
南開大學
Nankai University

计算机学院

大数据计算及应用

# 推荐算法实验报告

姓名：杨馨仪 姚翔载

学号：2011440 2011273

专业：计算机科学与技术

2024 年 6 月 14 日

# 目录

# 1 实验数据

## 1.1 数据集介绍

- 属性数据集 itemAttribute.txt

  共列出 507172 个物品的属性 1 和属性 2 的属性值。其中有 42240 个物品缺乏属性 1，63485 个物品缺乏属性 2，40846 个物品同时缺乏属性 1 和属性 2。

- 训练集 train.txt

  包含 19835 个用户对于 455691 个物品做出的共计 5001507 条评分记录；其中评分均值为 49.5063，评分标准差为 38.2163，分数范围为 0 100。在全部物品中收到评分最多的物品是 147073 号物品，共收到 8332 条评分。而评分最多的用户是 3024 号用户，他/她给出了 21398 个不同的物品评分。

- 测试集 test.txt

  给出了 19835 个用户，每个用户六个待评分物品，涉及物品 28242 个。

## 1.2 数据预处理

由于原始数据存储在文本文件中，占用了大量内存空间，并且其格式不适合直接用于模型训练。为了增强数据的可用性、可靠性和存储效率，同时便于后续的数据处理和分析，我们需要在进行模型训练之前对数据集进行预处理。这里展示了对训练集与测试集的数据预处理工作。在基于 MLP 的推荐方法中还应用了物品属性数据集，其数据处理工作将在第二章关键代码处具体介绍。

### 1.2.1 训练数据

输入文件为”data/train.txt”，包含用户对物品的评分数据。预处理将其转化为一个字典，形式为：{user_id:[[item_id,rating],...],...}。

训练数据预处理代码

```python
def read_train_data(file_path):
    user_data = {}
    with open(file_path, 'r') as file:
        lines = file.readlines()
        i = 0
        while i < len(lines):
            user_info = lines[i].strip().split('|')
            user_id = int(user_info[0])
            item_count = int(user_info[1])
            i += 1

            items = []
            for _ in range(item_count):
                item_info = lines[i].strip().split()
                item_id = int(item_info[0])
                score = int(item_info[1])
                items.append([item_id, score])
                i += 1
```

```
19
20              user_data[user_id] = items
21      return user_data
```

### 1.2.2 测试数据

输入文件为"data/test.txt"，包含用户与需要预测分数的物品。预处理将其转化为一个字典，形式为：{user_id:[item_id,...],...}。

测试数据预处理代码

```
1  def read_test_data(file_path):
2      user_items = {}
3      with open(file_path, 'r') as file:
4          lines = file.readlines()
5          i = 0
6          while i < len(lines):
7              user_info = lines[i].strip().split('|')
8              user_id = int(user_info[0])
9              item_count = int(user_info[1])
10             i += 1
11
12             items = []
13             for _ in range(item_count):
14                 item_id = int(lines[i])
15                 items.append(item_id)
16                 i += 1
17
18             user_items[user_id] = items
19     return user_items
```

# 2 基于 MLP 的推荐方法

## 2.1 算法介绍

MLP(Multilayer Perceptron) 算法是一种基于神经网络的机器学习算法。MLP 由多个神经元组成的多层神经网络构成，包括输入层、隐藏层和输出层。每个神经元都与前一层的所有神经元连接，并通过权重来调整连接的强度。其的核心思想是，通过前向传播和反向传播来训练神经网络，以学习输入和输出之间的复杂映射关系

MLP 的训练过程如下：

1. 前向传播:

   输入数据传递给网络的输入层，并通过隐藏层传播到输出层，得到网络的输出值。其中每一层的输出 $z$ 和激活函数 $\sigma$ 计算方法为：

$$z^{(l)} = w^{(l)} \cdot a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \sigma(z^{(l)})$$

其中，$l$ 表示神经网络的第 $l$ 层；$a^{(l)}$ 是第 $l$ 层的激活值；$w^{(l)}$ 是第 $l$ 层的权重矩阵；$b^{(l)}$ 是第 $l$ 层的偏置向量。

2. 损失计算:

   损失函数用于衡量模型预测值与真实标签之间的差异。常见的损失函数包括均方误差、交叉熵损失等。

3. 反向传播:

   反向传播算法用于计算损失函数对模型参数（权重和偏置）的梯度，并使用梯度下降法来更新模型参数。反向传播的主要步骤如下：

   - 计算输出层的误差 $\delta^{(L)}$：
   $$\delta^{(L)} = \nabla_a Loss \cdot \sigma'(z^{(L)})$$
   - 计算隐藏层的误差 $\delta^{(l)}$：
   $$\delta^{(l)} = ((w^{(l+1)})^T \cdot \delta^{(l+1)}) \odot \sigma'(z^{(l)})$$
   - 计算权重和偏置的梯度：
   $$\frac{\partial Loss}{\partial w^{(l)}} = \delta^{(l)} \cdot (a^{(l-1)})^T$$
   $$\frac{\partial Loss}{\partial b^{(l)}} = \delta^{(l)}$$
   - 使用梯度下降法更新权重和偏置：
   $$w^{(l)} \leftarrow w^{(l)} - \eta \cdot \frac{\partial Loss}{\partial w^{(l)}}$$
   $$b^{(l)} \leftarrow b^{(l)} - \eta \cdot \frac{\partial Loss}{\partial b^{(l)}}$$

   其中，$\nabla_a Loss$ 表示损失函数对模型输出的梯度，$\sigma'$ 表示激活函数的导数，$\odot$ 表示逐元素相乘，$\eta$ 表示学习率。

4. 迭代训练:

   重复以上步骤，不断地进行前向传播、损失计算和反向传播，直到网络的性能收敛或达到预定的迭代次数。

MLP 作为一种深度神经网络结构，具有强大的非线性建模能力。推荐系统中的用户行为数据往往具有复杂的非线性关系，MLP 可以通过多层神经网络的组合非线性激活函数来捕捉这种复杂的关系，从而更好地学习用户和物品之间的潜在特征。同时 MLP 可以实现端到端的学习，直接从原始数据中学习特征表示和预测模型，而无需手工设计特征，这使得 MLP 能够自动地从用户行为数据中学习到更具代表性和可区分性的特征，从而提高了推荐系统的性能。

## 2.2 关键代码说明

1. **训练数据处理**

   为了训练 MLP,将经处理得到的字典形式的训练数据转化为一个列表的形式：[[user_id,item_id,score],...]

MLP 训练数据处理

```
def process_train_data(data_dict):
    res = []
    for user_id in list(data_dict.keys()):
        values = data_dict[user_id]
        # print(values)
        for value in values:
            element = [user_id]
            element.append(value[0])
            element.append(value[1])
            res.append(element)

    return res
```

2. **属性数据处理**

为了训练 MLP，并减小内存的消耗，我们首先将”data/itemAttribute.txt” 中的各个属性值映射为从 1 开始的编号，属性值为 None 则映射到编号 0。接着，我们处理得到由物品到属性编号的映射关系，并用字典储存，形式为：{item_id:[attr1_id,attr2_id],...}

MLP 属性数据处理

```
def read_attr_data(file_path):
    attr2id = dict()
    item2attr = dict()
    with open(file_path, 'r') as file:
        lines = file.readlines()
        i = 0
        while i < len(lines):
            item_info = lines[i].strip().split('|')
            if item_info[1] not in attr2id:
                attr2id[item_info[1]] = len(attr2id) + 1
            if item_info[2] not in attr2id:
                attr2id[item_info[2]] = len(attr2id) + 1
            i += 1

        j = 0
        while j < len(lines):
            item_info = lines[j].strip().split('|')
            item2attr[int(item_info[0])] = [attr2id[item_info[1]],
                attr2id[item_info[2]]]
            j += 1
    return item2attr, len(attr2id)
```

3. **MLP 网络架构**

本项目所使用的是一个七层神经网络。首先，通过 embedding 层的形式将用户 id，物品 id，属性 id 映射成向量，并拼接得到网络的特征输入。接着，经过两次全连接层加隐层 ReLU 激活层。然后在经过一次全连接层加输出 Sigmoid 激活层，最后得到一个分数。

MLP 网络架构

```python
class MLP(nn.Module):
    def __init__(self, n_user, d_user, n_item, d_item, n_attr, d_attr, out_fc1, out_fc2):
        super(MLP, self).__init__()
        self.embedding_user = nn.Embedding(n_user,d_user)
        self.embedding_item = nn.Embedding(n_item,d_item)
        self.embedding_attr = nn.Embedding(n_attr,d_attr)
        self.fc1 = nn.Linear(d_user + d_item + 2*d_attr, out_fc1)
        self.activate1 = nn.ReLU()
        self.fc2 = nn.Linear(out_fc1, out_fc2)
        self.activate2 = nn.ReLU()
        self.fc3 = nn.Linear(out_fc2, 1)
        self.activate3 = nn.Sigmoid()

    def forward(self, user_ids, item_ids, attr1_ids, attr2_ids):
        vec_user = self.embedding_user(user_ids)
        vec_item = self.embedding_item(item_ids)
        vec_attr1 = self.embedding_attr(attr1_ids)
        vec_attr2 = self.embedding_attr(attr2_ids)
        x = torch.cat((vec_user, vec_item, vec_attr1, vec_attr2), 1)
        x = self.activate1(self.fc1(x))
        x = self.activate2(self.fc2(x))
        x = self.activate3(self.fc3(x))
        return x*100
```

4. **MLP 网络训练**

首先，我们会划分训练数据集与验证数据集，比例为 8:2。训练时，使用均方差作为 loss function，使用 Adam 优化器进行优化，并且进行分批次训练，每一个 epoch 会使用多个批次，并遍历一次完整的训练数据集，计算损失值，并利用反向传播更新网络参数。之后，我们也会用划分得到的验证数据集进行性能的评估。

MLP 网络训练

```python
def train(batch_size, n_epoch, lr, model_dir, train_data, val_data, item2attr, model, device):
    print('\nstart training')
    model.train()
    loss_func = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    max_mse = float('inf')

    for epoch in range(n_epoch):
        print(f'Epoch {epoch+1}/{n_epoch}')
        for phase, data in [('Train', train_data), ('Valid', val_data)]:
            if phase == 'Train':
                model.train()
            else:
```

```
            model.eval()
        total_loss, total_mse = 0, 0

        for i in range(0, len(data), batch_size):
            batch = np.array(data[i:i + batch_size])
            users = torch.tensor(batch[:, 0], dtype=torch.long, device=device)
            items = torch.tensor(batch[:, 1], dtype=torch.long, device=device)
            ratings = torch.tensor(batch[:, 2], dtype=torch.float, device=device)
            attr1 = torch.tensor([item2attr.get(int(item), [0, 0])[0] for item in
                batch[:, 1]], dtype=torch.long, device=device)
            attr2 = torch.tensor([item2attr.get(int(item), [0, 0])[1] for item in
                batch[:, 1]], dtype=torch.long, device=device)

            optimizer.zero_grad()
            with torch.set_grad_enabled(phase == 'Train'):
                outputs = model(users, items, attr1, attr2).squeeze()
                loss = loss_func(outputs, ratings)
                if phase == 'Train':
                    loss.backward()
                    optimizer.step()
            mse = mean_squared_error(ratings.cpu().numpy(),
                outputs.cpu().detach().numpy())
            total_loss += loss.item()
            total_mse += mse
            print(f'[{phase} Epoch {epoch+1}, Batch {i//batch_size +
                1}/{len(data)//batch_size + 1}] Loss: {loss.item():.3f}, MSE:
                {mse:.3f}')

        print(f'\n{phase} | Loss:{total_loss/(len(data)//batch_size + 1):.5f}
            RMSE: {math.sqrt(total_mse/(len(data)//batch_size + 1)):.3f}')
        if phase == 'Valid' and total_mse < max_mse:
            max_mse = total_mse
            torch.save(model.state_dict(), f"{model_dir}/ckpt.model")
            print(f'Saving model with RMSE
                {math.sqrt(total_mse/(len(data)//batch_size + 1)):.3f}')
```

# 3 基于 ALS 的推荐方法

## 3.1 算法介绍

ALS（Alternating Least Squares）算法是一种基于矩阵分解的协同过滤推荐算法。该算法的核心思想是将用户-物品矩阵分解为两个低秩矩阵的乘积，以表示用户和物品之间的关系。

ALS 算法通过交替优化用户和物品的特征向量来实现矩阵分解。假设已知用户-物品矩阵 $R \in R^{m \times n}$，其中 m 为用户数，n 为物品数。算法初始化一个用户因子矩阵 $U$ 和物品因子矩阵 $I$。

在每次迭代中，算法固定其中一个矩阵（用户或物品），通过最小化均方误差来更新优化另一个矩阵:

- 固定 $U$，优化 $I$：

$$I = \left(U^T U\right)^{-1} U^T R$$

- 固定 $I$，优化 $U$：

$$U = \left(I^T I\right)^{-1} I^T R$$

这个过程交替进行，直到收敛或达到预定的迭代次数。在得到收敛的用户矩阵和物品矩阵后，可以使用它们来预测用户对物品的评分。

推荐系统中的用户-物品矩阵通常是非常稀疏的，ALS 算法能够有效地处理这种稀疏性，对于缺失值多的矩阵也能进行有效推断。

## 3.2 关键代码说明

0. **ALS 函数参数及返回值**

本算法代码的主体是 ALS 函数，首先介绍其参数及返回值参数：

- train_data: 训练数据，格式为 {user_id: [[item_id, score], [item_id, score], ...], ...}。
- num_factors：隐因子（特征）的数量，默认值为 10。
- num_iterations：迭代次数，默认值为 10。
- regularization：正则化参数，用于防止过拟合，默认值为 0.1。

返回值：

- user_matrix：用户特征矩阵，每行对应一个用户，每列对应一个隐因子。
- item_matrix：物品特征矩阵，每行对应一个物品，每列对应一个隐因子。
- user_to_index：用户 ID 到索引的映射。
- item_to_index：物品 ID 到索引的映射。

1. **索引映射与构建矩阵**

初始化用户和物品的隐向量矩阵，以及用户和物品的索引映射。rating_matrix 是用户-物品评分矩阵。

索引映射与构建矩阵

```
def als(train_data, num_factors=10, num_iterations=10, regularization=0.1):
    users = list(train_data.keys())
    items = {item for user_items in train_data.values() for item, _ in user_items}
    num_users = len(users)
    num_items = len(items)

    user_to_index = {user: idx for idx, user in enumerate(users)}
    item_to_index = {item: idx for idx, item in enumerate(items)}

    rating_matrix = np.zeros((num_users, num_items))
    for user, user_items in train_data.items():
        for item, rating in user_items:
            rating_matrix[user_to_index[user], item_to_index[item]] = rating
```

```
14
15      user_matrix = np.random.normal(size=(num_users, num_factors))
16      item_matrix = np.random.normal(size=(num_items, num_factors))
```

2. **用户矩阵更新**

对于每个用户 u，根据该用户的评分更新其隐向量。item_submatrix 是用户 u 已评分的物品的隐向量子集，ratings_submatrix 是相应的评分子集。通过求解线性方程组更新用户隐向量。

用户矩阵更新

```
1       for iteration in range(num_iterations):
2           for u in range(num_users):
3               user_ratings = rating_matrix[u, :]
4               non_zero_items = user_ratings.nonzero()[0]
5               item_submatrix = item_matrix[non_zero_items, :]
6               ratings_submatrix = user_ratings[non_zero_items]
7               A = item_submatrix.T @ item_submatrix + regularization *
                    np.eye(num_factors)
8               b = item_submatrix.T @ ratings_submatrix
9               user_matrix[u, :] = np.linalg.solve(A, b)
```

3. **物品矩阵更新**

对于每个物品 i，根据该物品的评分更新其隐向量。user_submatrix 是物品 i 已被评分的用户的隐向量子集，ratings_submatrix 是相应的评分子集。通过求解线性方程组更新物品隐向量。

物品矩阵更新

```
1           for i in range(num_items):
2               item_ratings = rating_matrix[:, i]
3               non_zero_users = item_ratings.nonzero()[0]
4               user_submatrix = user_matrix[non_zero_users, :]
5               ratings_submatrix = item_ratings[non_zero_users]
6               A = user_submatrix.T @ user_submatrix + regularization *
                    np.eye(num_factors)
7               b = user_submatrix.T @ ratings_submatrix
8               item_matrix[i, :] = np.linalg.solve(A, b)
```

4. **RMSE 计算和内存管理**

在每次迭代后，计算当前的预测评分矩阵并计算 RMSE，以评估模型性能。删除预测矩阵并调用垃圾回收，以管理内存使用。

RMSE 计算和内存管理

```
1           prediction_matrix = user_matrix @ item_matrix.T
2           rmse = calculate_rmse(rating_matrix, prediction_matrix)
3           print(f"Iteration {iteration + 1}/{num_iterations}, RMSE: {rmse}")
4
5           # 清理不再需要的内存并调用垃圾回收
```

```
        del prediction_matrix
        gc.collect()
```

# 4 实验结果

我们使用基于 MLP 的推荐算法和基于 ALS 的推荐算法进行了模型训练，最终计算了其误差，对算法的性能进行了评估。其中基于 MLP 的算法额外使用了 itemAttribute.txt 中提供的物品属性。

实验使用均方根误差（RMSE）来衡量预测值与真实值之间的误差，其计算公式如下：

$$\mathrm{RMSE}(X,h)=\sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(x_i\right)-y_i\right)^2}$$

实验结果如下所示：

| 实验结果 | MLP | ALS |
|---|---|---|
| RMSE | 27.220 | 12.885 |

图 4.1: MLP 与 ALS 在训练集上的实验结果

# 5 结果分析

根据上面表格中提到的 RMSE 值，可以判断 ALS 在此任务上表现更佳。但是观察其预测结果可以发现，ALS 的预测结果中部分 Item 的评分为 None，而 MLP 对于所有的 Item 都得到了评分。这是因为这些 item 在 train.txt 中没有出现过，而 ALS 仅根据训练集得到的模型，因此无法给予这些 item 评分。

而 MLP 额外还是用了 iterAttribute.txt 物品特征数据集，即使 train.txt 中没有这个 Item 的评分，依旧可以通过其特征判断该用户对他的喜爱。所以在这一点上，MLP 模型是更优秀的。

**ALS 模型**

原理：通过交替最小二乘法迭代优化用户和物品的隐含特征向量来逐渐逼近原始评分矩阵。

优点：像本次实验的大规模数据，ALS 模型可以并行计算每个用户和物品的特征向量；对于非常稀疏的评价矩阵，他可以自适应地适应正则化模型，减轻过拟合问题。

缺点：捕捉复杂关系的能力有限。

**MLP 模型**

优点：能够捕捉复杂的非线性关系，有更强的学习能力，理论上可以获得更好的预测效果。

缺点：训练慢、资源消耗大、容易过拟合，实验中效果不如 ALS。

根据实验结果和上述分析，可以得出结论：在当前的数据集和设置下，ALS 模型更适合用于该推荐系统任务。然而，如果能够对 MLP 模型进行更好的超参数调整、增加数据量或者改进网络结构，MLP 可能会在复杂的推荐场景中展现出更强的能力。