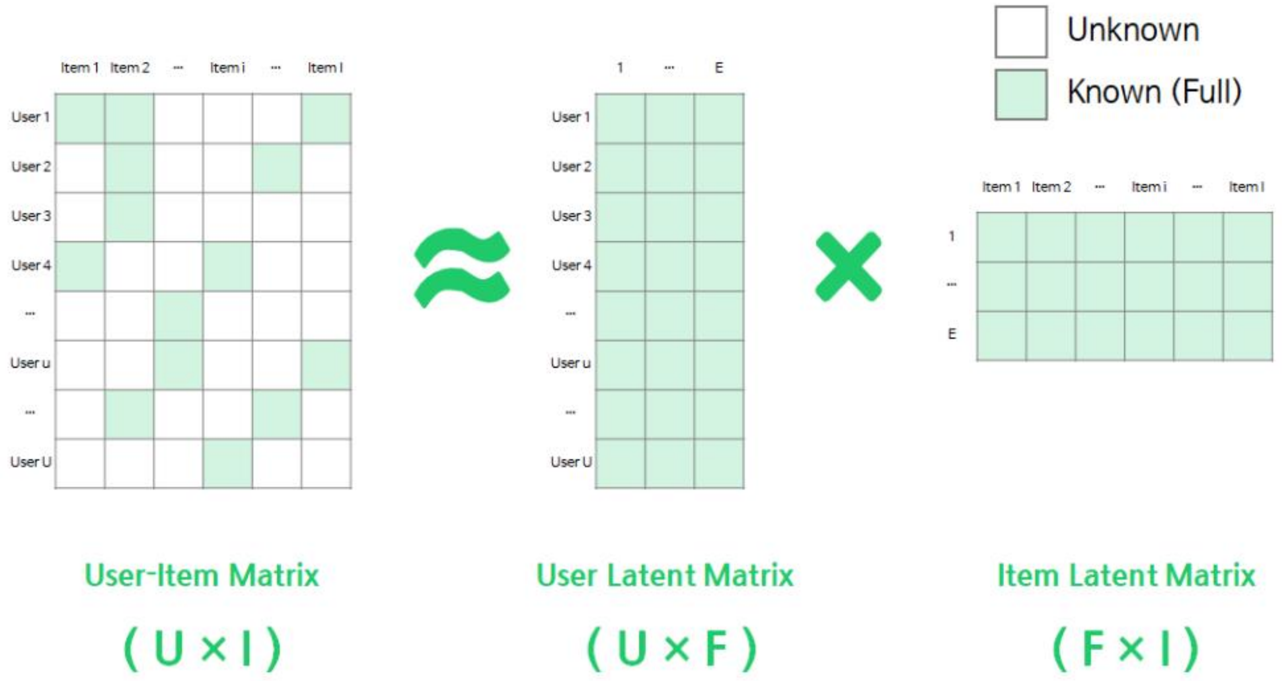
Probabilistic Matrix Factorization

지인선

2023.01.03

Contents

Introduction



One of the most popular approaches to collaborative filtering is based on low−dimensional factor models.

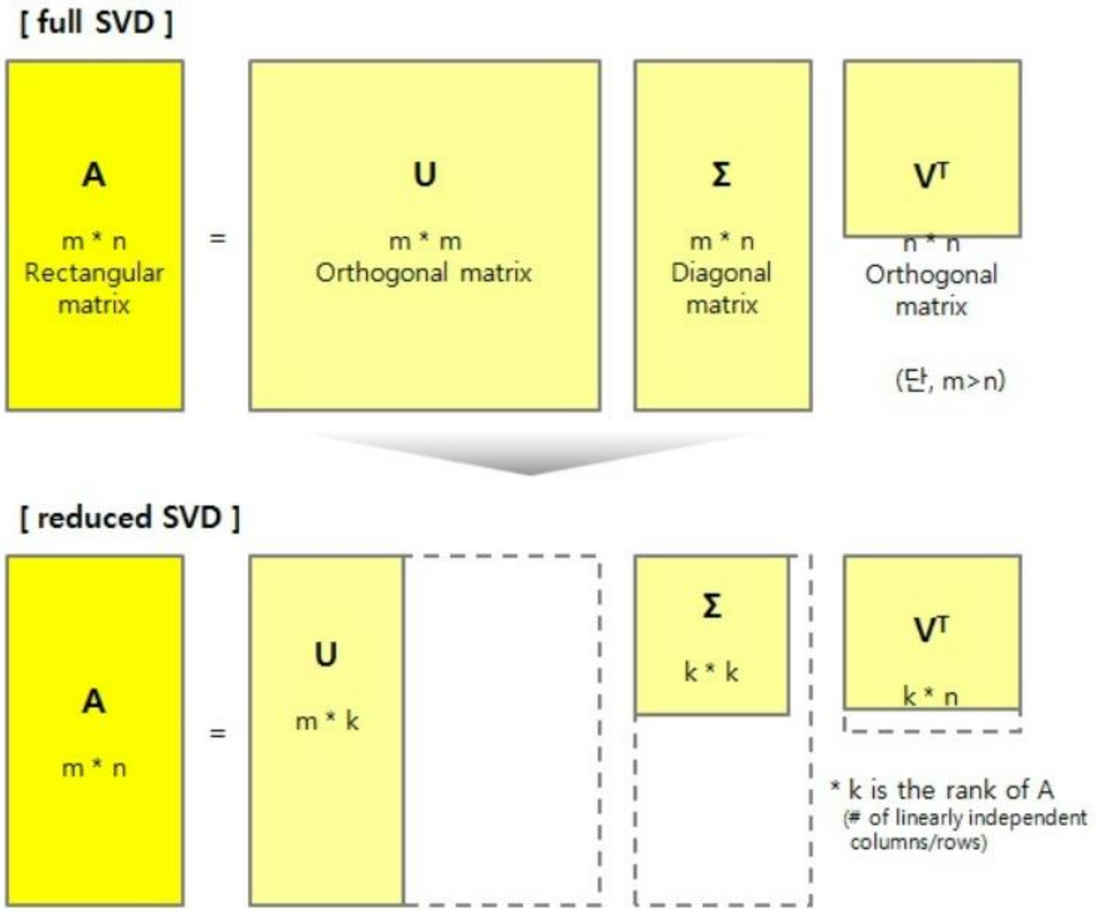
We want to approximate target matrix R (N×M matrix) with user matrix U (D×N matrix) and item matrix V (D×M matrix) such that $R \approx U^T V$

Introduction

SVD can find k–rank approximation of R which minimizes Frobenius norm.

But, since most real–world datasets are sparse, most entries in R will be missing.

In those cases, the sum–squared distance is computed only for the observed entries of the target matrix R, which results in a difficult non–convex optimization problem which cannot be solved using standard SVD implementations



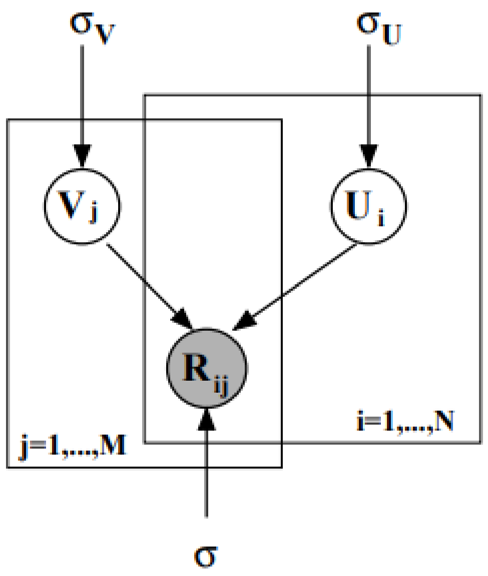
PMF

Suppose we have M movies, N users, and integer rating values from 1 to K.

Let R_{ij} represent the rating of user i for movie j, U and V be latent user and movie feature matrices, with column vectors U_i and V_j representing user−specific and movie−specific latent feature vectors respectively.

We define the conditional distribution over the observed ratings as

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



PMF

And we also place zero−mean spherical Gaussian priors on user and movie feature vectors

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

Since by Bayesian rule for parameter estimation,

$$p(\theta|\mathbf{X}, \alpha) = \frac{p(\mathbf{X}|\theta, \alpha)p(\theta|\alpha)}{p(\mathbf{X}|\alpha)} \propto p(\mathbf{X}|\theta, \alpha)p(\theta|\alpha)$$

the posterior distribution of U, V is proportional to

$$p(U, V|R, \sigma^2) \propto p(R|U, V, \sigma^2)p(U|\sigma_U^2)p(V|\sigma_V^2)$$

PMF

Since

$$p(U,V|R,\sigma^2) \propto \prod_{i=1}^{N}\prod_{j=1}^{M}\left[\mathcal{N}(R_{ij}|U_i^\top V_j,\sigma^2)\right]^{I_{ij}} \prod_{i=1}^{N}\mathcal{N}(U_i|0,\sigma_U^2)\prod_{j=1}^{M}\mathcal{N}(V_j|0,\sigma_V^2)$$

the log−likelihood can be expressed as

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

where C is constant that does not depend on parameters

PMF

Maximizing this log−likelihood is equivalent to minimizing

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

$\lambda_U = \sigma^2/\sigma_U^2$, $\lambda_V = \sigma^2/\sigma_V^2$

Also, instead of using a simple linear−Gaussian model, we can use sigmoid function to make predictions stay in the range of valid rating value.

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

Automatic Complexity Control for PMF model

Capacity control is essential to making PMF models generalize well

Changing the dimensionality of feature vector ->

When the dataset is unbalanced, i.e. the number of observations differs significantly among different rows or columns, this approach fails since any single number of feature dimensions will be too high for some feature vectors and too low for others.

Finding suitable regularization parameters from a set of reasonable parameter value ->

Computationally expensive

Automatic Complexity Control for PMF model

Introducing priors for the hyperparameters and maximizing the log−posterior of the model over both parameters and hyperparameters allows model complexity to be controlled automatically based on the training data.

$$\ln p(U, V, \sigma^2, \Theta_U, \Theta_V | R) = \ln p(R|U, V, \sigma^2) + \ln p(U|\Theta_U) + \ln p(V|\Theta_V) + \ln p(\Theta_U) + \ln p(\Theta_V) + C,$$

Prior is Gaussian:

The optimal hyperparameters can be found in closed form if the movie and user feature vectors are kept fixed.

Alternates between optimizing the hyperparameters and updating the feature vectors using steepest ascent with the values of hyperparameters fixed.

Prior is a mixture of Gaussians:

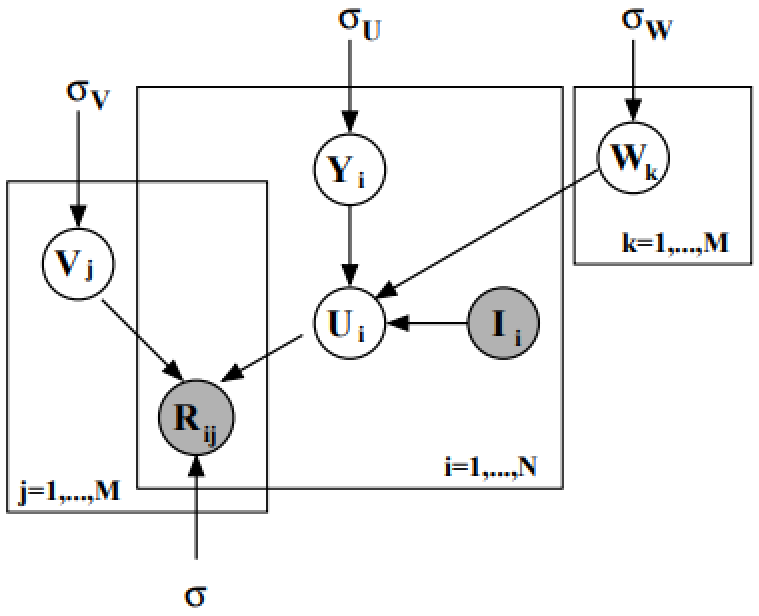
The hyperparameters can be updated by performing a single step of EM

Constrained PMF

Once a PMF model has been fitted, users with very few ratings will have feature vectors that are close to the prior mean, or the average user, so the predicted ratings for those users will be close to the movie average ratings

Let W be latent similarity constraint matrix.

$$U_i = Y_i + \frac{\sum_{k=1}^{M} I_{ik} W_k}{\sum_{k=1}^{M} I_{ik}}.$$



Y_i can be seen as s the offset added to the mean of the prior distribution for user i.

Constrained PMF is based on the assumption that users who have rated similar sets of movies are likely to have similar preferences.

Constrained PMF

We regularize the latent similarity constraint matrix W by placing a zero−mean spherical Gaussian prior on it.

$$p(W|\sigma_W) = \prod_{k=1}^{M} \mathcal{N}(W_k|0, \sigma_W^2 \mathbf{I})$$

And similarly, maximizing the log−likelihood is equivalent to minimizing

$$\begin{aligned} E \quad &= \quad \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{M} I_{ij}\left(R_{ij} - g\left(\left[Y_i + \frac{\sum_{k=1}^{M} I_{ik} W_k}{\sum_{k=1}^{M} I_{ik}}\right]^T V_j\right)\right)^2 \\ &+\frac{\lambda_Y}{2}\sum_{i=1}^{N} \| Y_i \|_{Fro}^2 + \frac{\lambda_V}{2}\sum_{j=1}^{M} \| V_j \|_{Fro}^2 + \frac{\lambda_W}{2}\sum_{k=1}^{M} \| W_k \|_{Fro}^2 \end{aligned}$$

Experiments

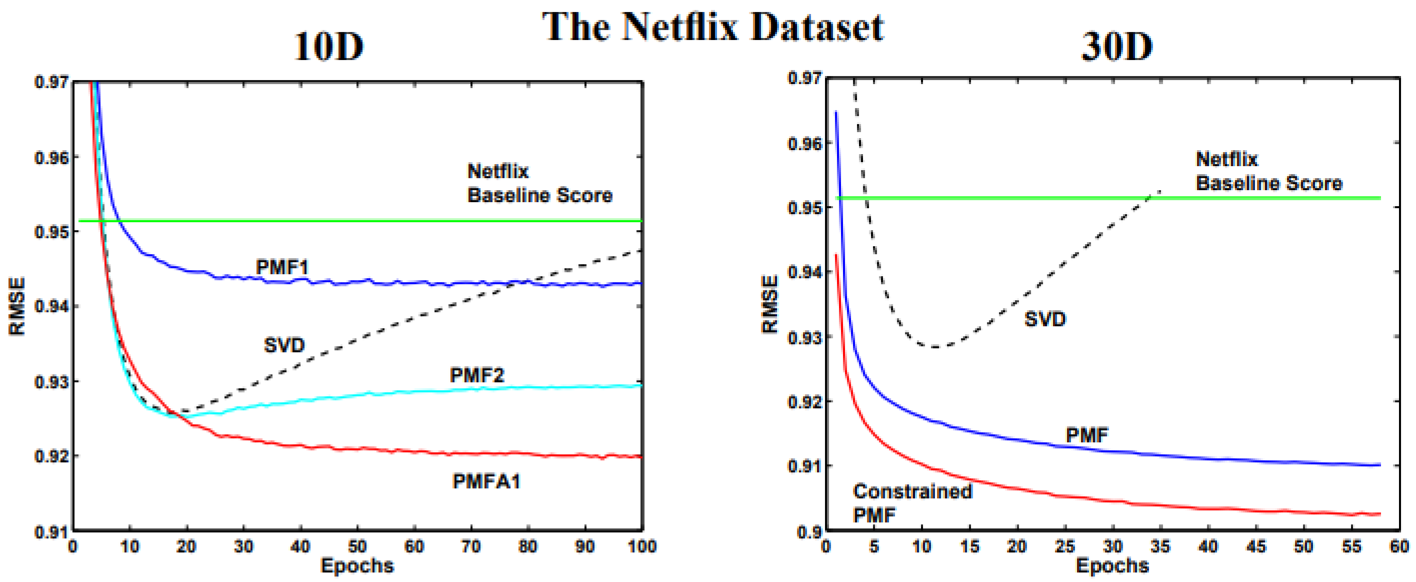
1. Description for Netflix dataset.

According to Netflix, the data were collected between October 1998 and December 2005 and represent the distribution of all ratings Netflix obtained during this period.

The training dataset consists of 100,480,507 ratings from 480,189 randomly−chosen, anonymous users on 17,770 movie titles.

To provide additional insight into the performance of different algorithms we created a smaller and much more difficult dataset from the Netflix data by randomly selecting 50,000 users and 1850 movies. The toy dataset contains 1,082,982 training and 2,462 validation user/movie pairs. Over 50% of the users in the training dataset have less than 10 rating.

Experiments

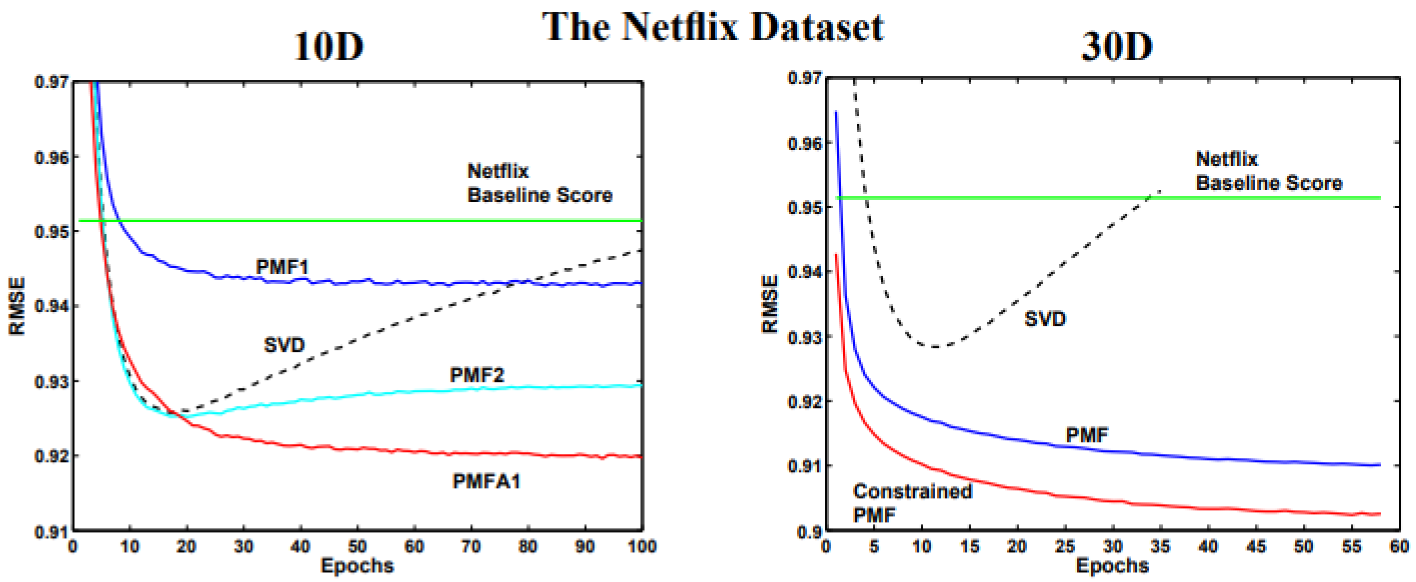


PMF1: $\lambda u = 0.01, \lambda v = 0.001$ PMF2: $\lambda u = 0.001, \lambda v = 0.0001$

PMFA1: adaptive prior ($\lambda u = 0.01, \lambda v = 0.001$, spherical covariance matrix)

PMFA2: adaptive prior ($\lambda u = 0.01, \lambda v = 0.001$, diagonal covariance matrix, not shown in figure)

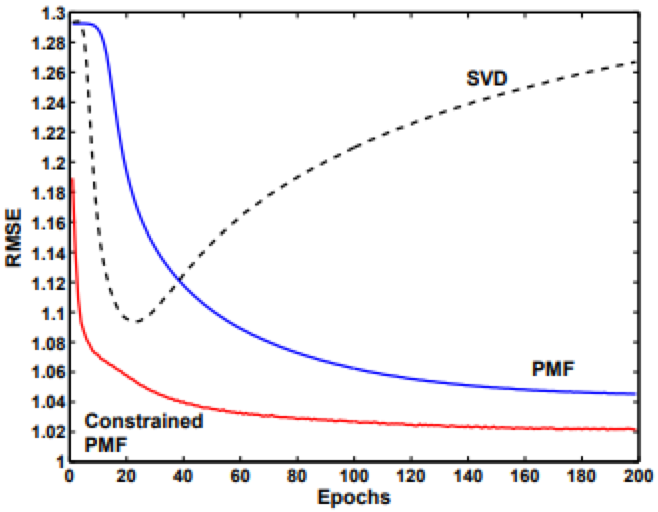
Experiments

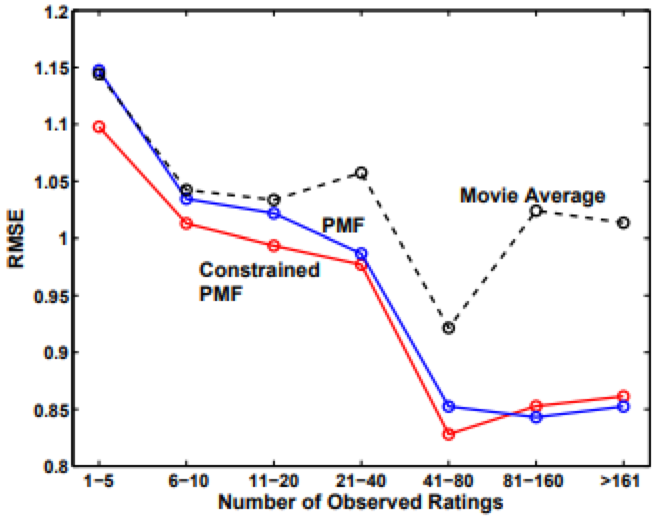


SVD model does almost as well as the moderately regularized PMF model before overfitting badly towards the end of training. The models with adaptive priors clearly outperform the competing models.

Experiments

Toy Dataset

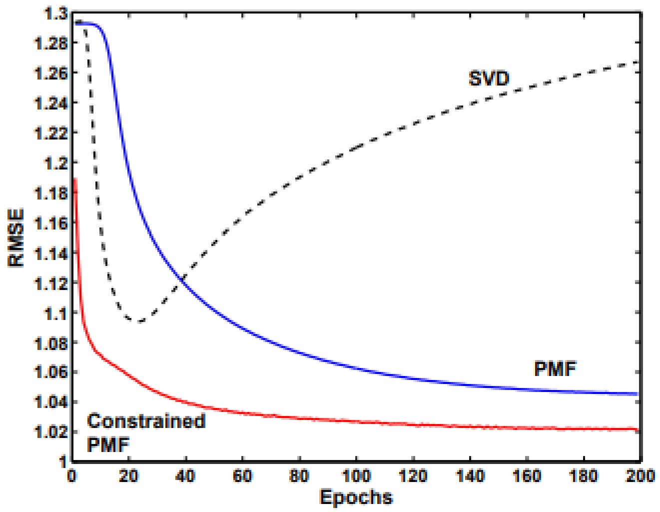


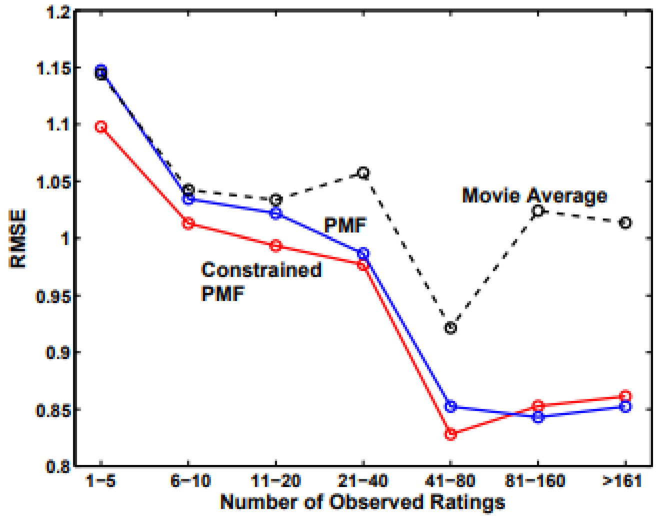


It is clear that the simple SVD model overfits heavily. The constrained PMF model performs much better and converges considerably faster than the unconstrained PMF model.

Experiments

Toy Dataset





Performance of the PMF model for a group of users that have fewer than 5 ratings in the training datasets is virtually identical to that of the movie average algorithm that always predicts the average rating of each movie.

The constrained PMF model, however, performs considerably better on users with few ratings.

As the number of ratings increases, both PMF and constrained PMF exhibit similar performance

Implementation of PMF

```
class PMF():
  def __init__(self, num_feat = 50, lr = 0.0002, momentum = 0.9, it = 200, lambU = 0.001, lambV = 0.001, lambW = 0.001, train_R = None, test_R = None, constrain = False):
    rns1 = np.random.RandomState(1234)
    rns2 = np.random.RandomState(123)
    self.num_feat = num_feat
    self.lr = lr
    self.momentum = momentum
    self.it = it
    self.lambU = lambU
    self.lambV = lambV
    self.lambW = lambW
    self.train_R = train_R
    self.test_R = test_R

    self.constrain = constrain
    self.I = copy.deepcopy(self.train_R)
    self.I[self.I > 0] = 1
    self.U = 0.1*rns1.randn(num_feat,train_R.shape[0])  # D*N
    self.V = 0.1*rns1.randn(num_feat,train_R.shape[1])  # D*M
    self.W = 0.1*rns2.randn(num_feat,train_R.shape[1])  # D*M
    self.T = None
    self.sigma_I = np.dot(self.I, np.ones(self.train_R.shape[1]))
```

Implementation of PMF

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

$$\begin{aligned} E \quad = \quad & \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{M} I_{ij}\left(R_{ij} - g\left(\left[Y_i + \frac{\sum_{k=1}^{M} I_{ik} W_k}{\sum_{k=1}^{M} I_{ik}}\right]^T V_j\right)\right)^2 \\ & + \frac{\lambda_Y}{2}\sum_{i=1}^{N} \| Y_i \|_{Fro}^2 + \frac{\lambda_V}{2}\sum_{j=1}^{M} \| V_j \|_{Fro}^2 + \frac{\lambda_W}{2}\sum_{k=1}^{M} \| W_k \|_{Fro}^2 \end{aligned}$$

```
def loss(self):
  if not self.constrain:
    return 0.5*np.sum(self.I*(self.train_R-np.dot(self.U.T, self.V))**2) + 0.5*self.lambU*np.sum(np.square(self.U)) + 0.5*self.lambV*np.sum(np.square(self.V))
  else:

    return 0.5*np.sum(self.I*(self.train_R-np.dot(self.U.T + (np.dot(self.W, self.I.T)/self.sigma_I).T, self.V))**2) + 0.5*self.lambU*np.sum(np.square(self.U)) + 0.5*self.lambV*np.sum(np.square(self.V))+0.5*self.lambW*np.sum(np.square(self.W))
```

Implementation of PMF

Gradients for PMF

```python
# derivate of Ui
grads_u = - (np.dot(self.I*(self.train_R-np.dot(self.U.T, self.V)), self.V.T)).T + self.lambU*self.U

# derivate of Vj
grads_v = - np.dot(self.U ,(self.I*(self.train_R-np.dot(self.U.T, self.V)))) + self.lambV*self.V
```

Gradients for Constrained PMF

```python
# derivate of U
grads_u = - (np.dot(self.I*(self.train_R-np.dot(self.U.T + (np.dot(self.W, self.I.T)/self.sigma_I).T, self.V)), self.V.T)).T + self.lambU*self.U

# derivate of V
grads_v = - np.dot(self.U ,(self.I*(self.train_R-np.dot(self.U.T + (np.dot(self.W, self.I.T)/self.sigma_I).T, self.V)))) + self.lambV*self.V

# derivative of W
grads_w = np.zeros(self.W.shape)

grads_w = -(np.dot( (self.I*(self.train_R-np.dot(self.U.T + (np.dot(self.W, self.I.T)/self.sigma_I).T, self.V))).T , np.dot(self.I / self.sigma_I.reshape(-1,1),self.V.T)))


# grads_w = - np.dot(self.U ,(self.I*(self.train_R-np.dot(self.U.T + (np.dots(self.W, self.I.T)/self.sigma_I).T, self.V))))
grads_w = grads_w.T + self.lambW*self.W
```

Implementation of PMF

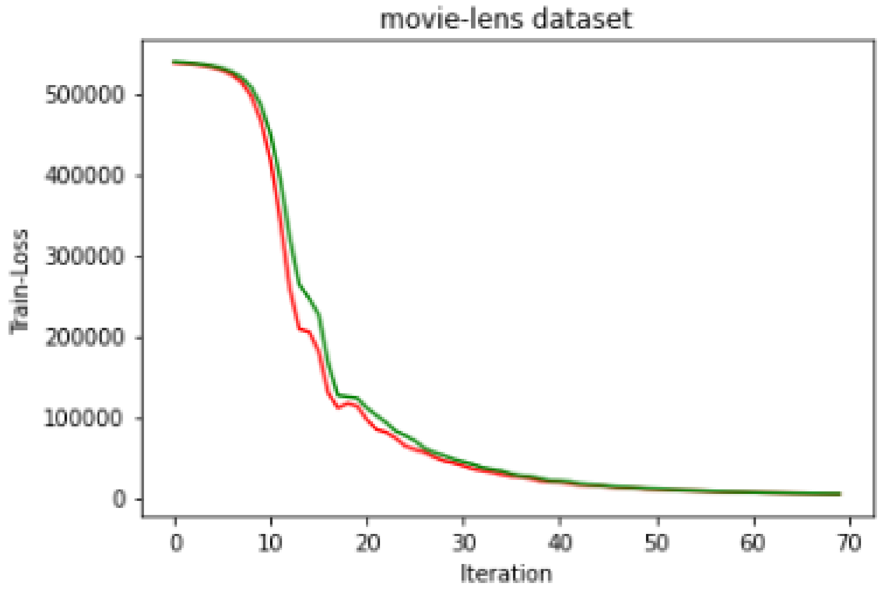
Movielens dataset: 610 users, 9724 items, 100836 ratings

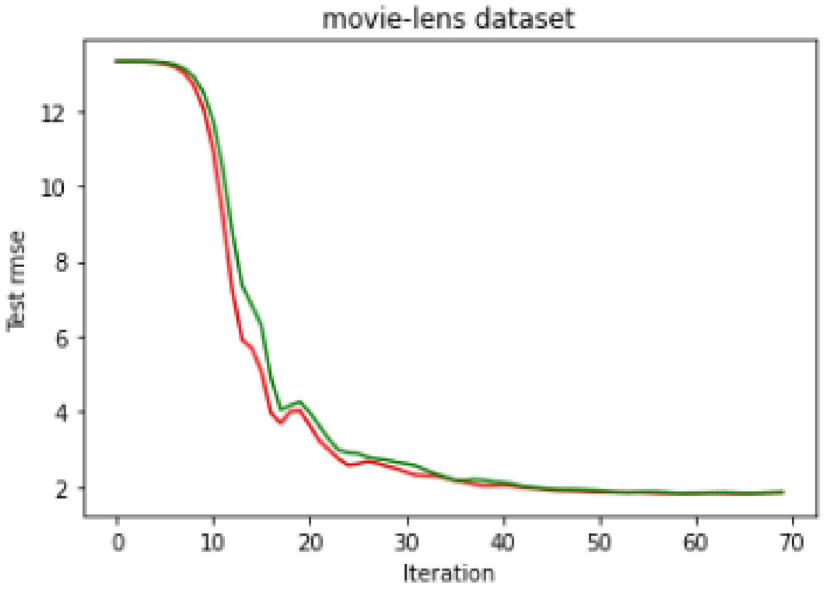
```
[14] import copy

     I = copy.deepcopy(train_R)
     I[I > 0] = 1
     print(I.shape[0])
     print(I.shape[1])
     print(I.sum())
     sum_I = I.sum(axis=1)
     print(np.count_nonzero(sum_I < 10))

     610
     9724
     80668.0
     0
```

userId	0	1	2	3	4	5	6	7	8	9	...	9714	9715	9716	9717	9718	9719	9720	9721	9722	9723
1	4.0	NaN	4.0	NaN	NaN	4.0	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
606	2.5	NaN	NaN	NaN	NaN	NaN	2.5	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
607	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
608	2.5	2.0	2.0	NaN	NaN	NaN	NaN	NaN	NaN	4.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
609	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
610	5.0	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

610 rows × 9724 columns

Implementation of PMF





Implementation of PMF

잘 된 점:

For-loop 없이 numpy를 이용해서 gradient 계산을 효율적으로 함.

잘 안된 점:

1. 논문과는 달리, constrained pmf와 pmf의 성능 차이가 극심하지 않음.

(sigmoid scalin을 하지 않음, dataset 차이 등등..)

2. PMFA를 구현하지 못함.