Contents

1 Introduction

矩阵分解算法是推荐系统最常用的算法之一[5]，其具有诸多变体和改进形式[1,2,3,4]。[6]在实验部分对多种矩阵分解算法在多个数据集上进行了多项指标的对比。

本次结课报告主要实现[6]的实验部分。由于[6]中的算法过于复杂，我们选择几种比较简单的算法[3]在多个数据集上进行多项指标的对比。

2 Datasets

在数据集方面，我们仿照[6]，选取了评分推荐中最常见的三个数据集：Epinions，Movielens(100k)，Netflix(1M)。Epinions 数据集来自于在线商品销售，后两者来自于电影评价网站。评分的值都是从 1 到 5。

Epinions[7]：此数据集是从 Epinions.com 网站进行的为期 5 周的抓取（2003 年 11 月/12 月）中收集的，在 Epinions 中，用户可以撰写评论并为各种产品评分。

Movielens（100k）[8]：此数据集由 GroupLens Research 收集，并从 MovieLens 网站提供了评级数据集，该网站是一个基于 Web 的电影评价社区。

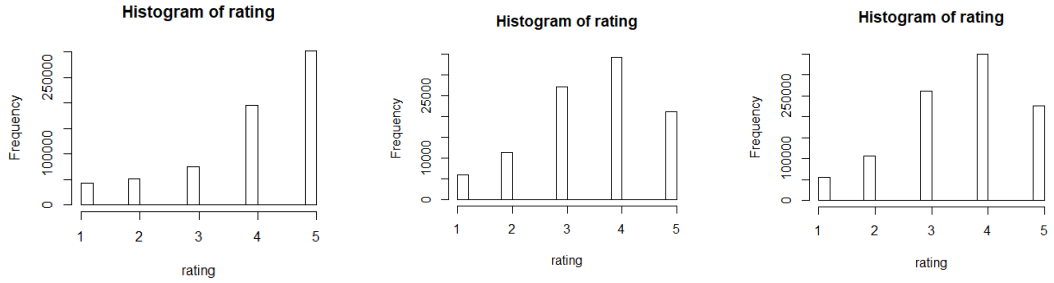
Netflix(1M)[9]：此数据集由 Netflix 举办的 Netflix 奖竞赛提供，Netflix 进行在线 DVD 租赁和视频流服务。

接下来，我们采用 R 语言展示更多数据集的信息。

```
rm(list=ls())
setwd("C:/Users/Lenovo/Code/R/r_pmf_paper")
data=read.table("epinion.txt", header = FALSE)
rating=data[,3]
movie=data[,2]
user=data[,1]

hist(rating)
data_num=length(rating)
movie_num= max(movie)#电影数量
user_num=max(user)#用户数量
sparse=data_num/(movie_num/10000*user_num)/10000#计算稀疏度
N=data_num/user_num
M=data_num/movie_num
```

我们先展示每个数据集各自评分的分布：



Epinions　　Movielens(100k)　　Netflix(1M)

Fig. 1. 数据集评分分布

之后计算出每个数据集的以下指标，可以让我们对数据集有一个多角度的认识

Table 1. 数据集多维度信息

Index	Epinions	Movielens(100k)	Netflix(1M)
#users	49289	943	6040
#items	139738	1682	3952
#ratings	664824	100000	1000209
density	9.65e-05	0.063	0.042
N	13	106	165
M	5	59	253

N：平均每个用户评价的物品数量

M：平均每个物品拥有的评分数量

3 Metrics

为了验证不同算法的预测效果，参考论文[6]，我们先选取了两种最常用的评价指标，即均方根误差(RMSE)与绝对平均误差(MAE)。其定义是：

$$RMSE = \sqrt{\frac{1}{|\Omega_t|}\sum_{(i,j)\in\Omega_t}(\mathbf{R}_{ij}-\hat{\mathbf{R}}_{ij})^2} \qquad MAE = \frac{1}{|\Omega_t|}\sum_{(i,j)\in\Omega_t}|\mathbf{R}_{ij}-\hat{\mathbf{R}}_{ij}|$$

其中Ω_t是测试集包含的所有元素，R_{ij}是用户i对物品j的真实评分。以上两种指标因为都代表了误差，所以都是越小越好。

由于推荐系统最终的目的是为用户推荐几项物品，所以所预测出的物品评分排名尤其重要。因此，推荐系统中还有一种评价方法，用来评价物品排名的准确度。我们定义用户 i 的前 K 个物品的预测精确率 Pre@K(i)与召回率 Re@K(i)。

$$Re@K(i) = \frac{|R(i)\cap T(i)|}{|T(i)|} \qquad Pre@K(i) = \frac{|R(i)\cap T(i)|}{K}$$

$R(i) = \{j \in \Omega(i) | R \geq 4\}$表示用户 i 评分超过 4 分的物品集合。$\Omega(i)$表示在测试集上用户 i 真实评分过的物品。$T(i) = \{j \in \Omega(i) | \hat{R}_{ij} \geq 4\}$表示我们预测出的用户最喜欢的物品（评分超过 4 分）的集合。精确率和召回率越大，排名效果越好。

4 Algorithm

本文的算法使用[3]提出的概率矩阵分解算法(PMF)及其变体，即以下三种算法：

PMF-SGD：使用随机梯度下降算法优化的概率矩阵分解算法

MF-momentum：使用动量算法优化[10]的矩阵分解算法

PMF-momentum：使用动量算法优化[10]的概率矩阵分解算法

4.1 Probabilistic Matrix Factorization

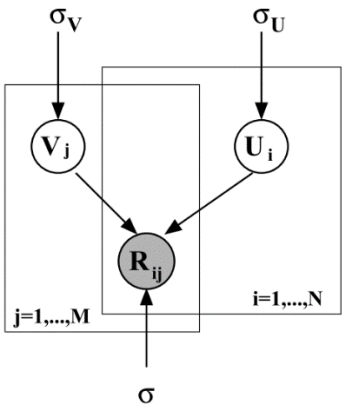


Fig. 2. PMF 的概率图模型

概率矩阵分解（PMF）是对于 FunkSVD[5]的概率解释版本。它假设评分矩阵中的元素R_{ij}是由用户潜在偏好向量 U_i和物品潜在属性向量V_j的内积决定的，并且服从均值为 $U_i^T V_j$，方差为σ^2的正态分布：

$$R_{ij} \sim N(U_i^T V_j, \sigma^2)$$

则观测到的评分矩阵条件概率为：

$$p(R|U,V,\sigma^2) = \prod_{i=1}^{N}\prod_{j=1}^{M}\left[\mathcal{N}(R_{ij}|U_i^T V_j, \sigma^2)\right]^{I_{ij}}$$

同时，假设用户偏好向量U_i与物品偏好向量V_j服从于均值都为 0，方差分别为 σ_u^2, σ_v^2的正态分布：

$$p(U|\sigma_U^2) = \prod_{i=1}^{N}\mathcal{N}(U_i|0,\sigma_U^2\mathbf{I}), \quad p(V|\sigma_V^2) = \prod_{j=1}^{M}\mathcal{N}(V_j|0,\sigma_V^2\mathbf{I}).$$

根据贝叶斯公式，可以得出潜变量 U,V 的 log 后验概率为：

$$p(U,V|R) = p(U,V,R)/p(R) \propto p(U,V,R) = p(R|U,V)p(U)p(V)$$

$$\ln p(U,V|R,\sigma^2,\sigma_V^2,\sigma_U^2) = -\frac{1}{2\sigma^2}\sum_{i=1}^{N}\sum_{j=1}^{M} I_{ij}(R_{ij}-U_i^T V_j)^2 - \frac{1}{2\sigma_U^2}\sum_{i=1}^{N} U_i^T U_i - \frac{1}{2\sigma_V^2}\sum_{j=1}^{M} V_j^T V_j$$
$$-\frac{1}{2}\left(\left(\sum_{i=1}^{N}\sum_{j=1}^{M} I_{ij}\right)\ln\sigma^2 + ND\ln\sigma_U^2 + MD\ln\sigma_V^2\right) + C, \qquad (3)$$

上式是一个极大化后验概率（MAP）的问题，我们令$\lambda_u = \frac{\sigma^2}{\sigma_u^2}, \lambda_v = \frac{\sigma^2}{\sigma_v^2}$ ，我们得到了最终需要最小化的损失函数:

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{M} I_{ij}\left(R_{ij}-U_i^T V_j\right)^2 + \frac{\lambda_U}{2}\sum_{i=1}^{N} \| U_i \|_{Fro}^2 + \frac{\lambda_V}{2}\sum_{j=1}^{M} \| V_j \|_{Fro}^2,$$

与原始的矩阵分解（MF）方法相比，相当于增加了关于参数的正则项。

根据损失函数，我们采用随机梯度下降（SGD）算法进行迭代求解:

$$\frac{\partial E}{\partial U_i} = -\left(R_{ij}-U_i^T V_j\right)V_j + \lambda_U U_i$$

$$\frac{\partial E}{\partial V_j} = -\left(R_{ij}-U_i^T V_j\right)U_i + \lambda_V V_j$$

$$U_i \leftarrow U_i - \eta\frac{\partial E}{\partial U_i} = U_i + \eta\left(\left(R_{ij}-U_i^T V_j\right)V_j - \lambda_U U_i\right)$$

$$V_i \leftarrow V_j - \eta\frac{\partial E}{\partial V_j} = V_j + \eta\left(\left(R_{ij}-U_i^T V_j\right)U_i - \lambda_V V_j\right)$$

4.2 Optimization

在这里，参考论文[10]，我们介绍一下本实验使用的两种优化方法，我们设 J(Θ)为损失函数。

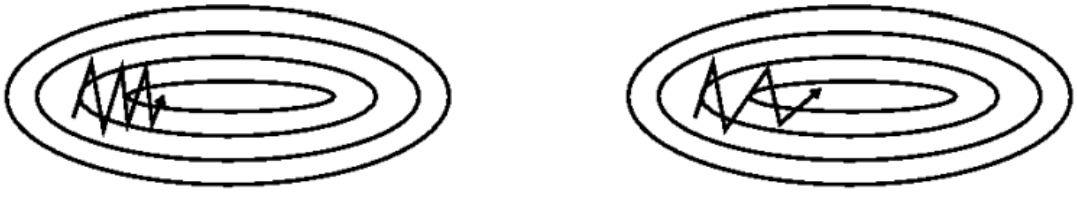
- 随机梯度下降算法

$$\theta = \theta - \eta\cdot\nabla_\theta J(\theta)$$

- 动量优化算法

$$v_t = \gamma v_{t-1} + \eta\nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

其中，动量优化可以加速优化、拥有更快的收敛速度和更低的方差[10]。



(a) SGD without momentum　　(b) SGD with momentum

Fig. 3. 两种优化方法的对比

5 Core Code

以下是 PMF-momentum 算法的核心代码, 优化改编自 Ruslan Salakhutdinov 提供的 matlab 代码[11]。输出为均方根误差(RMSE)与绝对平均误差(MAE), 以及准确率 Pre 与召回率 Re。作者也在 Github 上提供了全部的代码[12]。

```
rm(list=ls())
setwd("C:/Users/Lenovo/Code/R/r_pmf_paper")
data=read.table("movielen.txt", header = FALSE)
rating=data[,3]
movie=data[,2]
user=data[,1]
data_num=length(rating)
movie_num= max(movie)#电影数量
user_num=max(user)#用户数量

#parametre
epsilon=50 # Learning rate  学习率
lambda   = 0.1#Regularization parameter 正则化参数
momentum=0.7#动量优化参数
epoch=1#初始化 epoch
maxepoch=50#总训练次数
num_feat = 5 #Rank 10 decomposition  隐因子数量
err_train1=rep(0,maxepoch)
err_valid1=rep(0,maxepoch)
err_random=rep(0,maxepoch)

train_num=(data_num*9)%/%10;
train_vec=data[1:train_num,];#训练集
probe_vec=data[train_num:data_num,];#测试集

movie_num= max(movie)#电影数量 1682
user_num=max(user)#用户数量 943
#sum_list=max(data)
mean_rating = mean(train_vec[,3])#平均评级

pairs_tr = length((train_vec[,3]))#training data  训练集长度
pairs_pr = length((probe_vec[,3]))#validation data  验证集长度

numbatches= 9 #Number of batches 把数据分为 9 份
num_m = movie_num   # Number of movies  电影数量
num_p = user_num   #Number of users  用户数量

#初始化
```

```
w1_M1     = 0.1*matrix(runif(num_m*num_feat),nrow = num_m , ncol = num_feat ,byrow =T) # Movie feature vectors 生成用户物品特征矩阵 d=10
w1_P1     = 0.1*matrix(runif(num_p*num_feat),nrow = num_p , ncol = num_feat ,byrow =T) # User feature vecators
w1_M1_inc = matrix(rep(0,num_m*num_feat),nrow = num_m , ncol = num_feat ,byrow =T)#生成同 shape 的全零矩阵
w1_P1_inc = matrix(rep(0,num_p*num_feat),nrow = num_p , ncol = num_feat ,byrow =T)


for(epoch in epoch:maxepoch){
  #采用 mini batch 的方法，每次训练 9 个样本
  for(batch in 1:numbatches){
    N=10000 #number training triplets per batch 每次训练三元组的数量
    aa_p= train_vec[((batch-1)*N+1):(batch*N),1]#读取用户列每次读取一万个
    aa_m= train_vec[((batch-1)*N+1):(batch*N),2]#读取电影列
    rating = train_vec[((batch-1)*N+1):(batch*N),3]#读取评级列
    rating = rating-mean_rating; #Default prediction is the mean rating.
    # Compute Predictions %%%%%%%%%%%%%%%%%%
    pred_out= apply(w1_M1[aa_m,]*w1_P1[aa_p,],1,sum)#每一行进行求和,.*为对应元素相乘 10k*1，用隐特征矩阵相乘得到 1w 个预测评级
    f=sum((pred_out - rating)^2 + 0.5*lambda*(apply((w1_M1[aa_m,]^2 + w1_P1[aa_p,]^2),1,sum) ))
    #求出损失函数
    #Compute Gradients %%%%%%%%%%%%%%%%%%%%迭代
    IO =2*(pred_out - rating)
    kkkk=num_feat-1
    for(kkk in 1:kkkk){
      IO =cbind(IO,2*(pred_out - rating))
    }

    #IO = repmat(2*(pred_out - rating),1,num_feat)
    #将损失矩阵的二倍，复制 10 列
    lx_m=IO*w1_P1[aa_p,] + lambda*w1_M1[aa_m,]#损失*U-lambda*V 就是更新规则
    lx_p=IO*w1_M1[aa_m,] + lambda*w1_P1[aa_p,]#损失*V-lambda*U
    dw1_M1 = matrix(rep(0,num_m*num_feat),nrow = num_m , ncol = num_feat ,byrow =T)
    dw1_P1 = matrix(rep(0,num_p*num_feat),nrow = num_p , ncol = num_feat ,byrow =T)#生成全零用户特征矩阵

    for(ii in 1:N){#迭代一万次 每一行一行来 得到更新矩阵
      dw1_M1[aa_m[ii],]=   dw1_M1[aa_m[ii],] +   lx_m[ii,]
      dw1_P1[aa_p[ii],]=   dw1_P1[aa_p[ii],] +   lx_p[ii,]
    }

    # Update movie and user features %%%%%%%%%%
```

```
    w1_M1_inc = momentum*w1_M1_inc + epsilon*dw1_M1/N
    w1_M1 =    w1_M1 - w1_M1_inc#原矩阵-负导数*学习率
    w1_P1_inc = momentum*w1_P1_inc + epsilon*dw1_P1/N
    w1_P1 =    w1_P1 - w1_P1_inc
  }
  #此时所有(9 轮)batch 结束
  #现在已经得到了此轮 epoch 后的一组 U 和 V
  # Compute Predictions after Paramete Updates %%%%%%%%%%%%%%%%%%
  pred_out= apply(w1_M1[aa_m,]*w1_P1[aa_p,],1,sum)
  f_s=sum((pred_out - rating)^2 + 0.5*lambda*(apply((w1_M1[aa_m,]^2 +
w1_P1[aa_p,]^2),1,sum) ))
  err_train1[epoch] = sqrt(f_s/N)
  #Compute predictions on the validation set %%%%%%%%%%%%%%%%%%%%%%
  NN=pairs_pr#验证集长度
  aa_p = probe_vec[,1]#读取验证集的 user、movie、rating
  aa_m = probe_vec[,2]
  rating = probe_vec[,3]

  pred_out =apply(w1_M1[aa_m,]*w1_P1[aa_p,],1,sum) + mean_rating#预测结果加上 mean
才行
  pred_out[pred_out>5]=5
  pred_out[pred_out<1]=1 #使得预测结果超过评分区间的值依旧掉在区间内

  err_valid1[epoch]= sqrt(sum((pred_out- rating)^2)/NN)
  RMSE=sqrt(sum((pred_out- rating)^2)/NN)
  MAE=sum(abs(pred_out- rating))/NN
  print(paste('epoch',epoch,'Train RMSE',signif(err_train1[epoch], 4),'Test
RMSE',signif(err_valid1[epoch], 4)))
}
print(paste('RMSE',signif(RMSE, 4),'MAE',signif(MAE, 4)))

#计算 Pre@5 Re@5
# TOP-N 推荐

j=5
txt=numeric(user_num)
for(i in 1:user_num){
  user_i_rating_real=probe_vec[(probe_vec$V1==i),]
  user_i_rating_real=user_i_rating_real[order(user_i_rating_real$V3,decreasing = TRUE),]
  user_i_rating=w1_P1[i,]%*%t(w1_M1[user_i_rating_real$V2,])+mean_rating
  if(length(user_i_rating_real$V2)>j){
    txt[i]=sum(rank(-1*user_i_rating)[1:j]<=j)/j
  }
  if(length(user_i_rating_real$V2)<=j){
```

```
      ti=sum(user_i_rating_real$V3>=4)
      if(ti!=0){
        txt[i]=sum(user_i_rating[1:ti]>=4)/ti
      }
    }
}
Pre=mean(txt)

txt=numeric(user_num)
for(i in 1:user_num){
  user_i_rating_real=probe_vec[(probe_vec$V1==i),]
  user_i_rating_real=user_i_rating_real[order(user_i_rating_real$V3,decreasing = TRUE),]
  user_i_rating=w1_P1[i,]%*%t(w1_M1[user_i_rating_real$V2,])+mean_rating
  if(length(user_i_rating_real$V2)>j){
    user_i_rating[user_i_rating<4]=0
    user_i_rating[user_i_rating>4]=1
    ti=sum(user_i_rating)
    if(ti!=0){
      bigerthan4=sum(user_i_rating_real$V3>=4)
      tinri=sum(user_i_rating[1:bigerthan4])
      txt[i]=tinri/ti
    }
    }
  if(length(user_i_rating_real$V2)<=j){
    ti=sum(user_i_rating_real$V3>=4)
    if(ti!=0){
      txt[i]=sum(user_i_rating[1:ti]>=4)/ti
    }
  }
}
Re=mean(txt)
print(paste('Pre',Pre,'Re',Re))
```

5.2 Parameter Adjustment

在机器学习问题下，调参尤为重要，合理的超参数会使得模型的优点得到充分的展现。接下来，我们将 PMF 算法封装为一个函数方便调用，进行调参过程。以下代码展示了第一个超参数的调试过程。

```
rm(list=ls())
path="C:/Users/Lenovo/Code/r_pmf" #声明 test2.R 所在位置
setwd(path)  #把工作路径设置到 path
source('fun_pmf.R')#“预装“函数
err=fun_pmf(50,0.1,0.7,50,3)
err5=fun_pmf(50,0.1,0.7,50,5)
```

```
err7=fun_pmf(50,0.1,0.7,50,7)
err9=fun_pmf(50,0.1,0.7,50,9)
err11=fun_pmf(50,0.1,0.7,50,11)
#在训练集上的表现对比
plot(err$err1,type="l",col=4,lwd=2,main="Train Loss Comparison",xlab="epoch",ylab="Loss",ylim=c(0.8,1.1))
lines(err5$err1,type="l",col=3,lwd=2)
lines(err7$err1,type="l",col=2,lwd=2)
lines(err9$err1,type="l",col=1,lwd=2)
lines(err11$err1,type="l",col=5,lwd=2)

legend("topright",c("k=3","k=5","k=7","k=9","k=11"),lty=1,col=c(4,3,2,1,5))
#在测试集上的表现对比
plot(err$err2,type="l",col=4,lwd=2,main="Test Loss Comparison",xlab="epoch",ylab="Loss")
lines(err5$err2,type="l",col=3,lwd=2)
lines(err7$err2,type="l",col=2,lwd=2)
lines(err9$err2,type="l",col=1,lwd=2)
lines(err11$err2,type="l",col=5,lwd=2)

legend("topright",c("k=3","k=5","k=7","k=9","k=11"),lty=1,col=c(4,3,2,1,5))
```

5.2.1 Latent Factor Dimension-k

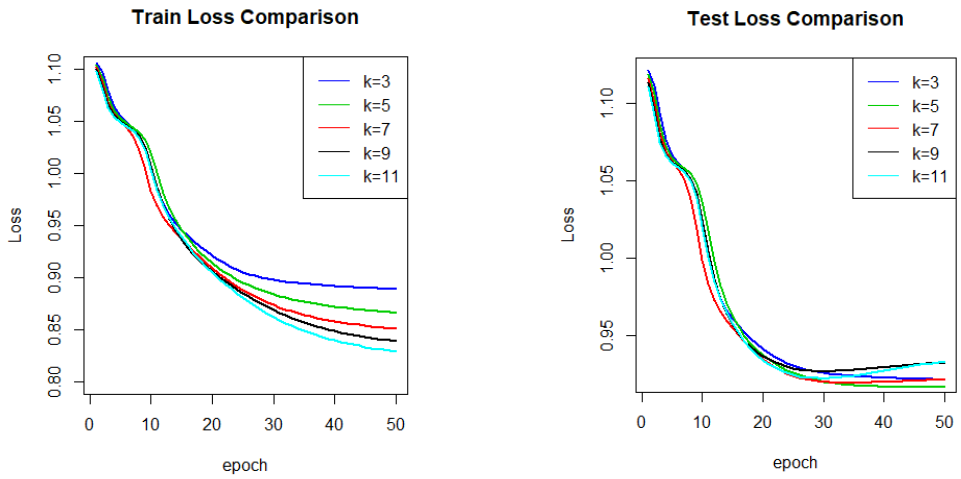


Fig. 4. 不同超参数在训练集和测试集的表现

最优值为 k=5

5.2.2 Learning Rate

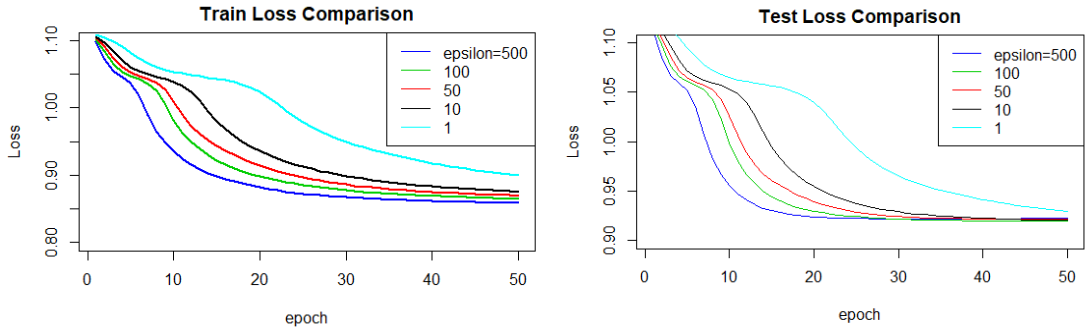


Fig. 5. 不同超参数在训练集和测试集的表现

最优值为 eplison=100

5.2.3 Regularization Parameter

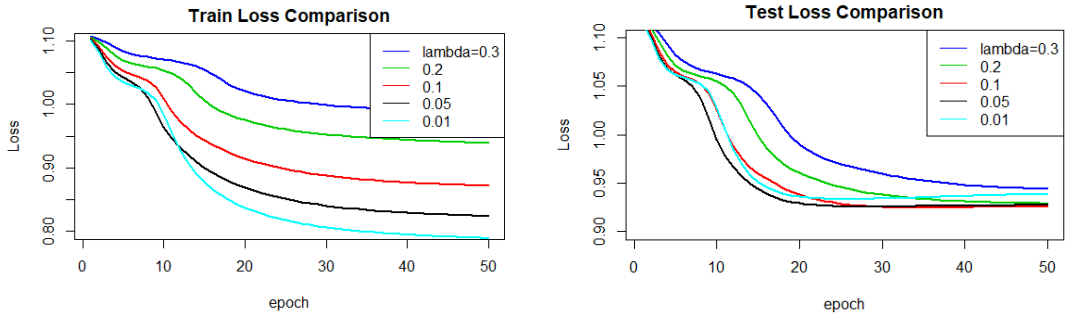


Fig. 6 不同超参数在训练集和测试集的表现

最优值为 lambda=0.1

5.2.4 Momentum Optimization Parameter

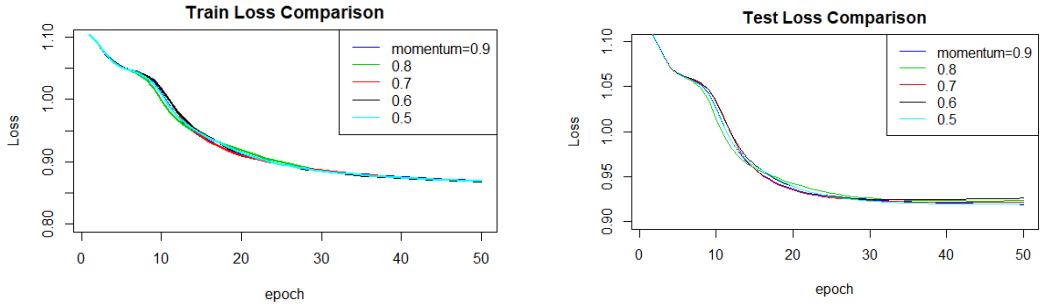


Fig. 7. 不同超参数在训练集和测试集的表现

最优值为 momentum=0.9

5.3 Comparison

在这一部分，我们先大致对比一下三种算法在测试集上的表现：

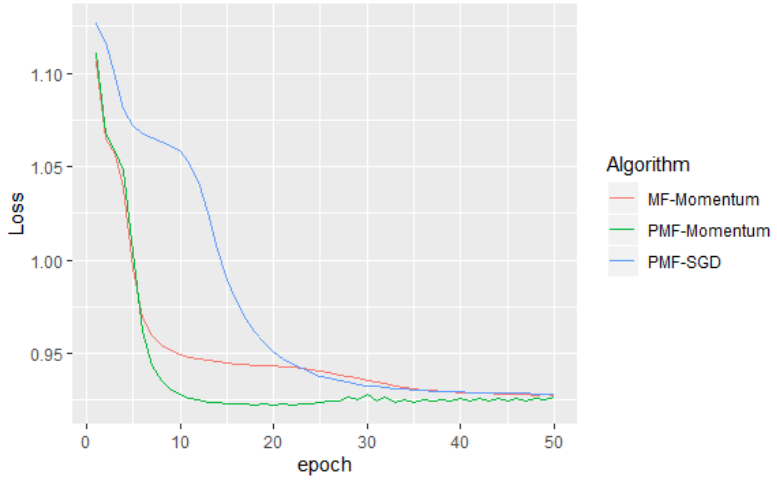


Fig. 8. 三种算法在测试集上的表现

6 Experiments

通过调参，我们确定了每种算法在每个数据集上的最佳参数，输出了其四种度量指标。下面是三种算法在三个数据集上四种指标的对比结果。

6.1 MAE/RMSE

Table 2.算法效果对比

Datasets	Epinions		Movielens		Netflix	
Metrics	MAE	RMSE	MAE	RMSE	MAE	RMSE
MF-m	**0.9293**	**1.211**	0.7334	0.9484	0.8662	1.127
PMF-m	0.9309	1.213	0.7286	0.9283	0.8202	1.053
PMF-SGD	0.9331	1.217	**0.7262**	**0.926**	**0.776**	**0.9845**

从预测评分的损失角度，综合来看，PMF-SGD 算法的效果最好。

6.2 Re@K(i)/Pre@K(i)

Table 3. 算法效果对比

Datasets	Epinions			
Metrics	Re@5	Re@10	Pre@5	Pre@10
MF-m	**0.1177**	**0.1115**	**0.09794**	**0.1062**
PMF-m	0.1035	0.09358	0.08529	0.08856
PMF-SGD	0.1035	0.0943	0.08543	0.08947

Table 4. 算法效果对比

Datasets	Movielens			
Metrics	Re@5	Re@10	Pre@5	Pre@10
MF-m	**0.4807**	**0.4382**	**0.4251**	**0.4239**
PMF-m	0.452	0.4024	0.4102	0.3931
PMF-SGD	0.4423	0.3963	0.4072	0.3855

Table 5. 算法效果对比

Datasets	Netflix			
Metrics	Re@5	Re@10	Pre@5	Pre@10
MF-m	**0.5567**	**0.521**	**0.3976**	**0.4576**
PMF-m	0.5455	0.5031	0.395	0.4437
PMF-SGD	0.4842	0.4439	0.383	0.405

从排序推荐的角度，我们可以看到 MF-m 算法获得了压倒性的优势，原因可能在于该算法没有经过正则化，在数据集过大容易欠拟合时会取得更好的效果。

7 Summary and Discussions

本次实验，我完整得进行了一次推荐算法研究的过程（借助 R 语言）。即：学习理论-编写代码-确定度量-找数据集-调整参数-统计实验结果，很大得锻炼了实践能力，也得到了一些结论：

- 不同的数据集之间差别非常大。只有非常优秀的算法才能在很多数据集上显著好于其他算法。这也正说明了为什么好的文章都要在很多数据集上和许多最新的算法进行对比，并表示自己的算法更优。
- 切实得体会到了调参是个非常繁琐复杂的任务。如果两个算法的效果差距不大，其实真的很难搞清楚到底是由于参数没有调好，还是一个算法确实优于另一个。所以现在不是很敢相信那些宣称自己比其他算法好了百分之零点零几的文章。
- 其实正是因为上述两条，所以启发我们除了一些数值型的度量指标，是否可以采用一些其他方法来证明算法的优越性，比如直接展示某位用户的真实评分和预测评分的区别。
- 同样，由于调参的痛苦，我们是否可以采用一些非参数的方法，比如用贝叶斯方法消除参数，或者让模型自动找到一个最优的参数。比如[4]中建立了贝叶斯概率矩阵分解模型，使用 MCMC 的方法自动确定参数。

8 References

[1] Y. Koren. Factorization meets the neighborhood: Amultifaceted collaborative filtering model. In Proceeding of the 14th ACM SIGKDD international conferenceon Knowledge discovery and data mining, 2008.

[2] Y. Koren. Collaborative filtering with temporal dynamics. In KDD-09, 2009.

[3] R. Salakhutdinov and A. Mnih. Probabilistic matrixfactorization. In Advances in Neural Information Processing Systems (NIPS), volume 20, 2007.
[4] R. Salakhutdinov and A. Mnih. Bayesian probabilistic matrix factorization using markov chain monte carlo.In Proceedings of the Twenty-Fifth International Conference on Machine Learning (ICML 2008), Helsinki,Finland, 2008.
[5] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems.Computer 42, 8 (2009), 30–37.
[6]Huafeng Liu, Liping Jing, Yuhua Qian, and Jian Yu. 2019. Adaptive Local Low-rank Matrix Approximation for Recommendation. ACM Transactions on Information Systems *, *, Article * (June 2019), 32 pages.
[7]http://www.trustlet.org/downloaded_epinions.html
[8]https://grouplens.org/datasets/movielens/
[9]https://www.netflixprize.com
[10] Ruder S. An overview of gradient descent optimization algorithms[J]. arXiv preprint arXiv:1609.04747, 2016.
[11] http://www.utstat.toronto.edu/~rsalakhu/BPMF.html
[12] https://github.com/stxupengyu/Matrix-Factorization-for-Recommendation