华东师范大学计算机科学技术系上机实践报告

课程名称: 智能推荐系统　　**年级:** 2018　　**上机实践成绩:**
指导教师: 张伟　　**姓名:** 张翼鹏　　**创新实践成绩:**
实验编号: 03　　**学号:** 10185102105　　**上机实践日期:** 2021/05/30
实验名称: Model-based Collaborative Filtering: Latent Factor Models

1 问题重述

本次实践要求使用隐变量模型对指定用户对指定物品的打分做出预测，其中给出的数据包括 train.csv 文件和 test.csv 文件，前者每行数据包含了用户、商品、打分分值和打分时间，后者为测试文件，要求将预测结果输出到该文件中。

我们在第一次大作业中已经实现了一个基于记忆的协同过滤推荐系统，也很容易看出其不足之处——对评分的预测依赖于历史数据，不论是用户还是物品，我们必须通过历史交互数据去寻找相似的个体留下的数据进行预测，但用户-物品评分矩阵是高度稀疏的，最后的预测值几乎体现不出来算法的优势。

这样的计算相似度的方式其实是没有考量物品与用户背后的关联性的，因此我们引入隐变量模型（或称“隐语义模型”，“潜在因子算法等）。

2 算法概述

隐变量模型的算法思想是：认为每个用户都有自己的偏好，同时每个物品也包含所有用户的偏好信息；那么可以认为用户对于物品的高评分体现的是物品中所包含的偏好信息恰好就是用户喜好的信息。而这个偏好信息我们是无法明显的找出的，所以我们可以认为这个就是潜在影响用户对物品评分的因子，即潜在因子。所以只要我们可以得到用户-潜在因子矩阵 Q 和物品-潜在因子矩阵 P 就可以计算出用户对于物品的评分信息。

我们可以从 Daniel Pyrathon 在 PyCon 2018 上的演讲中给的用户对电影评分的例子入手对隐变量模型有一个初步的把握。首先从原始数据中获取到用户电影评分的结果，用矩阵存储：

图 1: 用户-电影评分矩阵

我们要对’?’ 部分的评分进行预测。对于如何补全一个矩阵，历史上有过很多的研究。一个空的矩阵有很多种补全方法，而我们要找的是一种对矩阵扰动最小的补全方法。那么什么才算是对

矩阵扰动最小呢? 一般认为，如果补全后矩阵的特征值和补全之前矩阵的特征值相差不大，就算是扰动比较小。所以，最早的矩阵分解模型就是从数学上的 SVD（奇异值分解）开始的。” 给定 m 个用户和 n 个物品，和用户对物品的评分矩阵 $R \in \mathbb{R}^{m \times n}$。首先需要对评分矩阵中的缺失值进行简单地补全，比如用全局平均值，或者用户/物品平均值补全，得到补全后的矩阵 R'。接着，可以用 SVD 分解将 R' 分解成如下形式:

$$R' = U^T S V$$

其中 $U \in \mathbb{R}^{k \times m}$，$R \in \mathbb{V}^{k \times n}$ 是两个正交矩阵，$S \in \mathbb{R}^{k \times k}$ 是对角阵，对角线上的每个元素都是矩阵的奇异值。为了对 R' 进行降维，可以取最大的 f 个奇异值组成对角矩阵 S_f, 并且找到这 f 个奇异值中每个值在 U V 矩阵中对应的行和列，得到 U_f V_f, 从而可以得到一个降维后的评分矩阵:

$$R'_f = U_f^T S_f V_f$$

其中 $R'(u,i)$ 就是用户 u 对物品评分的预测值。SVD 分解是早期推荐系统研究常用的矩阵分解方法, 不过该方法具有以下缺点，因此很难在实际系统中应用：

- 该方法首先需要用一个简单的方法补全稀疏评分矩阵。一般来说，推荐系统中的评分矩阵是非常稀疏的，一般都有 95% 以上的元素是缺失的。而一旦补全, 评分矩阵就会变成一个稠密矩阵，从而使评分矩阵的存储需要非常大的空间，这种空间的需求在实际系统中是不可能接受的。

- 该方法依赖的 SVD 分解方法的计算复杂度很高，特别是在稠密的大规模矩阵上更是非常慢。一般来说，这里的 SVD 分解用于 1000 维以上的矩阵就已经非常慢了，而实际系统动辄是上千万的用户和几百万的物品，所以这一方法无法使用。如果仔细研究关于这一方法的论文可以发现，实验都是在几百个用户、几百个物品的数据集上进行的。

正是由于上面的两个缺点，SVD 分解算法提出几年后在推荐系统领域都没有得到广泛的关注。直到 2006 年 Netflix Prize 开始后，Simon Funk 提出了矩阵分解方法（后来被 Netflix Prize 的冠军 Koren 称为 Latent Factor Model，LFM）。

如果我们将评分矩阵 R 分解为两个低维矩阵相乘:

$$\hat{R} = P^T Q$$

其中 $P \in \mathbb{R}^{f \times m}$ 和 $Q \in \mathbb{R}^{f \times n}$ 是两个降维后的矩阵。那么，对于用户 u 对物品 i 的评分的预测值 $R(\hat{u,i}) = r_{\hat{ui}}$，可以通过如下公式计算:

$$r_{ui} = \sum_{\hat{f}} p_{uf} q_{if}$$

其中 $p_{uf} = P(u,f), q_{if} = Q(i,f)$。接下来可以直接通过训练集中的观察值利用最小化 RMSE 学习 P、Q 矩阵。将上面的用户-电影评分矩阵分解成两个维度 $LF1$ 和 $LF2$，如下图 2:

图 2: 评分矩阵分解

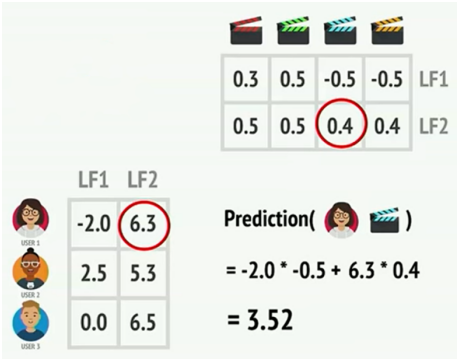

图 3: 评分预测

我们在用户和电影中分别得到了两个隐性特征，如果将这两个矩阵重新相乘，会得到与原评分矩阵相似的结果。这里只用到了两个隐性特征，事实上隐性特征可以根据需要生成，隐性特征越多，就表示从原始矩阵中提取到的信息越多，最后我们据此即可进行评分预测，见上图 3。后文中将给出基于此朴素模型的不同的算法方案。

而我们本次实验要求的 LFM 可以概括为：一种基于矩阵分解的用来预测用户对物品兴趣度的推荐算法。python 的 surprise 库中提供了相关的函数包用于训练这种模型，大大简化了代码量，在接下来的步骤中我尝试研究了三个基于矩阵分解的模型原理和源码并以 RSME 为主要指标作出了评估。

3 SVD 算法模型

3.1 算法原理

SVD 朴素模型运用到的就是我们课上讲的基础公式：

$$f(u,i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i$$

其中 α 为全局评分（也即用户-物品评分矩阵找那个所有非零值的均值），β_u 表示用户 u 的历史评分均值，β_i 表示物品 i 的历史被评分均值，$\gamma_u \cdot \gamma_i$ 表示降维后的内积。

为了估算所有未知数，我们将以下正则平方误差最小化：

$$\arg\min_{\alpha,\beta,\gamma} \sum_{u,i} (\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i})^2 + \lambda \left[\sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|_2^2 + \sum_u \|\gamma_u\|_2^2 \right]$$

如果要最小化目标函数，必须往其负梯度方向搜索。这就是梯度下降法（注意它是一个局部优化算法，也就是说有可能落到局部最优解而不是全局最优解）。通过简单的随机梯度下降来执行最小化：

$$\begin{aligned}
\beta_u &\leftarrow \beta_u + \gamma (e_{ui} - \lambda \beta_u) \\
\beta_i &\leftarrow \beta_i + \gamma (e_{ui} - \lambda \beta_i) \\
\gamma_u &\leftarrow \gamma_u + \gamma (e_{ui} \cdot \gamma_i - \lambda \gamma_u) \\
\gamma_i &\leftarrow \gamma_i + \gamma (e_{ui} \cdot \gamma_u - \lambda \gamma_i)
\end{aligned}$$

其中 e_{ui} 是真实值和预测值的差值。

3.2 源码解读

我们这里选取计算预测值部分的源码来看：

```
def estimate(self, u, i):

    known_user = self.trainset.knows_user(u)
    known_item = self.trainset.knows_item(i)

    # 有偏估计，用公式算
    if self.biased:

        # 全局平均分
        est = self.trainset.global_mean

        # 用户平均分
        if known_user:
            est += self.bu[u]

        # 物品平均分
        if known_item:
            est += self.bi[i]

        # 降维后的内积
        if known_user and known_item:
            est += np.dot(self.qi[i], self.pu[u])

    # 无偏估计，这里用不到，考虑为什么
    else:
        if known_user and known_item:
            est = np.dot(self.qi[i], self.pu[u])
        else:
            raise PredictionImpossible('User and item are unknown.')

    return est
```

可以看到源码中 biased 参数是一个很重要的量，如果把它设为 False 我们可以把它看做“全盲”的算法，也就是在我们的算法概述中讲的电影评分，单纯的分解评分矩阵。我尝试传参的时候传一个 False 值进去，结果惨不忍睹，RMSE 高达 3 左右。其实简单思考一下就可以理解原因：评分矩阵是高度稀疏的，使用无偏估计必然十分不准确，就像我们第一次实践中，由于交互信息太少，多数预测值实际上输出的就是平均值，那如果我们采用无偏估计，连平均值信息都用不上了，结果必然是十分不准确的。

3.3 核心代码

```
# 读取 train.csv 文件并初始化数据集
csv_file = "train.csv"
csv_data = pd.read_csv(csv_file, low_memory = False)
df = pd.DataFrame(csv_data)
df.drop(['date'], axis=1,inplace=True)

# 创建一个读取器，定义评分中的高低值
reader = Reader(rating_scale=(1, 5))
dataset = Dataset.load_from_df(df, reader)

# 按 8:2 划分训练集和测试集
trainset, testset = train_test_split(dataset, test_size=.2)
model = SVD()

# 将 SVD 运行到数据集上
model.fit(trainset)

# 画图参数
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus'] = False
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(xmax=1600,xmin=0)
plt.ylim(ymax=6,ymin=0)
area = np.pi * 4**2  # 点面积
colors1 = '#00CED1' # 点的颜色
colors2 = '#DC143C'

# 将模型运用到测试集上评估预测结果
size = len(testset)
sum_RMSE = 0
cnt = 1
x = []
y_real = []
y_pred = []
for item in testset:
        user = item[0]
        business = item[1]
        real = item[2]
        pred = model.predict(user, business).est
        sum_RMSE += (pred-real)**2
```

```
        x.append(cnt)
        cnt += 1
        y_real.append(real)
        y_pred.append(pred)
RMSE = (sum_RMSE/size)**0.5
print("RMSE = {:.3f}".format(RMSE))

plt.scatter(x, y_real, s=area, c=colors1, alpha=0.4, label='真实值')
plt.scatter(x, y_pred, s=area, c=colors2, alpha=0.4, label='预测值')
plt.legend(loc='lower right')
plt.savefig(r'C:\Users\lenovo\Desktop\SVD.png', dpi=300)
plt.show()
```

3.4 模型评估

如下图所示，我们通过散点图来观察一下预测效果，X 轴表示测试集中的测试点，Y 轴表示评分制，蓝色点为真实值，红色点为预测值，左图采用有偏估计，这种情况下点几乎都分布在 [3,5] 之间，这也是全局评分的效果体现，很明显 3,4（被遮挡，但从右图可以推断）,5 的条带颜色较深，因为大多数用户给出的评分在此区间。

右图为无偏估计单纯分解评分矩阵得到的结果，预测值和真实值几乎没什么联系。

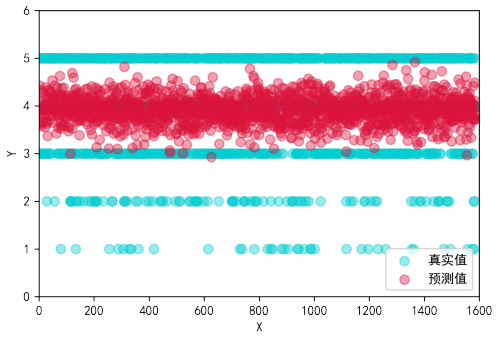

图 4: SVD-预测值-真实值对比

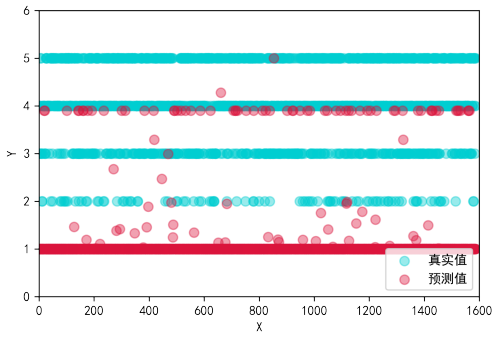

图 5: SVD-预测值-真实值对比（无偏估计）

另外，我们将这个模型的 RMSE 和运行时间进行记录：

执行次数	1	2	3	4	5	6	7
RMSE	0.975	0.957	0.978	0.988	0.986	0.936	0.962
执行时间 (sec)	1.055	0.962	0.942	0.946	0.971	1.018	0.969

和第一次的协同过滤算法比，RMSE 值有一定的下降，同时执行时间大幅度减少，模型性能不错。

4 SVD++ 算法模型

4.1 算法原理

从名称上看，SVD++ 是 SVD 的改进版，事实确实如此。实际生产中，用户评分数据往往很稀少，就像我们的评分矩阵特别稀疏，也就是说显示数据比隐式数据少很多，这些隐式数据能否加入模型呢?

SVD++ 就是在 SVD 模型中融入用户对物品的隐式行为。通俗来讲，我们可以认为评分 = 显式兴趣 + 隐式兴趣 + 偏见，也就是我们的 PPT 上扩展部分的公式:

$$f(u,i) = \alpha + \beta_u + \beta_i + \left(\gamma_u + \frac{1}{\|N(u)\|}\sum_{a\in N(u)}\rho_a\right)\cdot\gamma_i$$

其中，$\|N(u)\|$ 表示用户 u 的行为物品集，是一个归一化项，ρ_a 表示物品 a 所表达的隐式反馈。

这里没有考虑用户对集合中的打过分的物品的具体评分值，而是简单认为，只要用户历史上评价过的物品大多数都在 n_factors 维度中的某一维度的权重很高，则可以说明用户偏好在该维度上权重值高的物品。所以 SVD++ 方法的思想是同时考虑显性特征和隐性偏好，代价函数用的也是平方误差。

4.2 源码解读

依然看计算预测值部分的代码:

```
def estimate(self, u, i):
    # 全局平均分
    est = self.trainset.global_mean

    # 用户平均分，有就加
    if self.trainset.knows_user(u):
        est += self.bu[u]

    # 物品平均分，有就加
    if self.trainset.knows_item(i):
        est += self.bi[i]

    # 这里的 Iu 就是我们课上的 |N(u)|
    if self.trainset.knows_user(u) and self.trainset.knows_item(i):
        Iu = len(self.trainset.ur[u])
        u_impl_feedback = (sum(self.yj[j] for (j, _)
                               in self.trainset.ur[u]) / np.sqrt(Iu))
        est += np.dot(self.qi[i], self.pu[u] + u_impl_feedback)

    return est
```

4.3 核心代码

和上一节几乎一样，只要修改调用模型的名称：

```
# SVD++ 算法模型

# 开始计时
start = datetime.datetime.now()
# 读取 train.csv 文件并初始化数据集
csv_file = "train.csv"
csv_data = pd.read_csv(csv_file, low_memory = False)
df = pd.DataFrame(csv_data)
df.drop(['date'], axis=1,inplace=True)

# 创建一个读取器，定义评分中的高低值
reader = Reader(rating_scale=(1, 5))
dataset = Dataset.load_from_df(df, reader)

# 按 8:2 划分训练集和测试集
trainset, testset = train_test_split(dataset, test_size=.2)
model = SVDpp(n_factors=100)

# 将 SVD++ 运行到数据集上
model.fit(trainset)

# 将模型运用到测试集上并评估预测结果
size = len(testset)
sum_RMSE = 0
cnt = 1
x = []
y_real = []
y_pred = []
for item in testset:
    user = item[0]
    business = item[1]
    real = item[2]
    pred = model.predict(user, business).est
    sum_RMSE += (pred-real)**2
    x.append(cnt)
    cnt += 1
    y_real.append(real)
    y_pred.append(pred)
RMSE = (sum_RMSE/size)**0.5
print("RMSE = {:.3f}".format(RMSE))
```

```
41
42  # 画图参数设置
43  plt.rcParams['font.sans-serif']=['SimHei']
44  plt.rcParams['axes.unicode_minus'] = False
45  plt.xlabel('X')
46  plt.ylabel('Y')
47  plt.xlim(xmax=1600,xmin=0)
48  plt.ylim(ymax=6,ymin=0)
49  area = np.pi * 4**2  # 点面积
50  colors1 = '#00CED1' # 点的颜色
51  colors2 = '#DC143C'
52
53  # 绘制散点图
54  plt.scatter(x, y_real, s=area, c=colors1, alpha=0.4, label='真实值')
55  plt.scatter(x, y_pred, s=area, c=colors2, alpha=0.4, label='预测值')
56  plt.legend(loc='lower right')
57  plt.savefig(r'C:\Users\lenovo\Desktop\SVD.png', dpi=300)
58  plt.show()
59
60  # 计时结束
61  end = datetime.datetime.now()
62  print (end-start)
```

4.4 模型评估

从 RMSE 指标上看，SVD++ 并没有比 SVD 有显著优势，不过其稍有改进，认真观察对比图，这次的预测值没有完全集中在 [3,5] 之间，可信度稍有提升，但是从执行时间上看，其性能远不如 SVD 模型：

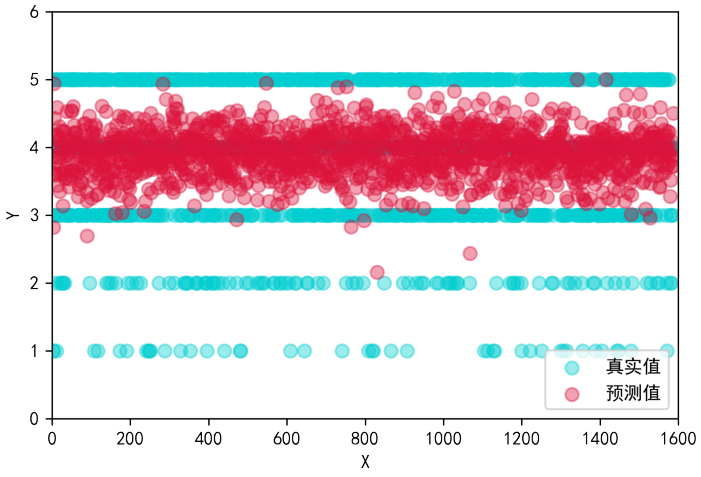

图 6: SVD++-预测值-真实值对比

执行次数	1	2	3	4	5	6	7
RMSE	0.991	0.971	0.968	1.011	0.955	0.967	0.978
执行时间 (sec)	8.587	8.519	8.301	8.219	8.580	8.430	8.548

5 NMF 算法模型

5.1 算法原理

NMF 方法是一种非负矩阵分解法，由于我们的评分矩阵的值都在 [1,5] 之间，可以考虑应用此方法。该方法中，评分预测函数如下：

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

预测公式与 SVD 方法相同，优化方法也是梯度下降法，但因为要求参数非负，所以参数的更新非常有特点：

$$p_{uf} \leftarrow p_{uf} \cdot \frac{\sum_{i \in I_u} q_{if} \cdot r_{ui}}{\sum_{i \in I_u} q_{if} \cdot \hat{r_u} + \lambda_u |I_u| p_{uf}}$$
$$q_{if} \leftarrow q_{if} \cdot \frac{\sum_{u \in U_i} p_{uf} \cdot r_{ui}}{\sum_{u \in U_i} p_{uf} \cdot \hat{r_{ui}} + \lambda_i |U_i| q_{if}}$$

这里的“非负”指的是矩阵 q 和矩阵 p 中的所有元素非负，偏置项仍然可以为负。在公式中，I_u 表示用户 u 历史上评价过的物品的集合，U_i 表示评价过物品 i 的用户集合，r_{ui} 表示预测的用户 u 对物品 i 的评分。显然，当 $\widehat{r_{ui}}$ 越接近 r_{ui} 时，更新的系数就越接近于 1，模型趋向于收敛；当 $\widehat{r_{ui}} < r_{ui}$，更新系数越大，参数增大；当 $\widehat{r_{ui}} > r_{ui}$，更新系数越小，参数减小。这里，通过对集合 I_u 的处理，考虑了用户的隐性偏好，除此之外，通过对集合 U_i 的处理还考虑了 item 侧的隐性偏好。

5.2 源码解读

```
def estimate(self, u, i):
    # 全局评分
    est = self.trainset.global_mean

    # 用户 u 历史上评价过的物品的集合
    if self.trainset.knows_user(u):
        est += self.bu[u]

    # 评价过物品 i 的用户集合
    if self.trainset.knows_item(i):
        est += self.bi[i]

    if self.trainset.knows_user(u) and self.trainset.knows_item(i):
        Iu = len(self.trainset.ur[u])  # nb of items rated by u
        u_impl_feedback = (sum(self.yj[j] for (j, _)
                               in self.trainset.ur[u]) / np.sqrt(Iu))
        est += np.dot(self.qi[i], self.pu[u] + u_impl_feedback)
```

```
18
19      return est
```

5.3 核心代码

```
1   # NMF 算法模型
2
3   # 开始计时
4   start = datetime.datetime.now()
5   # 读取 train.csv 文件并初始化数据集
6   csv_file = "train.csv"
7   csv_data = pd.read_csv(csv_file, low_memory = False)
8   df = pd.DataFrame(csv_data)
9   df.drop(['date'], axis=1,inplace=True)
10
11  # 创建一个读取器，定义评分中的高低值
12  reader = Reader(rating_scale=(1, 5))
13  dataset = Dataset.load_from_df(df, reader)
14
15  # 按 8:2 划分训练集和测试集
16  trainset, testset = train_test_split(dataset, test_size=.2)
17  model = NMF()
18
19  # 将 NMF 运行到数据集上
20  model.fit(trainset)
21
22  # 将模型运用到测试集上并评估预测结果
23  size = len(testset)
24  sum_RMSE = 0
25  cnt = 1
26  x = []
27  y_real = []
28  y_pred = []
29  for item in testset:
30      user = item[0]
31      business = item[1]
32      real = item[2]
33      pred = model.predict(user, business).est
34      sum_RMSE += (pred-real)**2
35      x.append(cnt)
36      cnt += 1
37      y_real.append(real)
```

```
    y_pred.append(pred)
RMSE = (sum_RMSE/size)**0.5
print("RMSE = {:.3f}".format(RMSE))

# 画图参数设置
plt.rcParams['font.sans-serif']=['SimHei']
plt.rcParams['axes.unicode_minus'] = False
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim(xmax=1600,xmin=0)
plt.ylim(ymax=6,ymin=0)
area = np.pi * 4**2  # 点面积
colors1 = '#00CED1' # 点的颜色
colors2 = '#DC143C'

# 绘制散点图
plt.scatter(x, y_real, s=area, c=colors1, alpha=0.4, label='真实值')
plt.scatter(x, y_pred, s=area, c=colors2, alpha=0.4, label='预测值')
plt.legend(loc='lower right')
plt.savefig(r'C:\Users\lenovo\Desktop\NMF.png', dpi=300)
plt.show()

# 计时结束
end = datetime.datetime.now()
print (end-start)
```

5.4 模型评估

模型结果中规中矩，时间比 SVD++ 有所加快但仍然不如 SVD，RMSE 值最差：

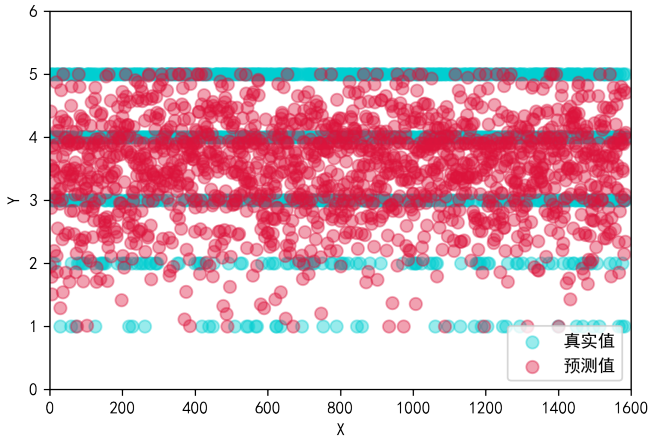

图 7: NMF-预测值-真实值对比

执行次数	1	2	3	4	5	6	7
RMSE	1.227	1.226	1.227	1.219	1.193	1.177	1.222
执行时间 (sec)	2.756	2.807	3.091	3.083	2.966	2.841	2.873

6 模型调参

通过比对以上三类模型的性能，最终选择了 SVD 模型。下面进行参数的调整。根据官方文档的说明，该模型有以下参数：

参数名称	参数说明	默认值
n_factors	The number of factors.	100
n_epochs	The number of iteration of the SGD procedure.	20
biased (bool)	Whether to use baselines (or biases).	True
init_mean	The mean of the normal distribution for factor vectors initialization.	0
init_std_dev	The standard deviation of the normal distribution for factor vectors initialization.	0.1
lr_all	The learning rate for all parameters.	0.005
reg_all	The regularization term for all parameters.	0.02
lr_bu	The learning rate for bu. Takes precedence over lr_all if set.	None
lr_bi	The learning rate for bi. Takes precedence over lr_all if set.	None
lr_pu	The learning rate for pu. Takes precedence over lr_all if set.	None
lr_qi	The learning rate for qi. Takes precedence over lr_all if set.	None
reg_bu	The regularization term for bu. Takes precedence over reg_all if set.	None
reg_bi	The regularization term for bi. Takes precedence over reg_all if set.	None
reg_pu	The regularization term for pu. Takes precedence over reg_all if set.	None
reg_qi	The regularization term for qi. Takes precedence over reg_all if set.	None
random_state	Determines the RNG that will be used for initialization. If int, random_state will be used as a seed for a new RNG. This is useful to get the same initialization over multiple calls to fit(). If RandomState instance, this same instance is used as RNG. If None, the current RNG from numpy is used.	None
verbose	If True, prints the current epoch.	False

这里我们选择 n_epochs,lr_all 和 reg_all 这三个参数进行调整：

```
# 调参

# 读取 train.csv 文件并初始化数据集
train_file = "train.csv"
train_data = pd.read_csv(train_file, low_memory = False)
train_df = pd.DataFrame(train_data)
train_df.drop(['date'], axis=1,inplace=True)
reader = Reader(rating_scale=(1, 5))
```

```
trainset = Dataset.load_from_df(train_df, reader)

# 指定参数选择范围
param_grid = {'n_epochs': [20, 50],
              'lr_all': [0.002, 0.005],
              'reg_all': [0.02, 0.04]}

gs = GridSearchCV(SVD, param_grid, measures=['rmse',], cv=3)
gs.fit(trainset)

# 打印最好的均方根误差 RMSE
print(gs.best_score['rmse'])

# 打印取得最好 RMSE 的参数集合
print(gs.best_params['rmse'])
```

其中参数的选择范围作了几次修改，用类似于二分的思想，最终得到的最优参数组合为n_epochs=50,lr_all=0.002,reg_all=0.02（默认值）。不过这里每次只能在参数中取几个固定值，将这些值固定组合，事实上有陷入局部最优解的可能。

7 运行方法

7.1 库要求

除了常见的 python 库，需要安装以下库：

- os
- numpy
- pandas
- surprise
- matplotlib

7.2 运行说明

jupyter 打开 ipynb 文件，每个模块都可以单独运行，结果已保留在文件中，可直接检查结果也可以重新运行，图片会被保存至桌面。

预测部分的核心代码如下：

```
# 评分预测

# 读取 train.csv 文件并初始化数据集
train_file = "train.csv"
test_file = "test.csv"
```

```
train_data = pd.read_csv(train_file, low_memory = False)
test_data = pd.read_csv(test_file, low_memory = False)
train_df = pd.DataFrame(train_data)
train_df.drop(['date'], axis=1,inplace=True)
f = open('test1.csv','w',encoding='utf-8',newline='')
csv_writer = csv.writer(f)
csv_writer.writerow(["user_id","business_id","date","pred_star"])
test_df = pd.read_csv(open('test.csv'))

# 创建一个读取器，定义评分中的高低值
reader = Reader(rating_scale=(1, 5))
trainset = Dataset.load_from_df(train_df, reader).build_full_trainset()

# 使用刚才最好的参数组合
model = SVD(n_factors=100, n_epochs=50, lr_all=0.002)

# 将 SVD 运行到数据集上
model.fit(trainset)

# 将模型运用到测试集上并预测结果
res = []
for index, row in test_df.iterrows():
    new_row = []
    new_row.append(row["user_id"])
    new_row.append(row["business_id"])
    new_row.append(row["date"])
    new_row.append("{:.2f}".format(model.predict(row["user_id"], row["business_id"]).est))
    res.append(new_row)

# 写入完毕
csv_writer.writerows(res)
f.close()
os.remove('test.csv')
os.rename('test1.csv', 'test.csv')
print('预测完成！')
```

8 拓展思考

随着 Netflix Prize 推荐比赛的成功举办，近年来隐变量模型受到越来越多的关注。隐语义模型最早在文本挖掘领域被提出，用于寻找文本的隐含语义，相关的模型常见的有潜在语义分析（Latent Semantic Analysis,LSA）、LDA（Latent Dirichlet Allocation）的主题模型（Topic Model）、矩阵分解（Matrix Factorization）等等。

其中矩阵分解技术是实现隐语义模型使用最广泛的一种方法，其思想也正来源于此，Yehuda Koren 凭借矩阵分解模型勇夺 Netflix Prize 推荐比赛冠军，以矩阵分解为基础，Yehuda Koren 在数据挖掘和机器学习相关的国际顶级会议（SIGIR,SIGKDD,RecSys 等）发表了很多文章，将矩阵分解模型的优势发挥得淋漓尽致。实验结果表明，在个性化推荐中使用矩阵分解模型要明显优于传统的基于邻域的协同过滤方法，如 UserCF、ItemCF 等，这也使得矩阵分解成为了目前个性化推荐研究领域中的主流模型。

不过 Yehuda Koren 团队最终获奖的模型并不是单一的模型，他们最终对几个基础模型进行了 50 多种类型的融合，将 RMSE 降到了 0.873 左右，因此模型融合可能对预测结果有一定的改良效果，但鉴于我们的训练集仅有 8000 多条数据，训练数据不足也会制约预测结果的准确性。

9 实践总结

本次实践调用了几个不同的模型并对预测结果进行了一些评估，虽然算法背后的原理涉及到矩阵分解、梯度下降等比以前模型更加复杂的数学知识，但由于 python 强大的库支持，代码实现并不难，重点在于体会隐变量模型背后的思想和原理。

参考文献

[1] 项亮. 推荐系统实践 [M]. 人民邮电出版社: 北京,2012:186.

[2] 邓日升, 岳昆, 武浩, 刘惟一. 面向商品评分预测的隐变量模型构建与推理 [J]. 小型微型计算机系统,2017,38(02):352-356.

[3] surprise 官方文档: https://surprise.readthedocs.io/

[4] Lee, Daniel & Seung, H.. (1999). Learning the Parts of Objects by Non-Negative Matrix Factorization. Nature. 401. 788-91. 10.1038/44565.