count = df['user_id'].nunique()
    item_count = df['item_id'].nunique()
    total_ratings = len(df)
    max_user_id = df['user_id'].max()
    max_item_id = df['item_id'].max()
    average_rating = df['score'].mean()
```

最终我们得到如下结果：

- 用户数: 19835
- 评分的物品数: 455691
- 评分总数: 5001507
- 最大用户 ID: 19834
- 最大物品 ID: 624960
- 平均评分：49.50627100991761

4. 获取均值与偏差

在推荐系统中，计算全局评分均值和用户、物品偏差是非常重要的步骤。具体步骤如下：

- **计算全局评分均值** μ：遍历 `train_data_user`，对每个用户的评分进行累加，然后将所有用户的评分总和累加到全局评分均值 μ 中。最后，将 μ 除以评分总数 `ratings_num`，得到全局评分均值 μ。
- **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，然后将评分总和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。接着，将该用户的平均评分减去全局评分均值 μ，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，然后将评分总和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。接着，将该物品的平均评分减去全局评分均值 μ，得到该物品的偏差 b_i。

最后，返回三个变量 μ、b_x 和 b_i，用于后续的评分预测。

```
def calculate_bias(train_data_user, train_data_item):
    global_total = sum(score for items in train_data_user.values() for _,
        score in items)
    global_average = global_total / ratings_num
    user_bias = np.zeros(user_num, dtype=np.float64)
    item_bias = np.zeros(item_num, dtype=np.float64)
    for user_id, items in train_data_user.items():
        total_score = sum(score for _, score in items)
        user_bias[user_id] = total_score / len(items) - global_average
    for item_id, users in train_data_item.items():
        total_score = sum(score for _, score in users)
        item_bias[item_id] = total_score / len(users) - global_average
    return global_average, user_bias, item_bias
```

通过这个函数，我们可以得到全局评分均值、用户偏差和物品偏差，用于后续的评分预测。

5. 优化：小批量梯度下降方法

在本次实验的优化部分中，我们尝试着使用了小批量梯度下降方法。小批量梯度下降是一种优化算法，广泛用于机器学习和深度学习中，特别是在处理大规模数据集时。该方法结合了批量梯度下降和随机梯度下降的特点，旨在平衡计算效率和收敛速度。

1. 算法优点

全批量梯度下降（Batch Gradient Descent）在每次更新中考虑所有训练样本，计算精确但在大数据集上效率低下。随机梯度下降（Stochastic Gradient Descent, SGD）每次仅使用一个样本更新参数，虽然计算快但收敛过程波动大，可能导致收敛到次优解。小批量梯度下降通过每次使用数据集的一个子集（小批量）来平衡这两种方法的优缺点。

- **提高计算效率**：每次更新不需要使用整个数据集，减少了计算资源的消耗。
- **减少训练时间**：通过并行处理各个小批量，可以在多核处理器上显著加速训练过程。
- **增加稳定性**：与 SGD 相比，使用小批量减少了参数更新的方差，使得训练过程更加稳定。
- **避免局部最小值**：小批量梯度下降在一定程度上保留了随机梯度下降跳出局部最小值的能力。

2. 实施步骤

1. **划分小批量**：将训练数据随机划分为多个小批量，每个批量包含固定数量的样本。
2. **迭代更新**：对于每个批量，计算批量的总损失，并基于这些损失更新模型参数。
3. **重复过程**：遍历所有批量直到完成一个完整的训练周期（即一个 epoch），然后重复进行多个 epochs，直至满足停止准则（如收敛或达到最大迭代次数）。

3. 关键代码展示

函数 `mini_batches` 负责将整个数据集划分成多个小批量。每个批量包含预设数量的样本，这对于实施小批量梯度下降是必需的。该函数随机打乱数据顺序，以确保每个批量都是随机的，从而提高模型的泛化能力。

```python
def mini_batches(self, data, batch_size):
    """生成小批量数据"""
    keys = list(data.keys())
    np.random.shuffle(keys)  # 随机打乱用户顺序
    for i in range(0, len(keys), batch_size):
        batch_keys = keys[i:i + batch_size]
        batch_data = {key: data[key] for key in batch_keys}
        yield batch_data
```

在 `train` 方法中，我们使用 `mini_batches` 函数产生的批量数据进行模型训练。对每个批量，我们计算损失并更新模型参数。这个过程在设定的训练周期（epochs）内重复进行。

```python
def train(self, epochs=10, batch_size=128, save=False, load=False):
    if load:
        self.load_weight()
    print('开始训练...')
    for epoch in range(epochs):
        for batch in tqdm(self.mini_batches(self.train_data, batch_size),
            desc=f'Epoch {epoch + 1}'):
            for user_id, items in batch.items():
                for item_id, score in items:
                    error = score - self.predict(user_id, item_id)
                    grad_P = error * self.Q[:, item_id] - self.lamda1 * self.
                        P[:, user_id]
                    grad_Q = error * self.P[:, user_id] - self.lamda2 * self.
                        Q[:, item_id]
                    grad_bx = error - self.lamda3 * self.bx[user_id]
                    grad_bi = error - self.lamda4 * self.bi[item_id]

                    self.P[:, user_id] += self.lr * grad_P
                    self.Q[:, item_id] += self.lr * grad_Q
                    self.bx[user_id] += self.lr * grad_bx
                    self.bi[item_id] += self.lr * grad_bi
    print('训练完成。')
    if save:
        self.save_weight()
```

二、 实验原理

(一) 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。

$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(l;x)} s_{ij}} \tag{1}$$

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，$N(i;x)$ 表示一系列与 i 相似且被用户 x 评过分的对象。

具体到应用场景上，User-user 推荐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、热点性等因素更为重要。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。

(二) 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。

$$r_{xi} = b_{xi} + \frac{\sum_{j\in\mathcal{N}(i;x)} s_{ij}\cdot(x_{xj}-b_{xj})}{\sum_{j\in\mathcal{N}(i;x)} s_{ij}} b_{xi} = \mu + b_x + b_i \tag{2}$$

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:

$$\widehat{r_{xi}} = b_{xi} + \sum_{j\in N(i;x)} w_{ij}\left(r_{xj}-b_{xj}\right) \tag{3}$$

(三) 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:

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。

已知正定矩阵 (positive definite matrix), 如果我们想分析它的特征值和特征向量，可将其分解为如下:

$$Q\Lambda Q^T$$

其中 Λ 为对角矩阵，即为原始矩阵的特征值。Q 为特征向量形成矩阵，如果原始矩阵为对称矩阵，矩阵 Q 可为标准正交矩阵，满足如下:

$$QQ^T = I$$

但，当矩阵非方阵 (rectangular matrix), 以上分解是行不通的，因为该矩阵没有特征值这一概

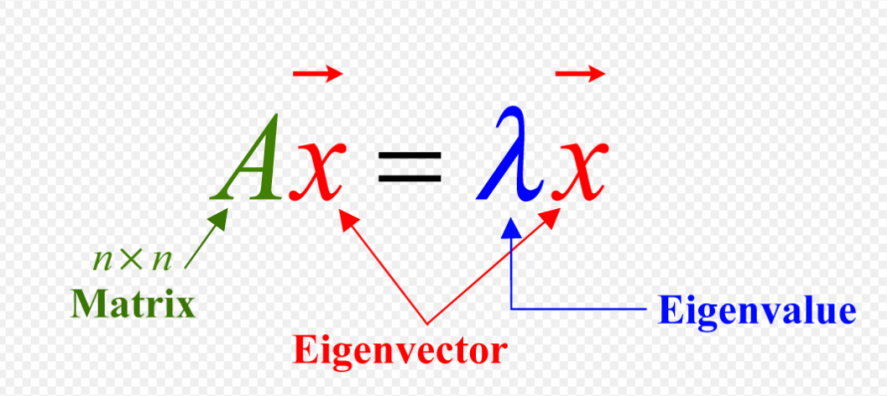

图 1

念的。

由此，便出现了对矩阵进行 SVD 分解，通式如下:

$$A = U\Sigma V^T$$

其中 Σ 为对角阵 (非方阵), 将其对角线处非零的元素记为 $\sigma_1, \cdots, \sigma_r$, 此对角矩阵有两种理解方式:

1. $A^T A$的特征值;
2. A 的奇异值 (singular value)

主对角线处的元素个数与原始矩阵 A 的秩有关。对任意 m 行 n 列的矩阵 A，奇异值分解的综合理解如下:

$$A = U\Sigma V^{\mathrm{T}} = (\textbf{orthogonal})(\textbf{diagonal})(\textbf{orthogonal}).$$

正交矩阵 U：m 行 m 列，该矩阵的每一个列向量都是 AA^T 的特征向量；正交矩阵 V：n 行 n 列，该矩阵的每一个列向量都是 $A^T A$ 的特征向量；对角阵 Σ: m 行 n 列，将 $A^T A$ 或 AA^T 的特征值开根号，得到的就是该矩阵主对角线上的元素，也可以看成矩阵 A 的奇异值。

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:

$$min_{P,Q} \sum_{(i,x)\in R} (r_{xi} - q_i \cdot p_x)^2 \tag{4}$$

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:

$$\min_{q^*,p^*} \sum_{(\mathrm{u,i})\in K} \left(r_{\mathrm{u,i}} - q_i^\top p_{\mathrm{u}}\right)^2 + \lambda(\|q_i\| + \|p_{\mathrm{u}}\|)^2 \tag{5}$$

三、 核心算法解析

(一) 基于物品属性的协同过滤

1. 余弦相似度

d_1, d_2 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: $[(user, score), \ldots]$。

Listing 1: Cosine Similarity Calculation

```
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: 由于物品数量为 50 万个，使用上述方法直接计算物品两两之间的相似度实际上会花费大量时间。即使将原本物品之间的相似度提前计算并存入本地，也需要约千 GB 的空间。因此，直接使用上述算法进行 $\mathcal{O}(n^3)$ 的计算，大约要耗费 7 天的时间。为了解决该问题，我们希望采取并行的方式加速计算，同时在计算两两之间相似度时可以考虑使用随机算法寻找邻居。

不过，SVD 实际上解决了协同过滤在空间和时间复杂度上的难题，因此下面直接讲解我们小组的主要实验。

(二) 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

1. Base 模型

模型参数如下:

模型输入	模型输出
完整数据集	GlobalMean;
完整数据集	user_bia:vector with shape
完整数据集	item_bia: vector with shape

- **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:

Listing 2: Baseline

```
class Baseline:
    def __init__(self):
        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)  # 索引映射
        self.train_user_data = load_pkl(train_user_pkl)  # 训练数据
        self.test_data = load_pkl(test_pkl)  # 测试数据
        self.globalmean = self.get_globalmean()  # 计算全局平均分
    def get_globalmean(self):
        score_sum = sum(score for items in self.train_user_data.values() for
            _, score in items)
        count = sum(len(items) for items in self.train_user_data.values())
        return score_sum / count if count else 0
    def predict(self, user_id, item_id):
        user_bias = self.bx[user_id] if user_id < len(self.bx) else 0
        item_bias = self.bi[item_id] if item_id < len(self.bi) else 0
        pre_score = self.globalmean + user_bias + item_bias
        return pre_score
    def rmse(self):
        scores = [score - self.predict(user_id, item_id) for user_id, items
            in self.train_user_data.items() for item_id, score in items]
        return np.sqrt(np.mean(np.square(scores)))
    def test(self, write_path='./result/result.txt'):
        print('开始测试...')
        predict_score = defaultdict(list)
```

```
        for user_id, item_list in self.test_data.items():
            for item_id in item_list:
                adjusted_item_id = self.idx.get(item_id, item_id)  # 获取调整后的item_id
                if adjusted_item_id >= len(self.bi):
                    pre_score = self.globalmean * 10
                else:
                    pre_score = self.predict(user_id, adjusted_item_id) * 10
                pre_score = np.clip(pre_score, 0, 100)  # 限制预测值在0到100之间
                predict_score[user_id].append((item_id, pre_score))
        print('测试完成。')
        if write_path:
            self.write_result(predict_score, write_path)
        return predict_score
    def write_result(self, predict_scores, write_path):
        print('开始写入结果...')
        with open(write_path, 'w') as f:
            for user_id, items in predict_scores.items():
                f.write(f'{user_id}|{len(items)}\n')  # 写入用户ID和条目数
                for item_id, score in items:
                    f.write(f'{item_id} {score}\n')
        print('写入完成。')
```

2. 基于 SVD 的协同过滤模型

模型输入	模型输出
UserBiasMatrix	scores
ItemBiasMatrix	scores
P: Matrix P with shape	scores
Q: Matrix Q with shape	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:

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_r 正则化系数	0.05
b_i 正则化系数	0.05
隐藏因素维数	50

模型训练 SVD 代码如下:

Listing 3: 有偏置，有正则化的矩阵分解

```
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, K=10):
        self.K = K  # 保存一个实例变量
        self.num_users = 19835  # 用户数量
        self.num_items = 455705  # 物品数量
        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, 乘以P
        self.lamda2 = lamda2  # 正则化系数, 乘以Q
        self.lamda3 = lamda3  # 正则化系数, 乘以bx
        self.lamda4 = lamda4  # 正则化系数, 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()
        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
        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
```

```
52      def precision_recall(self, k):
53          hits = 0
54          rec_count = 0
55          test_count = 0
56          for user_id in self.test_data.keys():
57              test_items = self.test_data[user_id]
58              scores = {item_id: score for item_id, score in self.
                    train_user_data[user_id]}
59              rec_items = sorted(scores.items(), key=lambda x: x[1], reverse=
                    True)[:k]
60              rec_items = [item[0] for item in rec_items]
61              for item in rec_items:
62                  if item in test_items:
63                      hits += 1
64              rec_count += len(rec_items)
65              test_count += len(test_items)
66          precision = hits / float(rec_count) if rec_count > 0 else 0
67          recall = hits / float(test_count) if test_count > 0 else 0
68          return precision, recall
69      def rmse(self):
70          rmse, count = 0.0, 0
71          for user_id, items in self.train_user_data.items():
72              for item_id, score in items:
73                  rmse += (score - self.predict(user_id, item_id)) ** 2
74                  count += 1
75          rmse /= count
76          rmse = np.sqrt(rmse)
77          return rmse
78      def train(self, epochs=10, save=False, load=False):
79          if load:
80              self.load_weight()
81          print('开始训练...')
82          train_losses = []
83          valid_losses = []
84          rmses = []
85          for epoch in range(epochs):
86              for user_id, items in tqdm(self.train_data.items(), desc=f'第␣{
                    epoch␣+␣1}␣轮'):
87                  for item_id, score in items:
88                      error = score - self.predict(user_id, item_id)
89                      grad_P = error * self.Q[:, item_id] - self.lamda1 * self.
                            P[:, user_id]
90                      grad_Q = error * self.P[:, user_id] - self.lamda2 * self.
                            Q[:, item_id]
91                      grad_bx = error - self.lamda3 * self.bx[user_id]
92                      grad_bi = error - self.lamda4 * self.bi[item_id]
93                      self.P[:, user_id] += self.lr * grad_P
94                      self.Q[:, item_id] += self.lr * grad_Q
```

```
                    self.bx[user_id] += self.lr * grad_bx
                    self.bi[item_id] += self.lr * grad_bi
            train_loss = self.loss()
            valid_loss = self.loss(is_valid=True)
            precision, recall = self.precision_recall(k=10)
            f1 = self.f1_score(k=10)
            rmse = self.rmse()
            train_losses.append(train_loss)
            valid_losses.append(valid_loss)
            rmses.append(rmse)
            print(f'第 {epoch + 1} 轮 训 练 损 失 : {train_loss:.6f}, 验 证 损 失 : {valid_loss:.6f}, RMSE: {rmse:.6f}')
            print(f'精 确 率 : {precision:.6f}, 召 回 率 : {recall:.6f}, F1分 数 : {f1:.6f}')
        print('训 练 完 成 。 ')
        if save:
            self.save_weight()
        plt.figure()
        plt.plot(range(len(train_losses)), train_losses, label='Train Loss')
        plt.plot(range(len(valid_losses)), valid_losses, label='Valid Loss')
        plt.title('Loss over epochs')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()
        plt.figure()
        plt.plot(range(len(rmses)), rmses)
        plt.title('RMSE over epochs')
        plt.xlabel('Epochs')
        plt.ylabel('RMSE')
        plt.show()
        return train_losses, valid_losses, rmses
    def f1_score(self, k):
        precision, recall = self.precision_recall(k)
        if precision + recall == 0:
            return 0
        f1 = 2 * precision * recall / (precision + recall)
        return f1
    def test(self, write_path='./result/result.txt', load=True):
        if load:
            self.load_weight()
        print('开 始 测 试 ...')
        predict_score = defaultdict(list)
        precision, recall = self.precision_recall(k=10)
        f1 = self.f1_score(k=10)
        print(f'精 确 率 : {precision:.6f}, 召 回 率 : {recall:.6f}, F1分 数 : {f1:.6f}')
        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('测试完成。')
        def write_result(predict_score, write_path):
            print('开始写入结果...')
            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('写入完成。')
        if write_path:
            write_result(predict_score, write_path)
        return predict_score
    def load_weight(self):
        print('加载模型权重...')
        self.bx = load_pkl(self.model_path + '/bx.pkl')
        self.bi = load_pkl(self.model_path + '/bi.pkl')
        self.P = load_pkl(self.model_path + '/P.pkl')
        self.Q = load_pkl(self.model_path + '/Q.pkl')
        print('加载完成。')
    def save_weight(self):
        print('保存模型权重...')
        if not os.path.exists(self.model_path):
            os.mkdir(self.model_path)
        save_data_to_pickle(self.model_path + '/bx.pkl', self.bx)
        save_data_to_pickle(self.model_path + '/bi.pkl', self.bi)
        save_data_to_pickle(self.model_path + '/P.pkl', self.P)
        save_data_to_pickle(self.model_path + '/Q.pkl', self.Q)
        print('保存完成。')
```

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 $\hat{score}$，然后计算真实评分 $score$ 减去预测评分 $\hat{score}$ 的平方和作为损失函数的一部分。
- 梯度下降：根据损失函数的梯度，对模型的参数进行更新。对于 SVD 模型，需要更新用户偏置 bx、物品偏置 bi、用户特征矩阵 P 和物品特征矩阵 Q。

- 每个 epoch 结束后，输出在训练集上的损失函数值，用来监控模型的训练效果。
- 初始时，模型的偏置 bx 和 bi 可以使用 Baseline 模型预先计算得到的偏置值作为初始值，以便在训练过程中更快地找到损失函数的最小值，这种做法通常称为“起跑线领先”策略。

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：

Listing 4: SVD_attr 模型

```
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
        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
        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)
```

四、 实验结果

(一) baseline 结果

用户 ID	物品 ID	评分	用户 ID	物品 ID	评分
0	127640	99.47764180793705	1	507010	100
0	192496	100	1	55629	100
0	147073	100	1	453396	100
0	70896	97.65374876595888	1	137915	100
0	578821	100	1	261386	100
0	522229	77.93515010949896	1	34525	100

最终的 RMSE 结果为 **1.719646**

(二) SVD 结果

用户 ID	物品 ID	评分	用户 ID	物品 ID	评分
0	127640	87.42613095515372	1	507010	90.47721714120786
0	192496	98.40496032035155	1	55629	90.21848811827564
0	147073	80.43463875565968	1	453396	91.07955639580526
0	70896	93.16644976446555	1	137915	89.21120473430902
0	578821	88.68369022596185	1	261386	90.68972944096242
0	522229	3.7546740838601966	1	34525	90.3152136488046

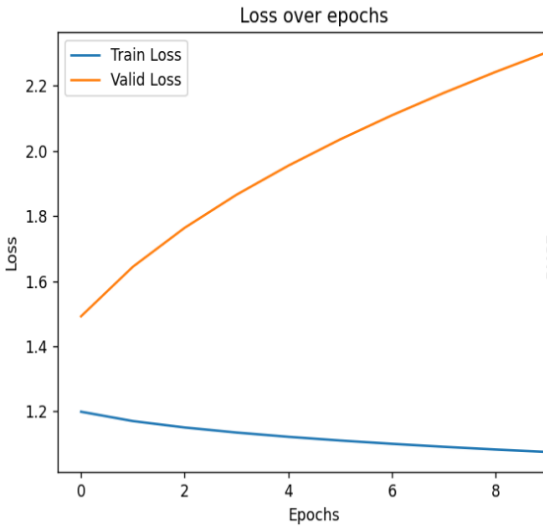

图 2: LSTM validation loss

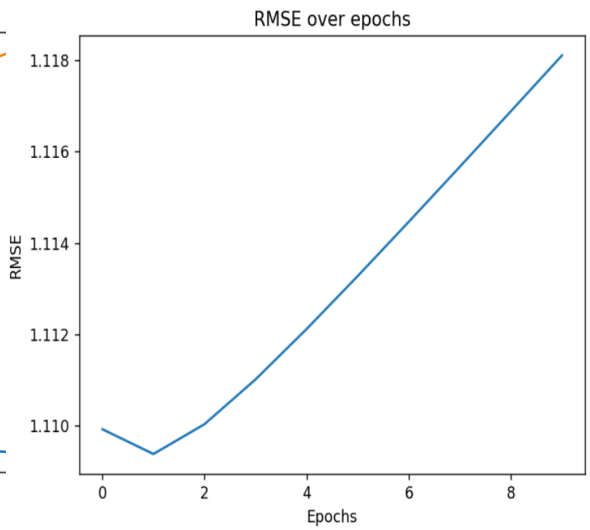

图 3: LSTM validation accuracy

最终的 RMSE 结果为 **1.562485**

(三) SVD_attr 结果

用户 ID	物品 ID	评分	用户 ID	物品 ID	评分
0	127640	94.73247072590503	1	507010	90.24450818879617
0	192496	92.0716347847509	1	55629	90.5120095898581
0	147073	76.74998483587689	1	453396	89.96793381823525
0	70896	100	1	137915	89.63536526319984
0	578821	93.08518865974875	1	261386	90.50732088543677
0	522229	26.50554508335081	1	34525	89.86495920574221

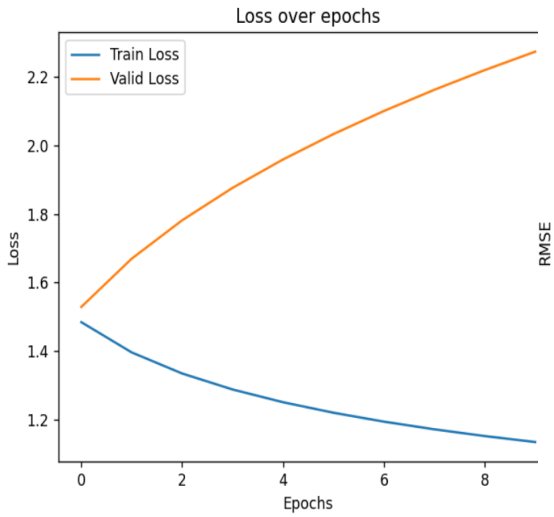

图 4: LSTM validation loss

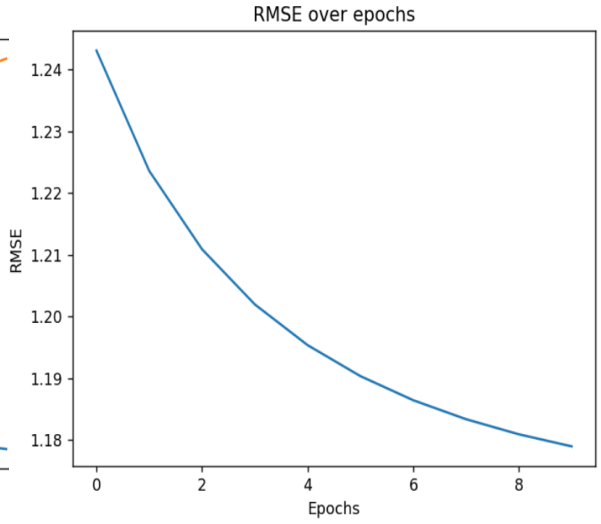

图 5: LSTM validation accuracy

最终的 RMSE 结果为 **1.217598**

(四) 小批量梯度下降处理结果

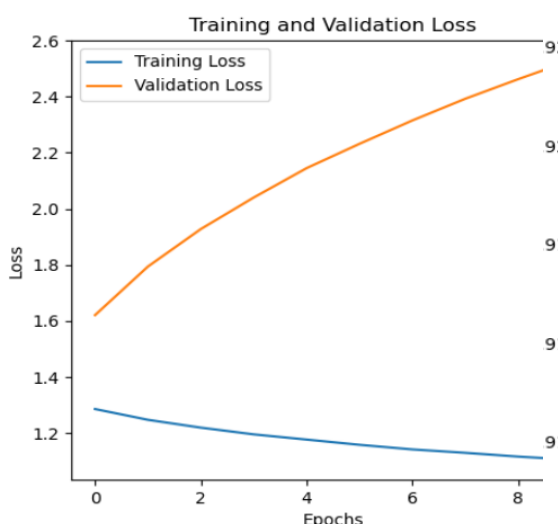

图 6: LSTM validation loss

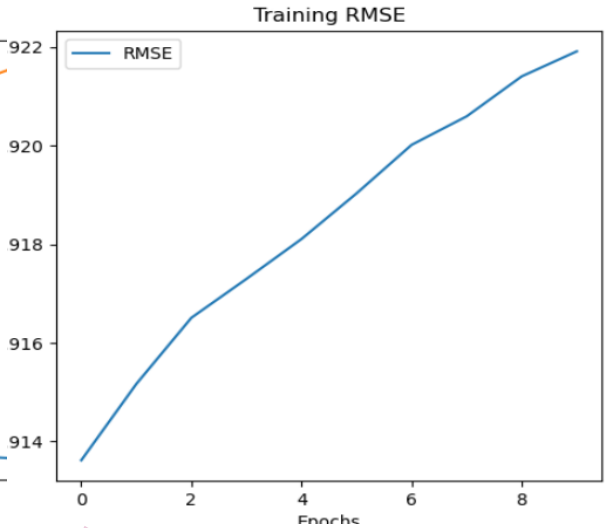

图 7: LSTM validation accuracy

五、 总结

(一) SVD 模型思考与总结

1. 模型基础与实现

SVD 模型是基于矩阵分解的推荐系统模型，通过将用户-物品评分矩阵分解为用户矩阵 P 和物品矩阵 Q 的乘积来进行评分预测。模型初始化包括学习率、正则化系数和隐向量维度等参数，这些参数直接影响模型的收敛速度和泛化能力。

2. 训练过程与优化

模型训练过程中，通过梯度下降优化用户偏置 b_x、物品偏置 b_i、用户矩阵 P 和物品矩阵 Q。训练的目标是最小化预测评分与真实评分之间的平方误差，并加入正则化项以防止过拟合。训练过程中的损失函数不仅考虑了预测误差，还包括了正则化项，通过调节正则化系数来控制模型的复杂度。

3. 模型评估与调优

在模型评估方面，采用了训练集和验证集进行损失函数的计算。训练集用于模型参数的优化，验证集则用于调整超参数和评估模型的泛化能力。常用的评估指标包括均方根误差（RMSE）和平均绝对误差（MAE），这些指标能够客观地评估模型的预测精度。

(二) SVD_attr 模型思考与总结

1. 模型创新与结合属性

SVD_attr 模型在基础的 SVD 模型上进一步引入了物品属性信息。通过继承 SVD 模型并加载两个额外的属性模型,SVD_attr 模型能够更充分地利用物品的属性信息来提升预测精度。每个属性模型都独立地学习物品的隐向量表示，然后通过加权平均的方式与基础 SVD 模型进行整合。

2. 模型整合与加权平均

SVD_attr 模型在预测时通过加权平均的方式，将基础 SVD 模型的预测结果与两个属性模

型的预测结果进行结合。这种加权平均的方式可以灵活调整每个模型在最终预测中的权重，使得不同模型的优势得到充分利用，从而提高推荐系统的整体精度和覆盖率。

3. 模型性能与应用

SVD_attr 模型在推荐系统中的应用能够更精确地为用户推荐物品，尤其是在面对大规模的用户-物品交互数据和复杂的物品属性信息时，能够提供更个性化和准确的推荐服务。通过合理设置属性模型的权重以及调整正则化系数，可以进一步优化模型的性能，提高其在实际应用中的效果。

综上所述，SVD 模型及其衍生模型 SVD_attr 在推荐系统中展现了其优秀的性能和灵活的应用场景，通过对用户行为数据和物品属性的深入挖掘，能够为用户提供更符合个性化需求的推荐服务。

4. 深度学习模型的尝试

对于本次实验，我们也进行了一些深度学习模型诸如 CNN 卷积神经网络，RNN 循环神经网络的尝试，但是由于数据集形状不是很符合的原因，再加上训练的次数比较不适合深度学习。因此，最终得到的结果并不是很好。

但是我们发现一些神经网络可能会较好的适配于这样的数据集，可以在合适的情况下进行一定的预测。未来我们可能会对相应的神经网络模型进行尝试，希望能够得到一个较好的结果。

六、 致谢

感谢杨老师和助教耐心细致的讲解，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。未来我们也将继续研究学习大数据相关的知识，希望能够未来在科研的道路上继续努力，更上一层楼。