华东师范大学数据科学与工程学院实验报告

课程名称：数据科学与工程算法基础	**年级**：2020	**上机实践成绩**：
指导教师：高明	**姓名**：李昕原	**学号**：10205501425
上机实践名称：矩阵分解		**上机实践日期**：12.1
上机实践编号：	**组号**：	**上机实践时间**：

一、实验目的

给定一个用户评分矩阵 $R_{m\times n}$，其中 m 为用户（user）的数量，n 为物品（item）的数量。矩阵元素 $r_{ij}\in R$ 表示用户 u_i 为物品 v_j 的评分值。任务目标有两个：

- 通过矩阵分解和凸优化技术，获得每个用户 u_i 和物品 v_j 的隐式向量，分别记作 $u_i\in R_k$ 和 $v_i\in R_k$，其中 k 为向量维度；所有用户和物品分别组成矩阵 $P\in R_{m\times k}$ 和 $Q\in R_{n\times k}$；
- 根据获得的用户和物品向量，预测该用户对某个物品的偏好程度 $r\hat{}_{ij}=u_iv_i^T$；

因为在实际应用中，这个用户评分矩阵式稀疏的。例如某电影网站一共有 100k 用户和 10k 部电影，有一些用户可能只看不超过 10 部电影，或者有些的电影可能被观影的人数很少。即这个用户评分矩阵存在大量的缺失值，因此通过矩阵分解可以先挖掘出已存在的用户行为，再预测这些缺失值。

二、实验任务

（1） 实现矩阵分解推荐系统项目，包括数据读取、模型训练、优化算法、模型验证与测试；

- 实现随机梯度下降和批量梯度下降算法，完成 P 和 Q 矩阵的参数更新；
- 进一步优化算法，包括正则化、偏置项和协同过滤；
- 鼓励使用或自研其他算法技术来进一步提升矩阵分解的泛化性能；

（2）完成代码后，将训练好的模型，在测试集上进行预测，并提交至榜单；

（3）完成实验报告，报告内容要求：

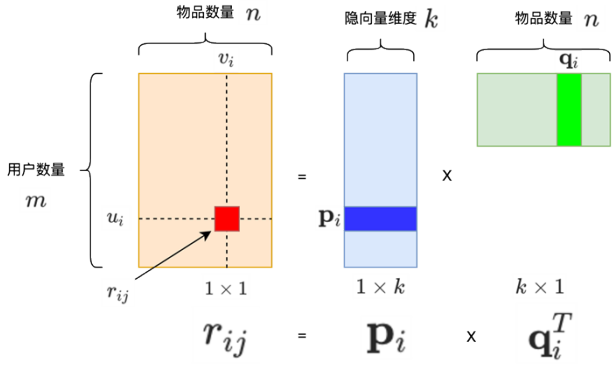
- 对矩阵分解在推荐系统的应用背景介绍、个人理解等；
- 几种优化算法、以及改进算法的代码实现原理，以及对应的实验对比结果（即验证集和测试集的结果）；
- 探索矩阵分解算法的一些优势和缺点；
- 项目完成感悟和总结；

三、实验过程

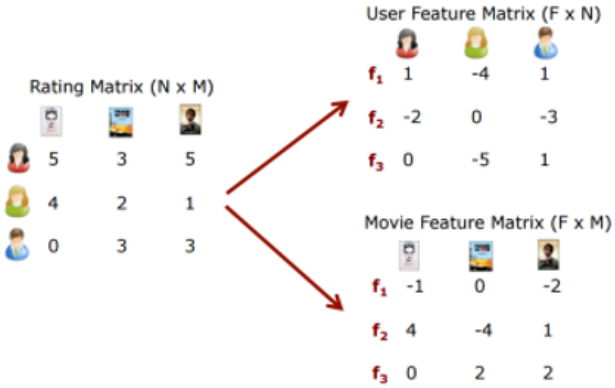
1.应用背景与个人理解

随着用户数据的发展与扩充，个性化推荐系统的实现成为当前数据应用的重点，对于每个用户需要找到其可能感兴趣的商品或服务，因此个性化推荐系统实际上是在一个显式或隐式空间中对用户兴趣和待推荐对象进行比较。

矩阵分解是推荐系统的常见实现方法之一，用户购买或评论项目的行为可以建模成一个矩阵R，通过将该矩阵进行分解，将用户兴趣和待推荐项目映射到一个 k 维的隐空间中，通过比较用户兴趣向量和待推荐项目向量确定用户对待推荐项目的感兴趣程度。从而对预测感兴趣程度较高的项目进行推荐。其中所涉及的隐因子模型通过根据评分模式推断出的因子来表征商品和用户，从而解释评分，矩阵分解模型将用户和物品映射到一个维数为 k 的联合隐因子空间，用户与物品的交互被建模为该空间的内积。每个项目、用户均与一个向量对应，两者点乘的结果即对应该用户和项目的交互，即用户对项目特征的总体兴趣，近似于用户对项目的评分，推荐系统完成映射后就可以估计用户对任何项目的评价。



通俗理解并结合具体的电影评分情景而言，当我们知道了一个用户对于很多电影的评分后，便可以推知该用户喜欢何种类型的电影，而得知一部电影的诸多用户评分后，我们也能大概推知该电影的受众与质量等属性，而两者结合，便可预测该用户对于该电影的喜好程度（假设用户并未看过此电影），通过对该用户未看过的电影进行此类预测评分，并根据预测评分的高低对用户进行推荐，便可以实现个性化的推荐系统。更多地，还可以计算两个用户对于电影的偏好相似度，两个电影偏好相似的用户也会倾向于对同一部电影得出近似相同的评分，进而通过其中一个用户的电影评分预测另一个用户的电影评分，即协同过滤。



但通过用户-评分矩阵进行矩阵分解来进行评分预测及个性化推荐同样有缺陷：一是冷启动问题，对于新用户而言，缺少可以用于评分预测及个性化推荐的有效数据；二是特征不足问题，仅通过评分来衡量用户的感兴趣程度并做个性化推荐过于简化，实际情况下应考虑更多的用户特征（如性别、身份、喜好等）来进行预测（**详细的矩阵分解优缺点讨论将在实验分析中给出**）。

2.随机梯度下降与批量梯度下降

读取数据并对 PQ 矩阵初始化，进行模型训练，将经过优化算法后得到的 PQ 矩阵通过验证集进行验证，保存训练中效果最好的 PQ 矩阵及 RMSE 值，其中优化算法包括随机梯度下降与批量梯度下降算法。

随机梯度下降算法在计算梯度时随机从数据集中愁绪一个样本 $\mathbf{x_i}$ 用于计算并更新梯度，用单个样本的梯度近似所有样本的梯度的均值，从而减少迭代时间，加快参数的更新速度。

算法 10.2　矩阵分解算法

输入: 评分矩阵 $\boldsymbol{R}$, 用户集合 $\boldsymbol{U}$, 项目集合 $\boldsymbol{V}$, 学习率 ϵ, 隐空间维度 k

输出: $\boldsymbol{P} \in \boldsymbol{R}^{|U| \times k}, \boldsymbol{Q} \in \boldsymbol{R}^{k \times |V|}$

while 没有达到停止准则 **do**
- 计算梯度:
- $g_1 \leftarrow \nabla_{p_{uj}} = -\sum_{i:(u,i)\in K} e_{ui} q_{ji}$;
- $g_2 \leftarrow \nabla_{q_{ji}} = -\sum_{u:(u,i)\in K} e_{ui} p_{uj}$;
- 参数更新:
- $p_{uj} \leftarrow p_{uj} - \epsilon g_1$;
- $q_{ji} \leftarrow q_{ji} - \epsilon g_2$;

return $\boldsymbol{P}$,$\boldsymbol{Q}$;

代码如下：

```
def sgd(self, P, Q):
        '''
        ********************************
        基本分：请实现【随机梯度下降】优化
        加分项：进一步优化如下
        - 考虑偏置项
        - 考虑正则化
        - 考虑协同过滤
        ********************************
        '''
        for uid, iid, real_rating in self.train_data.itertuples(index=False):
            p_u, q_i = P[uid], Q[iid]
            err = np.float32(real_rating - np.dot(p_u, q_i))
            p_u += self.lr * err * q_i
            q_i += self.lr * err * p_u
            P[uid] = p_u
            Q[iid] = q_i
        return P, Q
```

随机梯度下降算法会使梯度方向变化很大，难以很快收敛到全局最优解，因此可以采用折中的方法批量梯度下降法。批量梯度下降算法需要规定批大小 batch_size，然后将训练数据集分割成数个 batch_size 大小的小批量样本集，用小批量中的样本梯度均值近似所有样本的梯度均值。

算法 10.1　小批量梯度下降算法

输入: 初始学习率 ϵ_0, 训练数据集 T, 批量大小 m, 轮次 epoch

输出: 参数向量 $\boldsymbol{w}$

初始化: 参数向量 $\boldsymbol{w}$;

$k \leftarrow 0$;

while $k <$ epoch or 模型收敛 **do**
- 将训练集随机划分成若干互不重叠、大小为 m 的小批量样本集;
- **for** 每个小批量样本集 **do**
  - 计算梯度估计: $\hat{g} \leftarrow \frac{1}{m} \sum_i \nabla_{\boldsymbol{w}} L(\boldsymbol{x}_i, y_i; \boldsymbol{w})$;
  - 应用更新: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \hat{\boldsymbol{g}}$;
- 更新学习率: $\epsilon_{k+1} \leftarrow$ 更新学习率 ϵ_k;
- $k \leftarrow k + 1$;

return $\boldsymbol{w}$;

```
def bgd(self, P, Q, batch_size: int=8):
        '''
        **********************************
        基本分：请实现【批量梯度下降】优化
        加分项：进一步优化如下
        - 考虑偏置项
        - 考虑正则化
        - 考虑协同过滤
        **********************************
        '''
        #训练集划分成若干互不重叠且大小为batch_size 的批量数据集
        num_batches = len(self.train_data) // batch_size
        for i in range(num_batches):
            batch_start = i * batch_size
            batch_end = (i + 1) * batch_size
            data = self.train_data[batch_start : batch_end]
            for uid, iid, real_rating in data.itertuples(index=False):
                p_u, q_i = P[uid], Q[iid]
                err = np.float32(real_rating - np.dot(p_u, q_i))
                p_u += self.lr * err * q_i
                q_i += self.lr * err * p_u
                P[uid] = p_u
                Q[iid] = q_i
        return P, Q
```

3.正则化

通过随机梯度下降与批量梯度下降优化后，计算 RMSE 值，发现 RMSE 值偏大，因此需要对模型进行优化（不同优化的结果对比将在**实验分析**中给出）。由于模型存在大量待估计参数，易出现过拟合情况，因此考虑加入正则项来提高模型的泛化能力。采用 L_2 范数作为正则化项，将参数压缩到 0 附近来降低模型的复杂度，从而避免模型过拟合。更改矩阵分解的目标函数，设置正则化系数后，在计算梯度时加入正则化项：

```
def sgd(self, P, Q):
        '''
        **********************************
        基本分：请实现【随机梯度下降】优化
        加分项：进一步优化如下
        - 考虑正则化
        **********************************
        '''
        for uid, iid, real_rating in self.train_data.itertuples(index=False):
            p_u, q_i = P[uid], Q[iid]
            err = np.float32(real_rating - np.dot(p_u, q_i))
            p_u += self.lr * (err * q_i - self.reg_p * p_u)
            q_i += self.lr * (err * p_u - self.reg_q * q_i)
            P[uid] = p_u
            Q[iid] = q_i
        return P, Q

    def bgd(self, P, Q, batch_size: int=8):
        '''
```

```
        **********
        基本分：请实现【批量梯度下降】优化
        加分项：进一步优化如下
        - 考虑正则化
        **********
        '''
        #训练集划分成若干互不重叠且大小为batch_size 的批量数据集
        num_batches = len(self.train_data) // batch_size
        for i in range(num_batches):
            batch_start = i * batch_size
            batch_end = (i + 1) * batch_size
            data = self.train_data[batch_start : batch_end]
            for uid, iid, real_rating in data.itertuples(index=False):
                p_u, q_i = P[uid], Q[iid]
                err = np.float32(real_rating - np.dot(p_u, q_i))
                p_u += self.lr * (err * q_i - self.reg_p * p_u)
                q_i += self.lr * (err * p_u - self.reg_q * q_i)
                P[uid] = p_u
                Q[iid] = q_i
        return P, Q
```

加入正则化项后 RMSE 有明显降低（具体数据及对比分析将在实验分析中给出）。

4.考虑偏置项

当某用户对所有电影的评分都偏高时，那该用户倾向于对新电影的评分也会较高，即存在正的偏置项，而偏置部分在推荐系统中能有效提高准确率。考虑三种偏置信息，偏置部分表示为：

$$b_{ui} = \mu + b_u + d_i$$

其中 μ 为所有评分的平均值，b_u 表示用户 u 的评分偏置，d_i 表示项目 i 得到的评分偏置，算法及代码如下：

算法 10.4　考虑偏置项的矩阵分解算法

输入: 评分矩阵 $\boldsymbol{R}$, 用户集合 U, 项目集合 V, 学习率 ϵ, 用户的偏置信息 $\boldsymbol{b}$, 项目的偏置信息 $\boldsymbol{d}$, 隐空间维度 k

输出: $\boldsymbol{P} \in \boldsymbol{R}^{|U| \times k}$, $\boldsymbol{Q} \in \boldsymbol{R}^{k \times |V|}$, $\boldsymbol{b} \in \boldsymbol{R}^{|U|}$, $\boldsymbol{d} \in \boldsymbol{R}^{|V|}$

while 没有达到停止准则 **do**

  计算梯度:

  $g_1 \leftarrow \nabla_{p_{uj}} = -\sum_{i:(u,i)\in K} e_{ui} q_{ji} + \lambda p_{uj}$;

  $g_2 \leftarrow \nabla_{q_{ji}} = -\sum_{u:(u,i)\in K} e_{ui} p_{uj} + \lambda q_{ji}$;

  $g_3 \leftarrow \nabla_{b_u} = -\sum_{i:(u,i)\in K} e_{ui} + \lambda b_u$;

  $g_4 \leftarrow \nabla_{d_i} = -\sum_{u:(u,i)\in K} e_{ui} + \lambda d_i$;

  参数更新:

  $p_{uj} \leftarrow p_{uj} - \epsilon g_1$;

  $q_{ji} \leftarrow q_{ji} - \epsilon g_2$;

  $b_u \leftarrow b_u - \epsilon g_3$;

  $d_i \leftarrow d_i - \epsilon g_4$;

return $\boldsymbol{P}, \boldsymbol{Q}$;

```
def sgd(self, P, Q, bu, bi):
        '''
        **********
        基本分：请实现【随机梯度下降】优化
        加分项：进一步优化如下
        - 考虑偏置项
        - 考虑正则化
```

```
        - 考虑协同过滤
        ********************************
        '''
        for uid, iid, real_rating in self.train_data.itertuples(index=False):
            p_u, q_i = P[uid], Q[iid]
            err = np.float32(real_rating - self.globalMean - bu[uid] - bi[iid] -
                             np.dot(p_u, q_i))
            p_u += self.lr * (err * q_i - self.reg_p * p_u)
            q_i += self.lr * (err * p_u - self.reg_q * q_i)
            P[uid] = p_u
            Q[iid] = q_i
            bu[uid] += self.lr * (err - self.reg_b * bu[uid])
            bi[iid] += self.lr * (err - self.reg_b * bi[iid])
        return P, Q, bu, bi


    def bgd(self, P, Q, bu, bi, batch_size: int=8):
        '''
        ********************************
        基本分：请实现【批量梯度下降】优化
        加分项：进一步优化如下
        - 考虑偏置项
        - 考虑正则化
        - 考虑协同过滤
        ********************************
        '''
    #训练集划分成若干互不重叠且大小为batch_size的批量数据集
        num_batches = len(self.train_data) // batch_size
        for i in range(num_batches):
            batch_start = i * batch_size
            batch_end = (i + 1) * batch_size
            data = self.train_data[batch_start : batch_end]
            for uid, iid, real_rating in data.itertuples(index=False):
                p_u, q_i = P[uid], Q[iid]
                err = np.float32(real_rating - self.globalMean - bu[uid] - bi[iid] -
                                 np.dot(p_u, q_i))
                p_u += self.lr * (err * q_i - self.reg_p * p_u) / batch_size
                q_i += self.lr * (err * p_u - self.reg_q * q_i) / batch_size
                P[uid] = p_u
                Q[iid] = q_i
                bu[uid] += self.lr * (err - self.reg_b * bu[uid]) / batch_size
                bi[iid] += self.lr * (err - self.reg_b * bi[iid]) / batch_size
        return P, Q, bu, bi
```

此外预测部分也需加入偏置项：

```
def predict_user_item_rating(self, uid, iid, P, Q, bu, bi):
        # 如果uid或iid不在，我们使用全剧平均分作为预测结果返回
        if uid not in self.users_ratings.index or iid not in self.items_ratings.index:
            return self.globalMean
        p_u = P[uid]
        q_i = Q[iid]
```

```
        return self.globalMean + bu[uid] + bi[iid] + np.dot(p_u,q_i)
```

5.协同过滤

在电影推荐系统中，相同类型的电影可能会被优先推荐，而给用户推荐时可以先找到与其偏好相似的用户群体，将这一群体评分较高的电影推荐给该用户，即协同过滤。协同过滤算法可以通过充分挖掘用户与用户、产品与产品之间的隐式关系，从而缓解数据稀疏和冷启动问题的影响。对于用户 u，suv 表示用户 u 与用户 v 的偏好相似度，目标是相似度越高的用户在隐空间中的投影向量越近，这里偏好相似度取 pearson 相似度：

```
      self.sim = {}
       for u1, _ in self.users_ratings.iterrows():
              for u2, _ in self.users_ratings.iterrows():
                  if u1 < u2:
                      self.sim[(u1, u2)] = self.Pearson(u1, u2)


def Pearson(self, u1, u2):
       e, sigma, sigma1, sigma2 = 0, 0, 0, 0
       row1, row2 = self.users_ratings.loc[u1], self.users_ratings.loc[u2]
       mean1, mean2 = np.mean(row1['movieId'].values[0]), np.mean(row2['movieId'].values[0])
       for i, v1 in enumerate(row1['movieId'].values[0]):
           for j, v2 in enumerate(row2['movieId'].values[0]):
               if(v1 == v2):
                   e += (row1['rating'].values[0][i] - mean1) * (row2['rating'].values[0][j] - mean2)
                   sigma1 += (row1['rating'].values[0][i] - mean1) * (row1['rating'].values[0][i] - mean1)
                   sigma2 += (row2['rating'].values[0][j] - mean2) * (row2['rating'].values[0][j] - mean2)
       sigma = np.sqrt(sigma1) * np.sqrt(sigma2)
       return e / sigma
```

协同过滤算法及代码如下：

算法 10.5　基于协同过滤思想的矩阵分解算法

输入：评分矩阵 $\boldsymbol{R}$，用户集合 U，项目集合 V，学习率 ϵ，用户 u 和用户 v 的偏好相似度 s_{uv}，隐空间维度 k

输出：$\boldsymbol{P} \in \boldsymbol{R}^{|U| \times k}, \boldsymbol{Q} \in \boldsymbol{R}^{k \times |V|}$

while 没有达到停止准则 **do**
- 计算梯度：
- $g_1 \leftarrow \nabla_{p_{uj}} = -\sum_{i:(u,i)\in \mathbf{K}} e_{ui} q_{ji} + \gamma \sum_{u,v \in \mathcal{U}} s_{uv}(p_{uj} - p_{vj})$;
- $g_2 \leftarrow \nabla_{q_{ji}} = -\sum_{u:(u,i)\in \mathbf{K}} e_{ui} q_{uj}$;
- 参数更新：
- $p_{uj} \leftarrow p_{uj} - \epsilon g_1$;
- $q_{ji} \leftarrow q_{ji} - \epsilon g_2$;

return $\boldsymbol{P}, \boldsymbol{Q}$;

```
    def sgd(self, P, Q):
        '''
        ********************************
        基本分：请实现【随机梯度下降】优化
        加分项：进一步优化如下
        - 考虑偏置项
        - 考虑正则化
        - 考虑协同过滤
        ********************************
        '''
        sum_ = np.zeros(self.hidden_size)
```

```
        for key, value in self.sim.items():
            sum_ += value * (P[key[0]] - P[key[1]])

        for uid, iid, real_rating in self.train_data.itertuples(index=False):
            p_u, q_i = P[uid], Q[iid]
            err = np.float32(real_rating - np.dot(p_u, q_i))
            p_u += self.lr * (err * q_i - self.reg_p * p_u - self.gamma * sum_)
            q_i += self.lr * (err * p_u - self.reg_q * q_i)
            P[uid] = p_u
            Q[iid] = q_i
        return P, Q

    def bgd(self, P, Q, batch_size: int=8):
        '''
        ********************************
        基本分：请实现【批量梯度下降】优化
        加分项：进一步优化如下
        - 考虑偏置项
        - 考虑正则化
        - 考虑协同过滤
        ********************************
        '''
        #训练集划分成若干互不重叠且大小为batch_size的批量数据集
        sum_ = np.zeros(self.hidden_size)
        for key, value in self.sim.items():
            sum_ += value * (P[key[0]] - P[key[1]])

        num_batches = len(self.train_data) // batch_size
        for i in range(num_batches):
            batch_start = i * batch_size
            batch_end = (i + 1) * batch_size
            data = self.train_data[batch_start : batch_end]
            for uid, iid, real_rating in data.itertuples(index=False):
                p_u, q_i = P[uid], Q[iid]
                err = np.float32(real_rating - np.dot(p_u, q_i))
                p_u += self.lr * (err * q_i - self.reg_p * p_u - self.gamma * sum_)
                q_i += self.lr * (err * p_u - self.reg_q * q_i)
                P[uid] = p_u
                Q[iid] = q_i
        return P, Q
```

6.初始化及调参优化

除了正则化、添加偏置项与协同过滤外，还可以通过更好的初始化提高收敛速度，通过调参提高准确率、降低 RMSE。

好的初始化相当于提供了先验信息，可以有效提高收敛速度。起初矩阵 P 与 Q 的初始化是设置服从(0,1)均匀分布的随机值作为初始值，这里采用标准正态分布的方式重新进行初始化：

```
def _init_matrix(self):
        '''
        ********************************
        用户矩阵 P 和物品矩阵 Q 的初始化也对算法优化有一定帮助，更好的初始化相当于先验信息。
```

```
        加分项：
        - 思考初始化的一些方法，正态分布等等；
        - 其他初始化方法？
        ********************************
        '''
        # User-LF
        P = dict(zip(
            self.users_ratings.index,
            np.random.randn(len(self.users_ratings), self.hidden_size).astype(np.float32)
        ))
        # Item-LF
        Q = dict(zip(
            self.items_ratings.index,
            np.random.randn(len(self.items_ratings), self.hidden_size).astype(np.float32)
        ))

        return P, Q
```

```
Starting training ...
Epoch: 0
Current dev metric result: 1.9134452655483896
Best dev metric result: 1.9134452655483896
```

通过标准正态分布初始化后第一轮迭代的 RMSE 值相比于均匀分布偏高，但后续迭代过程中 RMSE 下降要更快，最终在同样的参数迭代下（--learning_rate 0.01 --batch_size 8 --epoch 20 --reg_p 0.025 --reg_q 0.025 --hidden_size 30 --optimizer_type SGD），两者的 RMSE 值最终如下表：

初始化方式	正态分布	均匀分布
RMSE	0.9146	0.8614

可见标准正态分布效果不如均匀分布。此外更重要的是调整模型参数进行优化，对于该模型而言有以下比较重要的参数：

• **学习率 learning_rate 与迭代次数 epoch：**学习率代表梯度下降过程中每次迭代的步伐，学习率过大虽然可以提高收敛速度，但会导致易忽略全局最小值而一直震荡无法收敛，学习率过小会导致收敛速度慢，并且易收敛到局部最小值而非全局最小值。本次实验给定的初始学习率为 0.02，考虑向学习率减小的方向调整学习率，而学习率减小后要适当增多迭代次数使其充分收敛。

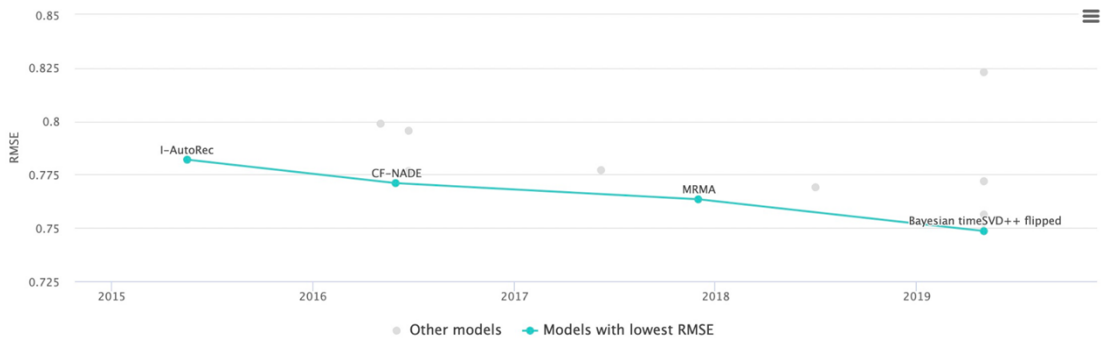
• **隐向量维度 hidden_size 与正则系数 reg：**矩阵分解中隐向量针对性地解决了推荐系统存在的主要问题，而隐向量维度 hidden_size 的大小决定了隐向量表达能力的强弱，hidden_size 较小时隐向量包含的信息较少，但模型的泛化能力强，而 hidden_size 较大时隐向量可以包含较多信息，但泛化能力较差，且 hidden_size 较大时模型的复杂度偏高。本次实验给定的初始隐向量维度为 10，考虑向 hidden_size 适当增大的方向调整学习率，同时为了限制隐向量维度增大时带来的模型复杂度高、泛化能力差等问题，将同时适当增大正则系数来防止过拟合。

• **批大小 batch_size：**对于批量梯度下降而言，batch_size 的选取非常重要，batch_size 过大会导致训练速度偏慢，时间成本偏大；而 batch_size 过小会使训练中的梯度出现较大的波动。本次实验中给定的初始 batch_size 为 8，考虑向 batch_size 增大的方向调整。

针对不同参数的调优及其对应的 RMSE 值对比将在实验分析中给出。

7.其他算法

基于矩阵分解的评分预测模型经过正则化、添加偏置项、协同过滤、初始化及参数调优后，RMSE 值最低可以达到 0.8 左右，而现有技术可以将 RMSE 降低至 0.75 以下。



这里选取 SVD++算法进行浅究，最初的 SVD 分解由于仅适用于稠密矩阵，对于实际应用中的稀疏矩阵进行分解建模，即使进行预测值填充仍会出现较大的误差，而后来改进的 RSVD 与 BiasSVD 分别进行了正则化与偏置部分，同时引入了模型训练的相关设置，有效提高了准确率。而 SVD++算法在 BiasSVD 的基础上进行了改进增强，考虑了用户的隐式反馈，即在 P_u 上添加了用户的偏好信息，相当于引入了可以侧面反映用户偏好的信息源，并可以有效解决冷启动问题。SVD++算法的预测评分及目标函数如下：

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T \left(p_u + |N_u|^{-0.5} \sum_{j \in N(u)} x_j + \sum_{a \in A(u)} y_a \right)$$

$$\min \sum_{r_{ui} \in R_{train}} \left(r_{ui} - \hat{r}_{ui} \right)^2$$

$$+ \lambda \left(||p_u||^2 + ||q_i||^2 + b_u^2 + b_i^2 + \sum_{j \in N(u)} ||x_j||^2 + \sum_{a \in A(u)} ||y_a||^2 \right)$$

其中 $N(u)$ 为用户u所产生行为的物品集合，$|Nu|^{-0.5}$ 为规范化因子，x_j为隐藏的对于商品j的个人喜好偏置，维数等于隐因子维数，每个分量代表对该商品的某一隐因子成分的偏好程度，对x_j求和实际上是将所有产生行为的商品对应的隐因子分量值分别求和得到求和后的向量，即用户对这些隐因子的偏好程度；$A(u)$ 为用户u的属性集合，y_a与x_j类似，即每个属性都对应一个隐向量。SVD++整体的优化学习算法不变，学习的参数多了向量 xy，一个是隐式反馈的物品向量，另一个是用户属性的向量。调用 surprise 包对 SVD++进行简单实现：

```
from surprise import SVDpp
from surprise import Dataset
from surprise.model_selection import cross_validate
from surprise import Reader
from surprise import accuracy
from surprise.model_selection import KFold
import pandas as pd
import numpy as np
reader = Reader(line_format='user item rating', sep=',', skip_lines=1)
data = Dataset.load_from_file('data/train.csv', reader=reader)
train_set = data.build_full_trainset()
model = SVDpp()
kf = KFold(n_splits=3) # 3 折交叉验证
for train_data, dev_data in kf.split(data):
    model.fit(train_data)
    predictions = model.test(dev_data)
    rmse = accuracy.rmse(predictions, verbose=True) #计算 RMSE
save_results = list()
dtype = [("userId", np.int32), ("movieId", np.int32)]
test_data = pd.read_csv("data/test.csv", usecols=range(1,3), dtype=dict(dtype))
test_data = pd.DataFrame(test_data)
```

```
for uid, iid in test_data.itertuples(index=False):
    pred = model.predict(uid, iid, r_ui=4, verbose=True)
    save_results.append(pred)
```

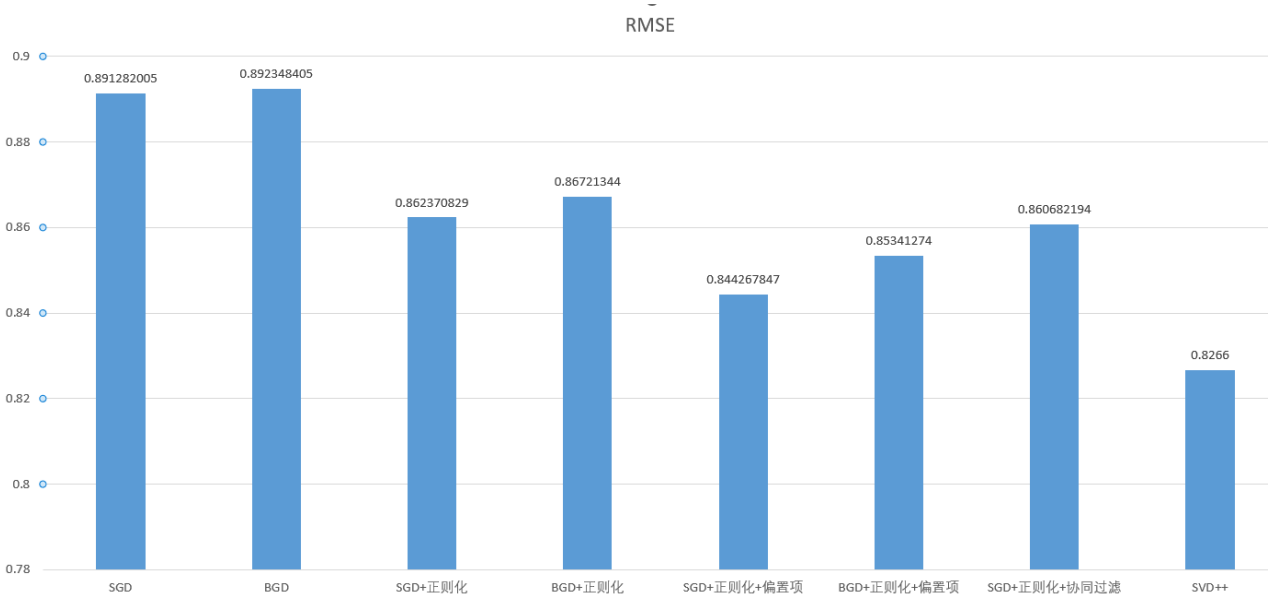
交叉验证得到的 RMSE 有明显降低：

```
RMSE: 0.8280
RMSE: 0.8281
RMSE: 0.8266
```

四、实验分析

首先使用原始给定参数（--learning_rate 0.02 --batch_size 8 --epoch 20 --reg_p 0.01 --reg_q 0.01 --hidden_size 10）对各个优化方法进行比较，分别记录随机梯度下降与批量梯度下降、正则化、正则化+偏置项、正则化+协同过滤、SVD++等情况下模型在验证集上预测的最终的 RMSE：

优化	SGD	BGD	SGD+正则化	BGD+正则化	SGD+正则化+偏置项	BGD+正则化+偏置项	SGD+正则化+协同过滤	SVD++
RMSE	0.8913	0.8923	0.8624	0.8672	0.8443	0.8534	0.8607	0.8266



由数据及图可知，随机梯度下降优化要略优于批量梯度下降优化，且随机梯度下降优化耗时较短，故后续调参以 SGD 为主。而正则化、添加偏置项与协同过滤均有效降低了验证集数据预测的 RMSE，其中 SGD+正则化+偏置项的表现最佳，而协同过滤算法由于时间复杂度偏高，未能完整迭代 20 轮，故该数据不进行讨论。横向对比情况下 SVD++算法的表现最优，这与常规认知相符合。

调参方面，首先对 BGD 的批大小 batch_size 进行调参优化，其余参数使用初始默认参数（由于 BGD 运行较慢，其余参数将使用 SGD 进行调参），记录不同 batch_size 下模型在验证集数据预测的 RMSE（BGD+正则化）：

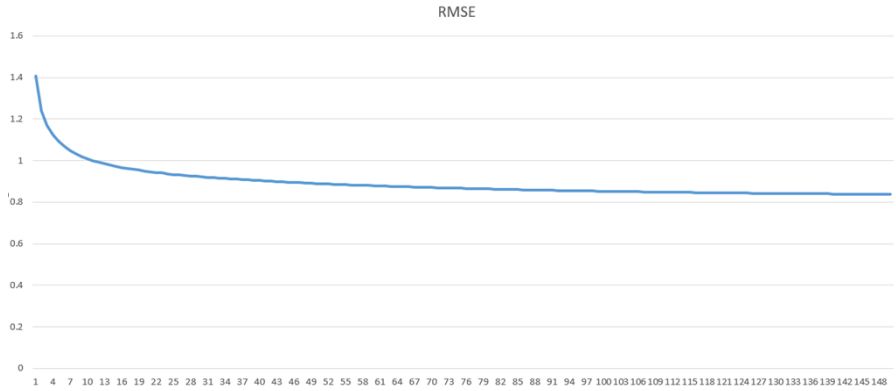
batch_size	4	8	16	32	64	128
RMSE	0.8694	0.8672	0.8646	0.8650	0.8662	0.8664

可见适当增大 batch_size 可以略微降低 RMSE，但效果不显著。其余参数的调参优化以隐向量维度 H 与学习率 L 两个参数为主，迭代次数与正则系数将根据实际情况适当增减（如学习率较低时增大

迭代次数），取调参时部分代表性参数数据记录如下（SGD+正则化）：

参数设置	H 10 L 0.02	H 15 L 0.02	H 20 L 0.02	H 25 L 0.02	H 30 L 0.02	H 40 L0.02	H 30 L0.01	H 30 L0.005	H 30 L0.001
RMSE	0.8624	0.8603	0.8569	0.8514	0.8476	0.8625	0.8412	0.8361	0.8320

设置 learning_rate=0.001，epoch=300，hidden_size=30，reg=0.025，使用随机梯度下降算法+正则化，绘制前 150 次迭代 RMSE-epoch 曲线图如下：



将该次的运行结果提交至 kaggle，取得 score 为 0.81953，截至实验报告完成前暂居第一：

#	Team	Members	Score	Entries	Last	Code
1	李昕原		0.81953	6	5m	

Your Best Entry!
Your most recent submission scored 0.81953, which is an improvement of your previous score of 0.82102. Great job!

Tweet this

综上通过实验数据对比分析了多种优化方式的实际效果及调参效果，现分析矩阵分解算法的优势及缺点。

优势：

- 应用面广，基于矩阵分解的协同过滤算法被广泛应用于各类推荐系统中，此外对相似度矩阵进行分解的多维尺度分析在信息可视化、统计分析和商业智能等领域被作为标准的降维方法；
- 解决了一般协同过滤算法对稀疏矩阵处理能力弱、泛化能力弱这一问题，矩阵分解模型的泛化能力有很大提升，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；
- 隐语义学习思路优秀且具有高可扩展性，矩阵分解算法是众多高级推荐算法的基础，一些推荐系统中的重要技术如 Embedding、FM 因子分解机及张量分解均体现了矩阵分解算法及其涵盖的隐语义思路。
- 具有高灵活性，通过矩阵分解产生的用户向量、物品向量可以与其他特征相结合，也可以与深度学习神经网络相结合，可以用多层神经网络取代向量内积从而增强表达能力；
- 空间复杂度低，在挖掘深层关联信息的同时可以起到“降维”的作用，相比于一般协同过滤算法大大降低了空间复杂度。

缺点：

- 冷启动问题，仅通过矩阵分解难以解决历史数据缺失引起的冷启动问题；
- 造成信息损失，矩阵分解算法仅依赖于用户的历史行为数据，一定程度上会造成信息损失，且难以处理用户的实时反馈；
- 时间复杂度较高，模型训练耗时长；
- 可解释性差，隐因子空间的隐含特征难以用现实概念描述。
- 仅使用内积这一简单操作来拟合用户兴趣，难以拟合非线性特征。

五、总结

本次实验完成了基于矩阵分解的电影评分预测，了解了基于矩阵分解的个性化推荐系统的基本原理与实现方法，实现了随机梯度下降与批量梯度下降优化，并通过正则化、添加偏置项、协同过滤、初始化优化及参数调优等多种方式对模型进行优化，有效降低了验证集的 RMSE 值。此外，还学习了 SVD++技术的基本原理并通过调包实现了简化的 SVD++模型。

本次实验的基本内容实现较易，但后续的优化手段实现、调参非常复杂，且未能对协同过滤等方法进行改善来降低时间复杂度，这是日后学习过程中需要重点关注的内容。但经过本次实验有效锻炼了自己的模型训练能力，提高了调参的直觉与效率，并对矩阵分解这一章的内容谙熟于心，对推荐系统的实现有了基本的认识与实操能力。