Project 2 个性化推荐

大数据分析（B）

1. 问题描述

给定用户行为矩阵X_{m*n},其中m是用户数，n是需要推荐的内容数量，X中的元素X_{ij}表示用户对某个电影的打分。因此，所谓的推荐任务就转化成，当我们已知X中的一部分值时，如何对未知值进行预测。

2. 数据集

使用的是 Netflix 推荐竞赛的一个子集，包含 10000 个用户和 10000 个电影。用户行为数据包含用户对电影的打分，分数的取值范围是 1–5. 我们选取行为数据的80%作为训练集，其余的 20%作为测试集。

3. 数据预处理

（1）处理用户列表 users.txt，将用户id在users.txt所在行数，作为每个用户在行为矩阵X中对应的行号。

```
user_dict = {} # user_id: matrix_index
with open('Project2-data/users.txt') as f:
    user_data = f.readlines()
    for i in range(len(user_data)):
        user_data[i] = user_data[i].strip()
        user_dict[user_data[i]] = I
```

（2）处理电影名称 movie_titles.txt，每个电影id，作为每个电影在行为矩阵X中对应的列号。

（3）处理训练集 netflix_train.txt，共有 6,897,746 条评分数据。根据用户 id，电影 id，分数，生成**训练矩阵 train**。对于分数未知的项，全定为 0 。

```
train = np.empty([10000,10000],dtype=np.int32)
with open('Project2-data/netflix_train.txt') as f:
    lines = f.readlines()
    for i,line in enumerate(lines):
        if i % 1000000 == 0:
```

```
            print i
        record = line.strip().split(' ') # user_id movie_id,score,date
        user_id,movie_id,score = record[0],record[1],int(record[2])
        user_index = user_dict[user_id]
        movie_index = int(movie_id)-1
        train[user_index][movie_index] = score
    print "import train data successfully"
```

（4）处理测试集 netflix_test.txt，共有 1,719,466 条评分数据。与根据用户 id，电影 id，分数，生成**测试矩阵 test**。代码处理逻辑同训练集 netflix_train.txt

选用全量数据进行实验。

4. 协同过滤

基于用户的协同过滤算法。当我们需要判断用户i是否喜欢电影 j，只要看与 i 相似的用户，看他们是否喜欢电影 j，并根据相似度对他们的打分进行加权平均。

$$score(i,j)=\frac{\sum_k sim(X(i),X(k))\cdot score(k,j)}{\sum_k sim(X(i),X(k))}$$

其中，X(i)表示用户 i 对所有电影的打分，就是X矩阵中第i行对应的 10000 维的向量（未知记为 0）。

$sim(X(i),X(k))$ 表示用户 i 和用户 k，对于电影打分的相似度，可以采用两个向量的 cos 相似度来表示，即：$cos(x,y)=\frac{x\cdot y}{|x|\cdot|y|}$。

通过上面的公式，我们就可以对测试集中的每一条记录，计算用户可能的打分。

采用 RMSE(Root Mean Square Error，均方根误差)作为评价指标，计算公式为：

$$RMSE=\sqrt{\frac{1}{n}(\sum_{<i,j>\in Test}(X_{ij}-\tilde{X}_{ij})^2)}$$

其中 Test 为所有测试样本组成的集合，X_{ij}为预测值，$\tilde{X}_{ij}$为实际值。

代码：

```
begin = datetime.datetime.now() # 开始计时
```

```
train_normalized = preprocessing.normalize(train, norm='l2') # 对train集做归一化
sim = train_normalized.dot(train_normalized.T) # 得到相似度矩阵sim(i,j)=train集用户i对train集用户j的cos相似度

numerator = sim.dot(train)  # 分子
denominator = sim.dot(np.ones((10000,10000)))  # 分母
score = numerator/denominator # 预测分数

flag_test = test
flag_test[flag_test > 0] = 1

RMSE = LA.norm(score*flag_test - test,"fro")/(test_len**0.5) # 均方根误差
 只考虑test集中有评价的那些打分
print "RMSE:",RMSE
print "运行时间：%d s"%((datetime.datetime.now() - begin).seconds) # 计时结束
```

运行结果：

RMSE: 0.709698188297
运行时间：42 s

代码逻辑：

为了求用户之间的相似度矩阵sim，将训练集用户行为矩阵train做归一化处理，生成归一化矩阵train_normalized，根据cos相似度的计算方法，train_normalized和train_normalized的转置的矢量乘即为相似度矩阵sim。根据计算score的方法，转化为矩阵乘的形式：

$$SCORE = \frac{SIM\ \ TRAIN}{SIM\ \ E}$$

其中SIM是相似度矩阵，TRAIN是训练集用户行为矩阵，E是全为1的矩阵，分子和分母做对应元素相除操作。得到预测的评分矩阵score，在做均方根误差时，只考虑在test集中有打分的项，所以用test的指示矩阵flag_test与score点乘后，再与测试集用户行为矩阵test相减，得到差值矩阵，对差值矩阵求F范数后除以test中非零元素个数的开方，得到RSME。

结果分析：

在计算RSME时，对预测打分矩阵score点乘了test的指示矩阵flag_test，即只考虑那些test即有打分的分数。当考虑test中包含0元素的值时，即直接用score与test相减，求得的RMSE更高一些。

	RMSE	运行时间
只考虑有打分的元素	0.709698188297	42 s
考虑所有元素	1.28879024694	43 s

5. 基于梯度下降的矩阵分解算法

对给定的行为矩阵X，我们将其分解为U，V两个矩阵的乘积，使UV的乘积在已知值部分逼近X，即：$X_{m*n} \approx U_{m*k}V^T_{N*k}$，其中k为隐空间的维度，是算法的参数。基于行为矩阵的低秩假设，我们可以认为U和V是用户和电影在隐空间的特征表达，它们的乘积矩阵可以用来预测X的未知部分。

使用梯度下降法优化求解这个问题。推荐算法的目标函数是：

$$J = \frac{1}{2}\|A \circ (X - UV^T)\|_F^2 + \lambda\|U\|_F^2 + \lambda\|V\|_F^2$$

其中，A是指示矩阵，A_{ij}=1意味着X_{ij}的值为已知，反之亦然。○是阿达马积（即矩阵逐元素相乘）。$\|\cdot\|_F$表示矩阵的Frobenius范数。计算公式为$\|A\|_F=\sqrt{\Sigma\Sigma a_{ij}{}^2}$。在目标函数J中，第一项为已知值部分，UV的乘积逼近X的误差。后面的两项是为防止过拟合加入的正则项，λ为控制正则项大小的参数，由我们自己定义。

当目标函数取得最小值时，算法得到最优解。首先，我们对U和V分别求偏导，结果如下：

$$\frac{\partial J}{\partial U} = (A \circ (UV^T - X)V) + 2\lambda U$$

$$\frac{\partial J}{\partial V} = (A \circ (UV^T - X)U) + 2\lambda V$$

之后，我们迭代对U和V进行梯度下降，具体算法如下：

Initialize U and V (very small random value);

Loop until converge:

$$U = U - \alpha\frac{\partial J}{\partial U}$$

$$V = V - \alpha \frac{\partial J}{\partial V}$$

End loop

算法中α为学习率，选择0.001、0.01、0.1三个实数值。算法的收敛条件，选择目标函数J的变化量小于阈值0.00005。

在给定的k和λ下，对目标函数J进行优化求解。

a）对于给定**k=50, λ=0.01**的情况，画出迭代过程中目标函数值J和测试集上RMSE的变化，给出最终的RMSE，并对结果进行简单分析。

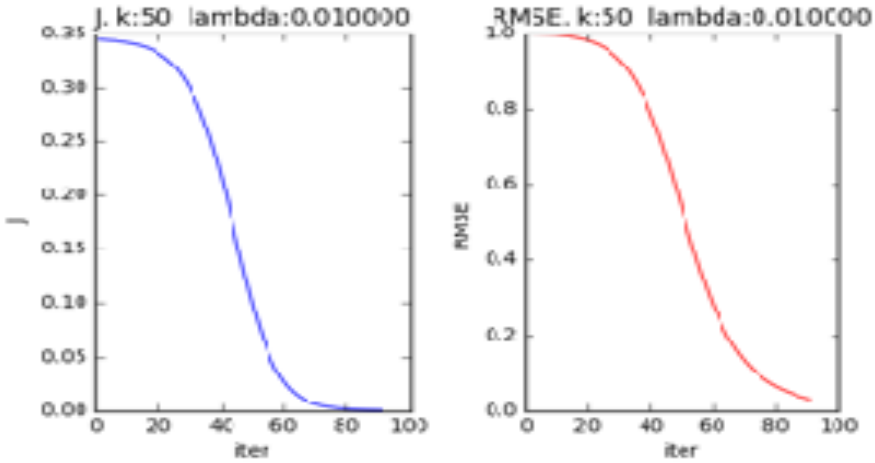


图1 J和RMSE随着迭代次数的变化

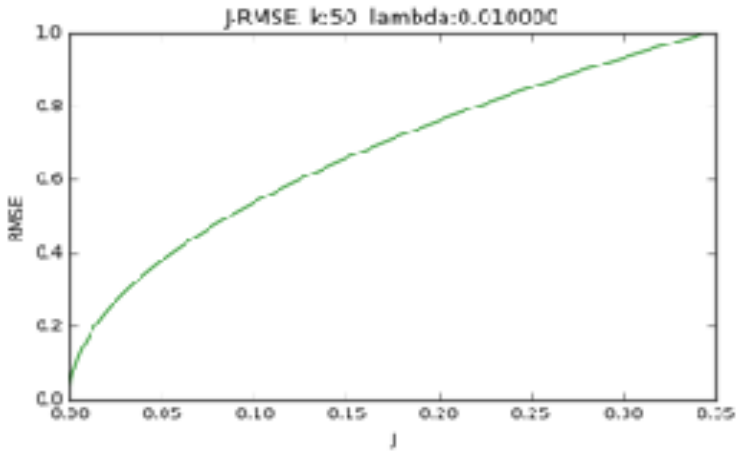


图2 J和RMSE之间的关系图

k	λ	α	RMSE	运行时间	迭代次数
50	0.01	0.00005	0.031467	540 s	91

分析：

J和RMSE随迭代次数的变化都是S型的，即下降速度由慢变快，又逐渐减慢，最后收敛到一个值。

J和RMSE的关系如图2。呈现正相关的特性。

与协同过滤算法相比，矩阵分解算法的计算精度更高，但是消耗时间更多。

	RSME	运行时间
协同过滤	0.709698	42 s
矩阵分解	0.031467	540 s

b）调整k的值（如20，50）和λ的值（如0.001，0.1），比较最终RMSE的效果，选取最优的参数组合。

k	λ	α	RMSE	运行时间	迭代次数
50	0.1	0.00005	0.031563	536 s	91
50	0.01	0.00005	0.031467	540 s	91
50	**0.001**	**0.00005**	**0.031276**	**541 s**	**91**
20	0.1	0.00005	0.031795	546 s	98
20	0.01	0.00005	0.031686	546 s	98
20	0.001	0.00005	0.031725	544 s	98

分析：

上表中标红的参数组合是表现最佳的参数组合（k=50，λ=0.001）。

从时间上看，不同参数k和不同参数λ对迭代次数没有很大的改变，所以运行时间也没有很大的变化。

从RMSE上看，给定k值，观察不同λ对RMSE的影响，在k=50的情况下，正则项系数λ越小，RMSE会稍微小一些。但是在k=20的情况下，正则项系数λ与RMSE不呈现单调性的特性。而且，λ的变化对RMSE的影响不大，所以，不能断言k的大小与RMSE有某种关系。给定λ值，观察不同k对RMSE的影响，在λ=0.1，0.01，0.001三种取值的情况下，k=50时的RMSE都比k=20时小一些。

另外，在对步长alpha进行试探时，发现步长较小时，收敛的效果更好。对U、V的取值，开始只是取了（0,1）之间的随机数，发现收敛效果很差，后来将U、V在（0，0.01）内取值。

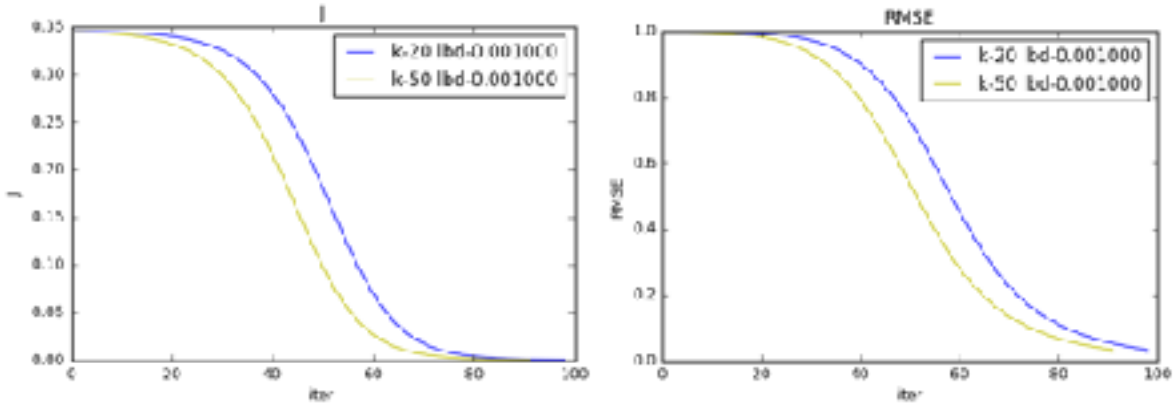
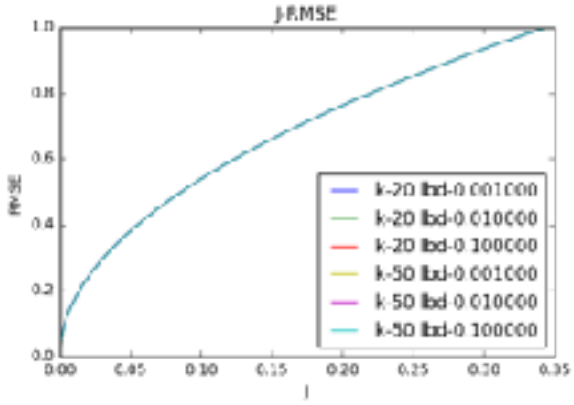


图3 不同参数下的J和RMSE随迭代次数的变化图

在不同参数下观察J和RMSE随迭代次数的变化曲线，根据试验，相同的k值下，J和RMSE随迭代次数的变化曲线没有明显差异，所以选取λ=0.01时，k分别取20和50时的曲线图。从图中观察可以看到，当k=50时，J和RMSE下降的速率比k=20时要快。



尝试对不同参数下的J–RMSE的关系图进行比对，发现六组参数下的J–RMSE关系曲线都没有明显的差异。

代码：

```
# 生成参数集
k_list = [20,50]
lambda_list = [0.001,0.01,0.1]
parameter_list = []
for k in k_list:
    for lbd in lambda_list:
        parameter_list.append((k,lbd))
```

```
iters = []
Js = []
RMSEs = []
Losses = []
with open('log.txt','w') as f:
    for idx,(k,lambda_) in enumerate(parameter_list):
        f.write("==========================\n")
        f.write("parameter %d: k: %d\tlambda: %f\n"%(idx+1,k,lambda_))
        f.write("==========================\n")
        # 初始化
        alpha = 0.0001
        L = 0.00005
        U = 0.01*np.random.rand(10000,k)
        V = 0.01*np.random.rand(10000,k)
        X = train
        A = train
        A[A > 0] = 1
        B = test
        B[B > 0] = 1

        J = 0.5*(LA.norm(A*(X-U.dot(V.T)),"fro")**2) + lambda_*(LA.norm(U,"fro")**2) +
lambda_*(LA.norm(V,"fro")**2)
        J = J/1e7
        Jo = J+1
        iter_list = []
        J_list = []
        RMSE_list = []
        Loss_list = []
        iter_ = 1
        begin = datetime.datetime.now()
        # 迭代进行梯度下降
        while Jo - J > L and iter_ < 100:
            Jo = J
            J_U = (A*(U.dot(V.T)-X)).dot(V) + 2*lambda_*U
            J_V = (A*(U.dot(V.T)-X)).dot(U) + 2*lambda_*V

            U = U - alpha * J_U
            V = V - alpha * J_V

            J = 0.5*(LA.norm(A*(X-U.dot(V.T)),"fro")**2) + lambda_*(LA.norm(U,"fro")**2) +
lambda_*(LA.norm(V,"fro")**2)
            J = J/1e7
            RMSE = LA.norm(B*U.dot(V.T) - test,"fro")/(test_len**0.5)
            loss = Jo - J

            f.write("iter: %d \t J: %.6f \t RMSE: %.6f \t Loss: %.6f\n"%(iter_,J,RMSE,loss))

            iter_list.append(iter_)
            J_list.append(J)
            RMSE_list.append(RMSE)
            Loss_list.append(loss)

            iter_ += 1

        iters.append(iter_list)
        Js.append(J_list)
        RMSEs.append(RMSE_list)
```

```
Losses.append(Loss_list)

f.write("========= end =========\n")
f.write("iter: %d \t J: %.6f \t RMSE: %.6f \t Loss: %.6f\n"%(iter_,J,RMSE,loss))
f.write("运行时间：%d s\n"%((datetime.datetime.now() - begin).seconds))
```

1. 首先生成需要运行的（k，λ）参数集
2. 初始化。包括下降步长alpha，收敛阈值L，分解矩阵U、V，指示矩阵A、B（分别对应train和test），初始化J。
3. 迭代进行梯度下降，迭代条件是上一轮迭代的J和新J的差值大于停止迭代的阈值L或者迭代轮次小于100。更新Jo（上一轮迭代的J），根据偏导J_U和J_V计算新一轮的U和V，根据U和V计算新一轮的J。根据这一轮的U和V值，求出RMSE。保存迭代轮次iter、这一轮的J、RMSE、Loss，为画图提供条件。
4. 将所有信息打印到log.txt，以供分析。

6. 协同过滤和矩阵分解的对比，讨论两者优缺点。

从时间上看，协同过滤的耗时较少，矩阵分解的耗时较大；从RMSE上看，协同过滤RMSE 较大，矩阵分解RMSE较小。可见，协同过滤效率更高，而矩阵分解更准确。

（1） 协同过滤

- 优点：

  效率高，速度快；

  随着数据量增加，准确性越来越高；

  便于基于社交媒体的好友推荐；

- 缺点：

  稀疏性问题；

  系统延伸性问题；

  新用户问题，系统开始时推荐质量较差；

（2） 矩阵分解

- 优点：

  容易编程实现，实现复杂度低；

预测效果也好，同时还能保持扩展性；

- 缺点：

  解释性没有协同过滤的推荐算法好；

  效率低，因子对意义不明确。