**目录**

**统计信息 ---------------------------------- 2**

**算法原理 ---------------------------------- 3**

**关键代码 ---------------------------------- 4**

**实验结果 ---------------------------------- 9**

**实验分析 ---------------------------------- 11**

**统计信息：**

训练集：

用户数量：19835

电影数量：455309

评分数量：5001507

电影编号最大值：624960

（用户数量、评分数量可由train.txt直接得到，电影数量由辅助程序n\_movies.py得到）

由此可见，用户×电影的评分矩阵是一个十分稀疏的矩阵，只处理矩阵中的非空值。

测试集：

用户数量：19835

电影数量：119010

**算法原理：**

一开始尝试了CF算法，但是因为实现得不理想，时间开销过大，转而使用了思路更简单的Latent Factor Model（简称为LFM）算法。

LFM相当于改进版的SVD算法，直接将评分矩阵R分解为两个低维矩阵相乘：

**R(m×n)=P(m×k)Q(k×n)**

其中Ρ和**Q**是两个降维后的矩阵,m是用户数量，n是电影数量，k是选取的维度数，为了便于编程计算，令**Q’(n×k)= QT(k×n)** 那么，对于用户u对电影i的评分的预测值**rui**可以通过如下公式计算：

**rui=**

LMF算法的思想很简单：可以直接利用训练集中的观察值最小化RMSE学习P、 Q矩阵。因此，定义损失函数C(P,Q),目的是使函数值最小：

**C(P,Q) = =**

直接优化上面的损失函数可能会导致学习的过拟合，因此还需要加入防止过拟合项：

λ(||P[u]||2 + ||Q’[i]||2)，其中 λ 是正则化参数，从而得到：

**C(P,Q)=+λ(||P[u]||^2 + ||Q’[i]||^2)**

下一步，利用随机梯度下降法最小化上面的损失函数，通过求参数的偏导数找到最速下降方向，然后通过迭代法不断地优化参数。上面定义的损失函数里有两组参数（P[u,k]，Q[i,k]），最速下降法需要首先对它们分别求偏导数，设**eui=R[u,i]-rui**(真实值减预测值)可以得到：

**= -2euiQik + 2λPuk**

然后，根据随机梯度下降法，需要将参数沿着最速下降方向(负梯度方向)向前推进，因此可以按照下面的公式更新P[u,k]：

**Puk = Puk + α( eui Qik - λPuk)**

同理可更新Q[i,k]:

**Qik = Qik + α( eui Puk - λQik)**

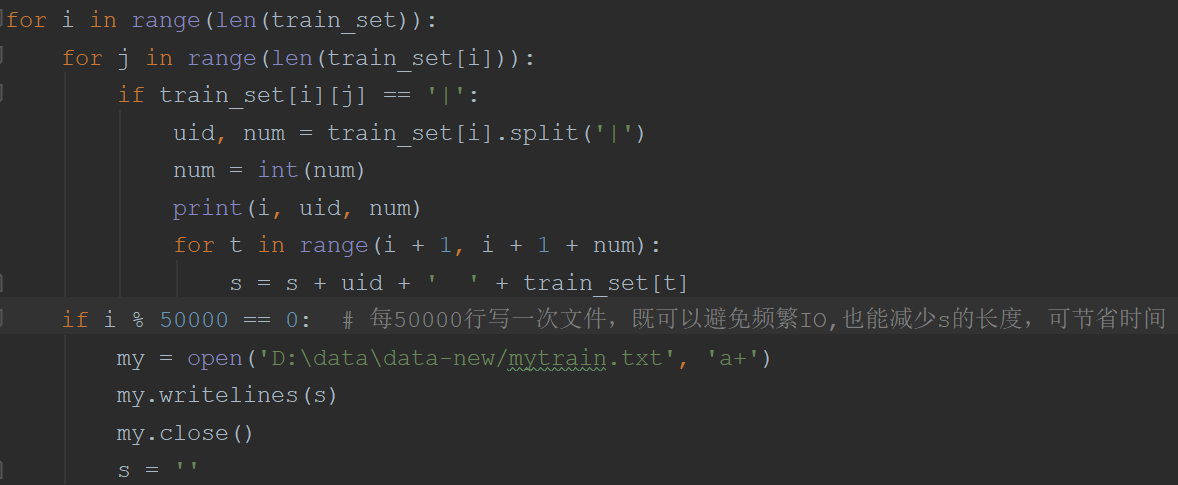
其中， α 是学习率（步长），λ 是正则化参数。

得到收敛后的P,Q便可预测原矩阵A的值了，A[i,j]=P[i]Q[j].

**关键代码：**

因为训练集很大，超过五百万条数据，所以将很多功能都拆分成小的程序了，避免都写在一起调试时间太长，也能确定已完成的功能的效果。

\*统一训练集的数据格式：



以“|”为分隔符，读取用户评分的数量，将其下面的num行都加上userid，存入新文件mytrain.txt中，将原始的训练集数据格式：

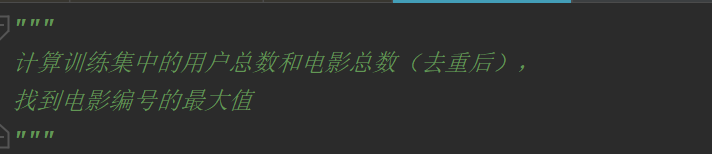
train.txt

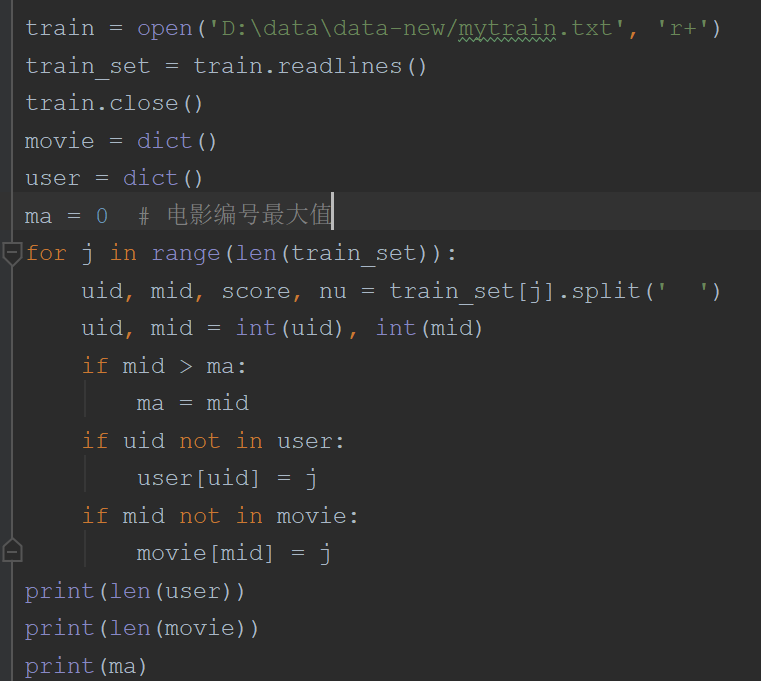
<user id>|<numbers of rating items>

<item id> <score>

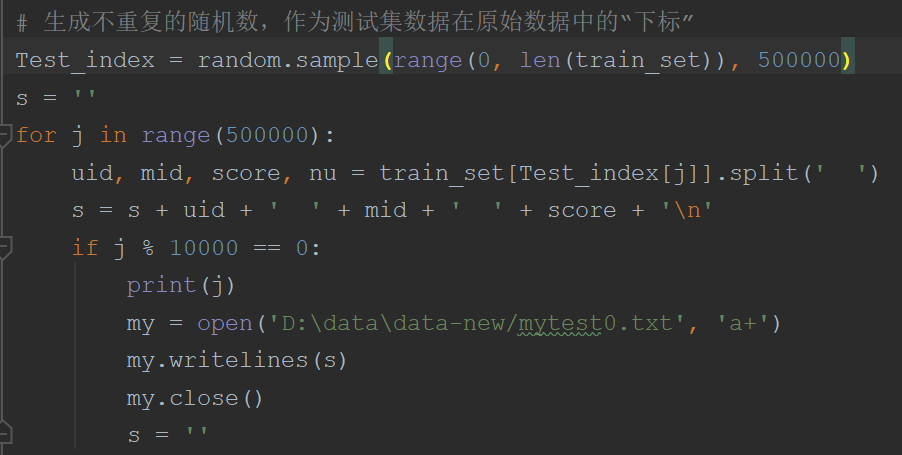
统一为 <user id> <item id> <score>的格式，方便使用。对测试集test.txt也做同样的处理。

\*计算统计信息：





\*在训练集上，大致按1：9的比例划分新的测试集和训练集，以便测试算法效果

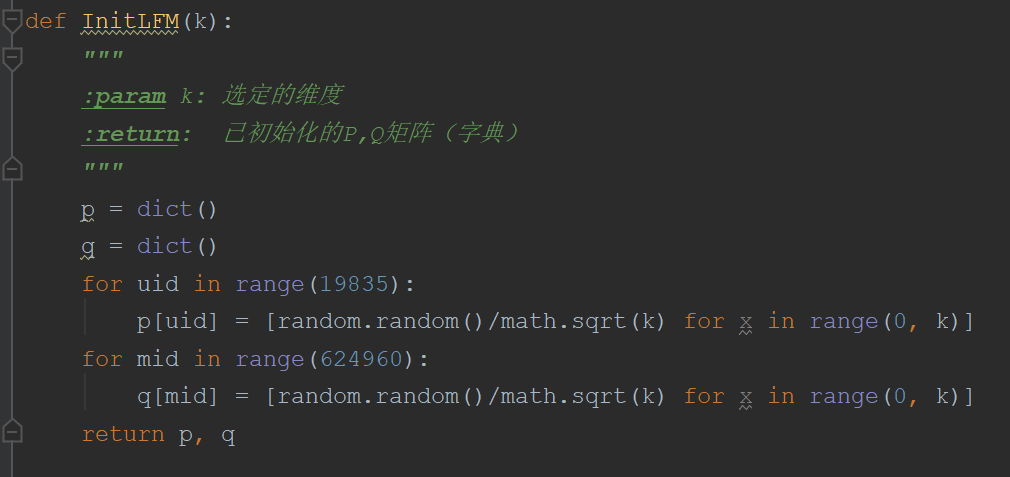


生成50万个不重复的随机数，作为测试集数据在原始数据中的“下标”，根据这些下标在原始数据中选择相应的行作为自己定义的测试集mytest0.txt

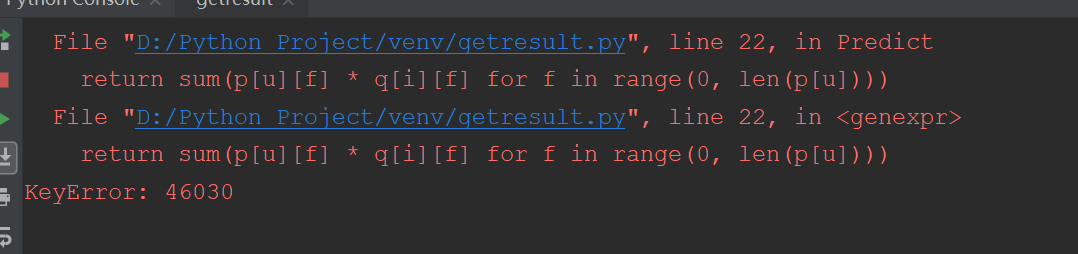


对下标数组按降序排序，按照下标从大到小的顺序，在训练集中删除相应的行，剩下的行作为自定义的训练集mytrain0.txt

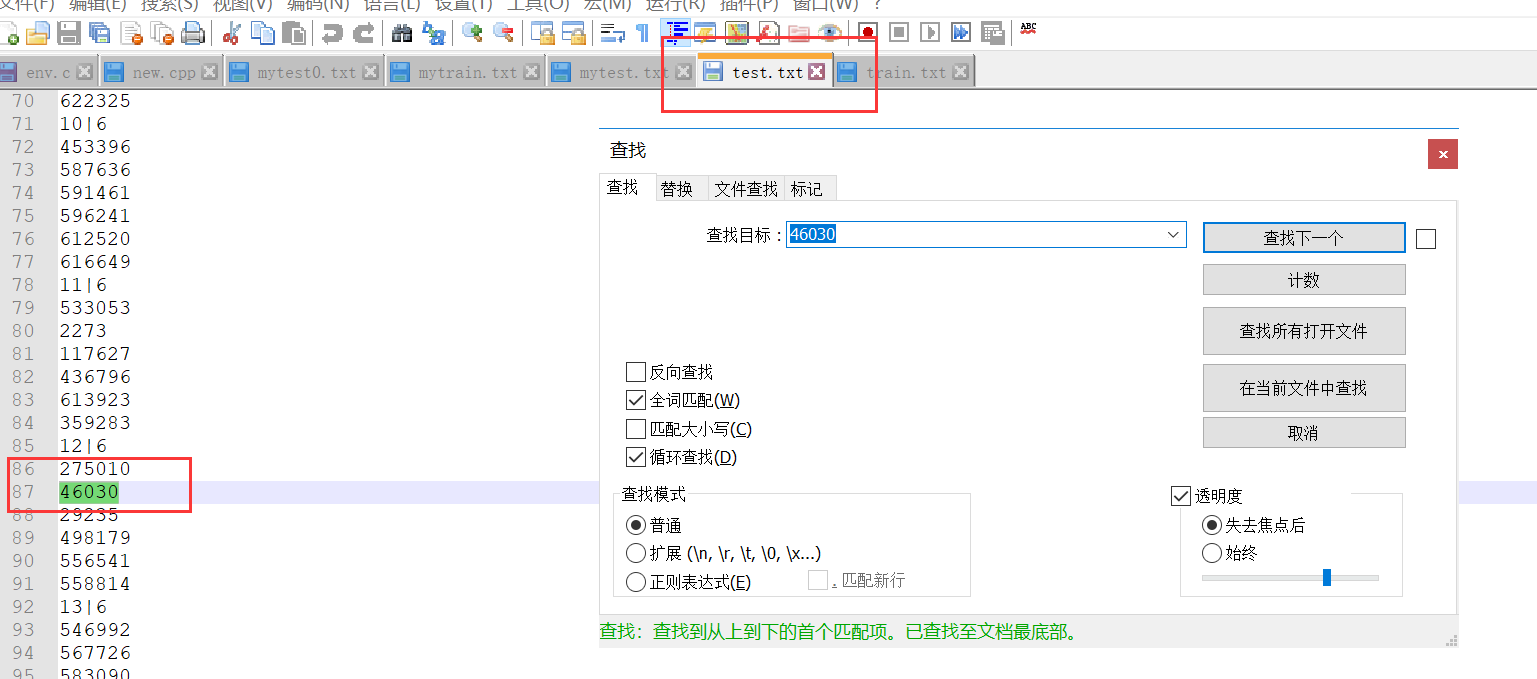
\*初始化P,Q矩阵



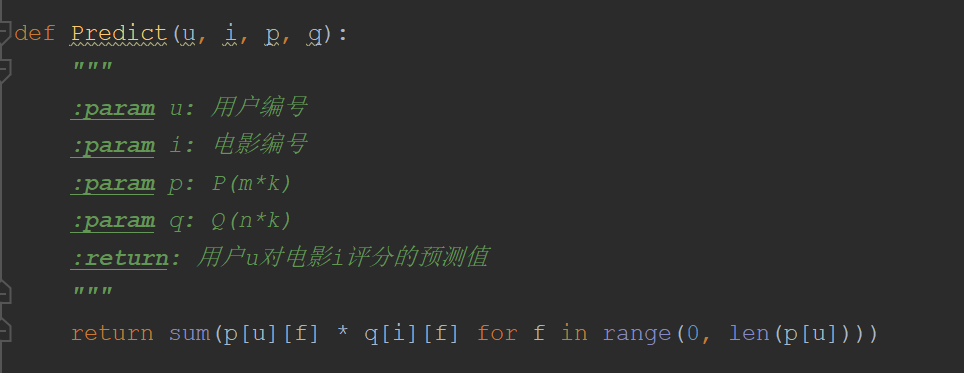
使用字典，用空间换时间，比list和矩阵索引的快；使用用户数量、电影编号的最大值分别初始化P,Q矩阵，尽管一些电影编号并没有在训练集中出现，但是测试集中会包含一些这类电影编号，在算取这类电影的评分时，使用的是初始的随机数（未经过更新）。



查了好久才意识到可能是数据集的问题，训练集中没有46030，而测试集中需要对其评分

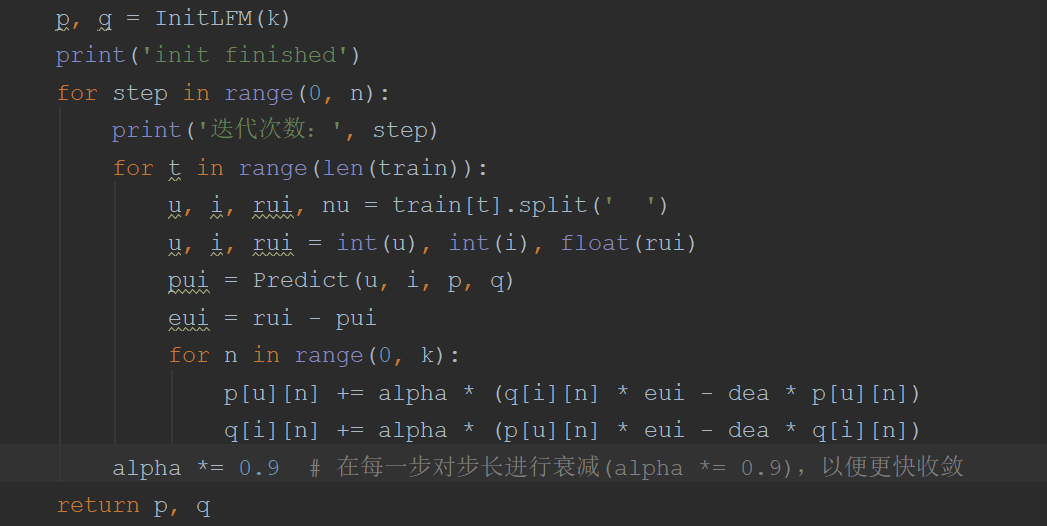
 

\*计算用户u对电影i的评分预测值



矩阵P,Q对应的行相乘就是预测值

\*迭代更新P,Q矩阵

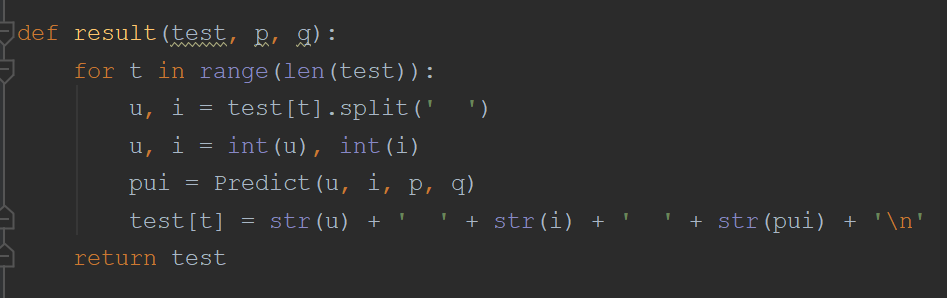


计算预测评分与实际评分的差值，使用公式更新P,Q，每次迭代对步长进行衰减，以便更快收敛。

**Puk = Puk + α( eui Qik - λPuk)**

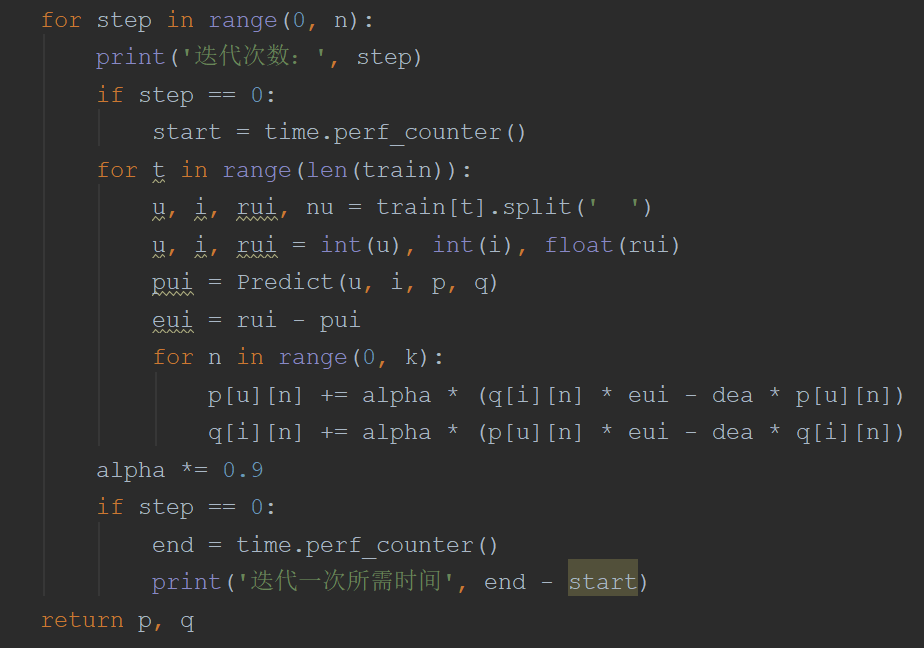
**Qik = Qik + α( eui Puk - λQik)**

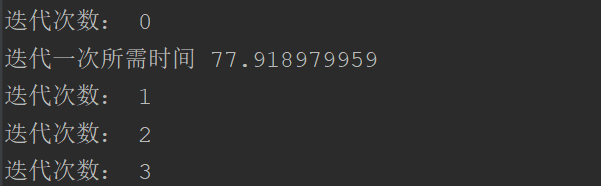
\*计算结果：



**结果**

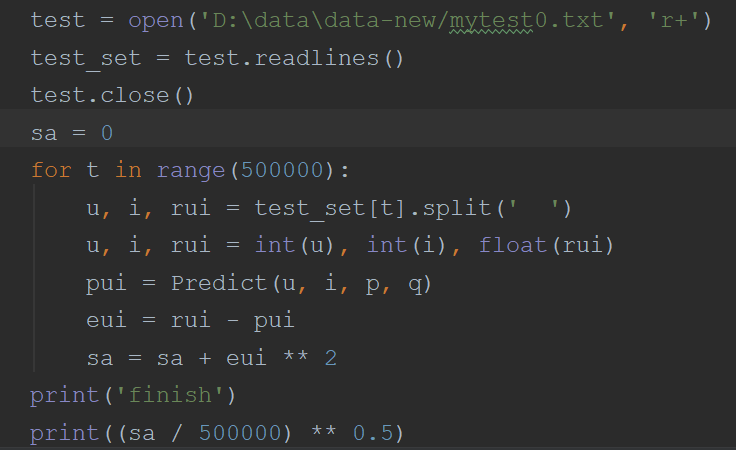
时间代价:

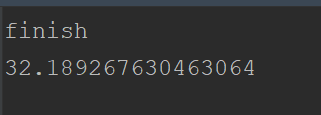




实际时间消耗稍长于一次迭代时间\*迭代次数

RMSE:





空间代价：O(k(m+n))

**实验分析**

算法：

\*使用LMF算法会不可避免地产生数据损失的问题，因为算法的基本思路是将一个大矩阵R(m×n)用两个低维矩阵P(m×k)、Q(k×n)来拟合，空间复杂度一下子从O(m×n)降到了O(m+n),时间复杂度也相应地降低了。牺牲一些准确度来换取时间和空间的收益是值得的，否则连R(m×n)矩阵都难以存储，更不必谈计算了。

\*算法思想简单，符合人类的直观思维，容易实现，这也是我选择改算法的主要原因。

\*难以解决冷启动问题，只能基于用户对电影的历史评分记录来预测用户对其他电影的评分，无法预测新电影或者新用户的评分。

实验分析：

维度k，学习率a的选取，

在一定范围内，k越大预测的准确率越高，但是时间复杂度和空间复杂度会成倍数的增长，最终选为15；学习率a越大，梯度下降的越快，但是有可能到达局部最小值，同时也有可能导致永远到达不了“谷底”，因此取a = 0.0001，并在每次迭代后也“更新”a，a=a\*0.9

存储P、Q矩阵的数据结构的选取，

一开始想到了array，但是仅是用随机数初始化P、Q矩阵就花费了较长的时间，后改用list也未有明显提升，最后使用的dict，查资料得知，当数据较大时，python中只有dict能保证O(1)的时间访问任意元素。简单尝试后，可以像使用矩阵一样使用dict存储矩阵P、Q.

最终RMSE在32左右，还是相对较大的，且得到的结果显示同一用户对六个电影的评分差距不是很大，最高分和最低分大致相差40，还有较大的提升空间，但是因为能力，时间，耐心有限，未做改进，希望有机会能更深入理解和改进推荐系统。