Collaborative Filtering for the NetFlix Prize

Masahiro Ono and Yang Zhang

December 5, 2007

Contents

1 Introduction

1.1 Project Overview

We implemented and evaluated two approaches to collaborative filtering: mixture model MLE and low-rank approximation. We analyze these approaches and evaluate them on a subset of the NetFlix Prize dataset. This report includes the theoretical basis of each approach and the empirical results of our experimental evaluation as well as complexity analysis and discussion of their feasibility for larger-scale problems.

1.2 Dataset

The NetFlix dataset is large: it contains 480,189 users, 17,770 movies, and 100,480,507 ratings. MATLAB's in-memory data management is not amenable to datasets of such scale, and working with such a large dataset incurs implementation challenges that are beyond the scope of our class proejct. Hence, we use a much smaller subset of the full Netflix Prize dataset consisting of 51 movies and 892 users, chosen so that the resulting rating matrix is fully packed. We then divide the rating matrix into training and test data. Since we have the fully packed rating matrix, we can arbitrary specify the *sparsity ratio*[1] of the training data to evaluate the dependency of the performance on the sparsity ratio.

In this report, we denote by $\mathbf{R}$ the rating matrix whose rows correspond to users and columns correspond to movies, such that entry $r_{i,j} \in \{1, \ldots, 5\}$ is the rating given by user i to movie j. We also denote by n_U and n_M the number of users and movies.

1.3 Performance Measure

The performance of each algorithm is measured by root-mean-squre error,

$$\text{RMSE} = \left[\sum_{(i,j) \in I_T} (\hat{r}_{i,j} - r_{i,j})^2 \right]^{\frac{1}{2}} \tag{1}$$

where I_T is the set of indices of the ratings included in the test data and $\hat{r}_{i,j}$ is the estimated rating of ith user for jth movie.

[1] *The sparsity ratio is the ratio of missing ratings to the size of the matrix. The original Netflix Prize data has a sparsity ratio of 99%.*

2 Mixture Model

We use a mixture model to cluster together ratings based on user types and movie types. There are various other mixture models for collaborative filtering; this model is the one presented in 6.867 lecture 15 and is known as the *flexible mixture model* [2]. A survey of various alternative models can be found in [1].

For this part of our project, we rigorously derived the E-M algorithm, implement the algorithm in MATLAB, and analyzed its results. In the following sections, we describe and justify the model, then derive the E-M algorithm and analyze its complexity. Next, we describe our implementation, and finally we evaluate its RMSE performance. Throughout these sections, we will identify key variables, but also provide complete definitions in the appendix for the reader's reference.

2.1 Overview

We say that *each user* has a distribution over user types (*e.g.*, mostly romantic, a bit of an intellectual), and each movie has a distribution over movie types (mostly horror, some action). Each pair of user type and movie type has a distribution over ratings (*e.g.*, romantics tend to give horrors low ratings). The likelihood of our model given the parameters is then

$$L(D \mid \theta) = \Pr(R_{1,1} = r_{1,1}, \ldots, R_{n_U,n_M} = r_{n_U,n_M} \mid \theta) \tag{2}$$

$$= \prod_{i,j} \sum_{u,m} \left[\Pr\left(R_{i,j} = r_{i,j} \,\middle|\, U_{i,j} = u, M_{i,j} = m, \theta^R_{u,m}\right) \cdot \Pr\left(U_{i,j} = u \,\middle|\, \theta^U_i\right) \cdot \Pr\left(M_{i,j} = m \,\middle|\, \theta^M_j\right) \right] \tag{3}$$

$$= \prod_{i,j} \sum_{u,m} \theta^R_{u,m}(r) \cdot \theta^U_i(u) \cdot \theta^M_j(m) \tag{4}$$

We see that there are three sets of parameters, and our approach is to use an E-M algorithm to perform maximum likelihood estimation of these parameters. A graphical model representation of our mixture model is in figure 1.

Each cell has its own user type and movie type. What does all this mean? Intuitively, we can think of each cell as the result of an *act* of rating. Hence, every time a user rates a movie, we are randomly sampling from among the user's multiple personalities (*e.g.*, the romantic in the user was rating this movie). We are also sampling from among the movie's multiple aspects (the user saw this movie as a horror movie). We will loosely use the term "rating" to mean either the act of the rating or the rating value itself; hopefully the meaning remains clear from the context.

It is natural to use multinomial distributions for generating the user types for a user, and the movie types for a movie. We are also using multinomial distributions to generate ratings given a user type and movie type; this we do for simplicity. We discuss this further in section 4.

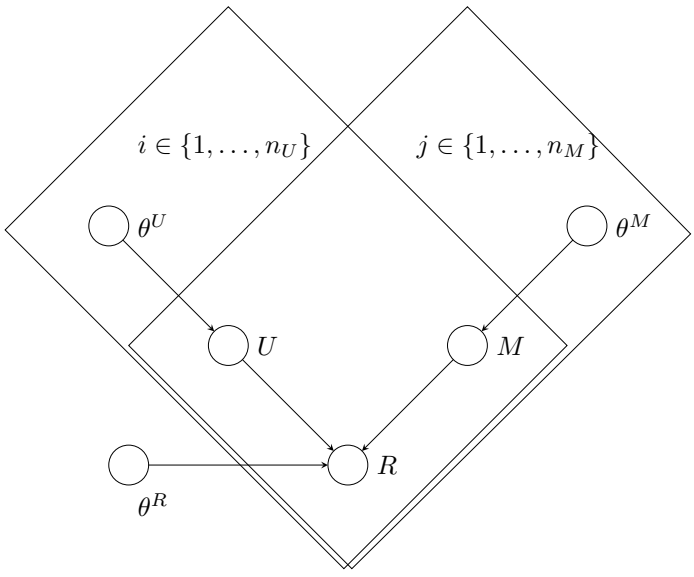

Figure 1: A graphical model representation of our mixture model.

Note that this model works for collaborative filtering for a static set of users and movies (filling in holes). New users/movies (augmenting our matrix with rows/columns that contain no data) would indeed be difficult to provide recommendations for regardless of model. However, even if these new rows/columns had data available, our approach

requires re-training the model, because each user and each movie has its own distribution over user types and movie types.

2.2 E-M Algorithm

2.2.1 M-step with complete data

To derive the E-M algorithm, it is helpful to start by assuming that we have "complete data"—that is, for each of the (observed) ratings i, j, we know precisely which user type u and movie type m the rating belongs to. With this information, we can directly and analytically find the parameters that maximize the likelihood of the data. We can equivalently optimize the log-likelihood: $\arg\max_\theta L(D \mid \theta) = \arg\max_\theta \log L(D \mid \theta)$.

$$
\begin{aligned}
l(D \mid \theta) &= \log L(D \mid \theta) && (5)\\
&= \sum_{i,j \in I} \begin{bmatrix} \log \Pr\left(R_{i,j} = r_{i,j} \,\middle|\, U_{i,j} = u_{i,j}, M_{i,j} = m_{i,j}, \theta^R\right) + \\ \log \Pr\left(U_{i,j} = u_{i,j} \,\middle|\, \theta^U_i\right) + \log \Pr\left(M_{i,j} = m_{i,j} \,\middle|\, \theta^M_j\right) \end{bmatrix} && (6)\\
&= \sum_{i,j \in I} \left[\log \theta^R_{u_{i,j}, m_{i,j}}(r_{i,j}) + \log \theta^U_i(u_{i,j}) + \log \theta^M_j(m_{i,j})\right] && (7)\\
&= \sum_{r,u,m,i,j} n_{r,u,m,i,j} \left[\log \theta^R_{u,m}(r) + \log \theta^U_i(u) + \log \theta^M_j(m)\right] && (8)
\end{aligned}
$$

Above, $n_{r,u,m,i,j}$ is 1 if rating i, j has rating value r, user type u, and movie type m, and 0 otherwise.

Implicit throughout the previous equations were a number of constraints due to the fact that we're dealing with probabilities. In particular,

$$
\sum_r \theta_{u,m}(r) = 1 \qquad \sum_u \theta_i(u) = 1 \qquad \sum_m \theta_j(m) = 1 \tag{9}
$$

To allow ourselves to optimize the equations analytically, we can incorporate these into the equation by using LaGrange multipliers (λs below):

$$
\begin{aligned}
&\sum_{r,u,m,i,j} n_{r,u,m,i,j} \left[\log \theta^R_{u,m}(r) + \log \theta^U_i(u) + \log \theta^M_j(m)\right] + \\
&\qquad \left\{\lambda^R_{u,m}\left(1 - \sum_r \theta^R_{u,m}(r)\right) \forall u, m\right\} + \left\{\lambda^U_i\left(1 - \sum_u \theta^U_i(u)\right) \forall i\right\} + \left\{\lambda^M_j\left(1 - \sum_m \theta^M_j(m)\right) \forall j\right\} \qquad (10)
\end{aligned}
$$

To find the maximizing value of a parameter, we find the zeros of the partial derivative with respect to that parameter. This turns out to be straightforward for the above expression, because the parameters are in separate terms, so that many terms in the sums and quantifiers "disappear."

For all u, i,

$$
\begin{aligned}
0 = \frac{\partial l(D \mid \theta)}{\partial \theta^U_i(u)} = \sum_{r,m,j} \frac{n_{r,u,m,i,j}}{\theta^U_i(u)} - \lambda^U_i \qquad &\Rightarrow \qquad \theta^U_i(u) = \frac{\sum_{r,m,j} n_{r,u,m,i,j}}{\lambda^U_i} && (11)\\
1 = \sum_u \theta^U_i(u) = \sum_u \frac{\sum_{r,m,j} n_{r,u,m,i,j}}{\lambda^U_i} \qquad &\Rightarrow \qquad \lambda^U_i = \sum_{r,u,m,j} n_{r,u,m,i,j} && (12)\\
&\therefore \theta^U_i(u) = \frac{\sum_{r,m,j} n_{r,u,m,i,j}}{\sum_{r,u,m,j} n_{r,u,m,i,j}} = \frac{n_{u,i}}{n_M} && (13)
\end{aligned}
$$

This makes intuitive sense, since $\theta^U_i(u) = \Pr\left(U_{i,j} = u \,\middle|\, \theta^U_i\right)$ for any movie j. The probability that we see the romantic in a user is the number of her other ratings where it was the romantic in her speaking, over her total number of ratings.

Similarly (almost symmetrically), we find:

$$
\forall m, j,\ \theta^M_j(m) = \frac{n_{m,j}}{n_U} \qquad \forall r, u, m,\ \theta^M_j(m) = \frac{n_{r,u,m}}{n_{u,m}} \tag{14}
$$

However, we do not have complete data—we do not have the above counts, since we do not know the user-/movie-type assignments. Since the incomplete data does not yield an analytical solution, we must use a numerical approach, namely an E-M algorithm. This algorithm iteratively updates the parameters in the M-step based on intermediately calculated posteriors in the E-step. Essentially, we use the posteriors to find the *expected* counts in the above equations.

2.2.2 E-step

For all $(i,j) \in I$, we calculate the posterior.

$$\begin{aligned}
&\Pr\left(U_{i,j}=u, M_{i,j}=m \,|\, R=\mathbf{R}, \theta\right) && && (15)\\
=&\Pr\left(U_{i,j}=u, M_{i,j}=m \,\middle|\, R_{i,j}=r_{i,j}, \theta_i^U, \theta_j^M, \theta_{u,m}^R\right) && \text{independencies can be seen in graphical model} && (16)\\
=&\frac{1}{Z}\begin{pmatrix}\Pr\left(R_{i,j}=r_{i,j} \,\middle|\, U_{i,j}=u, M_{i,j}=m, \theta_{u,m}^R\right)\\ \Pr\left(U_{i,j}=u, M_{i,j}=m \,\middle|\, \theta_i^U, \theta_j^M\right)\end{pmatrix} && \text{by Bayes' rule; } Z \text{ is a normalizer} && (17)\\
=&\frac{1}{Z}\begin{pmatrix}\Pr\left(R_{i,j}=r_{i,j} \,\middle|\, U_{i,j}=u, M_{i,j}=m, \theta_{u,m}^R\right)\\ \Pr\left(U_{i,j}=u \,\middle|\, \theta_i^U\right)\Pr\left(M_{i,j}=m \,\middle|\, \theta_j^M\right)\end{pmatrix} && \text{independencies in graphical model} && (18)\\
=&\frac{1}{Z}\theta_{u,m}^R(r_{i,j})\theta_i^U(u)\theta_j^M(m) && && (19)
\end{aligned}$$

Above,

$$Z = \Pr\left(R_{i,j}=r_{i,j} \,|\, \theta\right) = \sum_{u,m} \Pr\left(U=u, M=m \,\middle|\, R=r, \theta_i^U, \theta_j^M, \theta^R\right) \tag{20}$$

The above independencies can be verified from the moralized ancestral graphs of our model, but de do not have space to illustrate these graphs here.

For any i, j, letting $U = U_{i,j}, M = M_{i,j}, R = R_{i,j}$, we can marginalize the above joint distribution over M and U to get the distributions over U and M, respectively:

$$\Pr\left(U=u \,\middle|\, R=r, \theta_i^U, \theta_j^M, \theta^R\right) = \sum_m \Pr\left(U=u, M=m \,\middle|\, R=r, \theta_i^U, \theta_j^M, \theta^R\right) \tag{21}$$

$$\Pr\left(U=u \,\middle|\, R=r, \theta_i^U, \theta_j^M, \theta^R\right) = \sum_m \Pr\left(U=u, M=m \,\middle|\, R=r, \theta_i^U, \theta_j^M, \theta^R\right) \tag{22}$$

2.2.3 M-step

Now we can use these posterior values to find expected counts in order to update the parameters.

For all $i \in \{1, \ldots, n_U\}$, for any $j \in \{1, \ldots, n_M\}$,

$$\begin{aligned}
\mathrm{E}\left[n_{u,i} \mid \mathbf{R}, \theta\right] =& \mathrm{E}\left[\sum_{r,m,j} n_{r,u,m,i,j} \mid R=\mathbf{R}, \theta\right] && && (23)\\
=& \sum_{r,m,j} \mathrm{E}\left[n_{r,u,m,i,j} \mid R=\mathbf{R}, \theta\right] && && (24)\\
=& \sum_{r,m,j} \left[0 + 1 \cdot \Pr\left(R_{i,j}=r, U_{i,j}=u, M_{i,j}=m \,|\, R=\mathbf{R}, \theta\right)\right] && && (25)\\
=& \sum_j \Pr\left(U_{i,j}=u \,|\, R=\mathbf{R}, \theta\right) && \text{marginalized } M\text{, observed } R && (26)\\
=& \sum_j \Pr\left(U_{i,j}=u \,\middle|\, R_{i,j}=r_{i,j}, \theta_{u,m}^R, \theta_i^U, \theta_j^M\right) && \text{independencies in graph} && (27)\\
\theta_i^{U\prime}(u) =& \mathrm{E}\left[\frac{n_{u,i}}{n_M} \mid \theta\right] && \text{by Eq. 13} && (28)\\
=& \frac{1}{n_M}\sum_j \Pr\left(U_{i,j}=u \,\middle|\, R_{i,j}=r_{i,j}, \theta_{u,m}^R, \theta_i^U, \theta_j^M\right) && \text{weighted sum over the row} && (29)
\end{aligned}$$

Similarly, for all $j \in \{1, \ldots, n_M\}$,

$$\theta_j^{M\prime}(m) = \frac{1}{n_U}\sum_i \Pr\left(M=m \,\middle|\, R_{i,j}=r_{i,j}, \theta_{u,m}^R, \theta_i^U, \theta_j^M\right) \tag{30}$$

Finally, for all $u \in \{1, \ldots, k_U\}, m \in \{1, \ldots, k_M\}$,

$$\theta^{R}_{u,m}{}'(r) = \frac{\mathrm{E}\left[n_{r,u,m} \mid \theta\right]}{\mathrm{E}\left[n_{u,m} \mid \theta\right]} \tag{31}$$

$$= \frac{\sum_{i,j:r_{i,j}=r} \Pr\left(U_{i,j} = u, M_{i,j} = m \,\middle|\, R_{i,j} = r_{i,j}, \theta^{R}_{u,m}, \theta^{U}_{i}, \theta^{M}_{j}\right)}{\sum_{i,j} \Pr\left(U_{i,j} = u, M_{i,j} = m \,\middle|\, R_{i,j} = r_{i,j}, \theta^{R}_{u,m}, \theta^{U}_{i}, \theta^{M}_{j}\right)} \tag{32}$$

$$\tag{33}$$

2.3 Complexity Analysis

Here is a break-down of various dimensions of our model and algorithm. Each parameter involves "weighted sums" over some number of components of the joint posterior over U, M given the data R; the third column is the order of the number of terms in these summations (the cost of the summations). The fourth column is the total complexity, the product of the second and third columns. Note that in all of the following, we are providing a loose upper bound; the expense of this algorithm is more tightly bounded by the number of observed ratings in the matrix I, rather than the matrix's dimensions n_U and n_M.

component	no. parameters	no. posterior components	complexity
θ_R	$(k_R - 1) \cdot k_U \cdot k_M$	$\mathrm{O}(n_U \cdot n_M)$	$\mathrm{O}(k_R \cdot k_U \cdot k_M \cdot n_U \cdot n_M)$
θ_U	$(k_U - 1) \cdot n_U$	$\mathrm{O}(k_M \cdot n_M)$	$\mathrm{O}(k_U \cdot n_U \cdot k_M \cdot n_M)$
θ_M	$(k_M - 1) \cdot n_M$	$\mathrm{O}(k_U \cdot n_U)$	$\mathrm{O}(k_M \cdot n_M \cdot k_U \cdot n_U)$

(A multinomial of k possible outcomes needs only $k - 1$ parameters.)

The posterior joint distribution over U, M given the data R is in Eq. 19. This table is of size $k_U \cdot k_M \cdot k_R \cdot n_U \cdot n_M$, and each cell takes $\mathrm{O}(1)$ to compute. However, in our algorithm, we pruned the R dimension; it is unnecessary because we do not need to calculate the probabilities for all values of R, but only the values of R that are present in $\mathbf{R}$. This is the largest table in our algorithm, and thus it represents the asymptotic space usage.

2.4 Experimental Evaluation

We implemented the E-M algorithm in MATLAB. Despite our efforts to optimize the execution speed by fully vectorizing all operations, the algorithm takes up to 100 seconds (depending on the size of the parameter set θ) when run on even our reduced dataset; nonetheless, it allowed us to rapidly prototype a working implementation.

In addition to performing the E-step and M-step, each round we also calculate the likelihood of the data. (This turns out to share much of the same computation as the E-step, so we reuse this to avoid inflating the execution time.) We capped our algorithm to iterate 30 times at most—we found that it rarely needs to iterate this many times—and it additionally halts once the likelihood stops changing by any significant amount. Here are some measured execution times; notice the super-quadratic (degree-4) growth in execution time as the parameter set size grows quadratically, as predicted in the complexity analysis.

k_U, k_M	time (s)
2, 2	0.5
4, 4	1.5
8, 8	5.8
16, 16	22
32, 32	96

We partitioned the cells of $\mathbf{R}$ into a training set matrix and a test set matrix using sparsity ratios of 0.2, 0.5, and 0.7. The EM algorithm was repeated three times, and we took the result parameters that yielded the largest likelihood. The reason for this is that EM does not guarantee it will find the global maximum; where it ends up depends on how the parameters were initialized. The general way to deal with this is to randomly initialize the likelihood parameters, so we randomly initialize θ^R. Note that no two components therein must be equal, otherwise they would have the exact same likelihood of generating any of the points (*i.e.*, $\forall u, m, r, r'$, $\theta^{R}_{u,m}(r) \neq \theta^{R}_{u,m}(r')$). θ^U and θ^M *are* uniformly initialized; we are essentially starting with an "unbiased" prior (from a Bayesian perspective, we are doing the same thing as finding the maximum *a posteriori* parameters assuming a uniform prior).

In choosing the number of user types and movie types, an approach would be to try some different values and see how well they optimize some metric, settling with the value that best optimizes this metric. However, if we simply let the metric be the likelihood, then we can simply blow up the number of parameters: have as many user types as there are users, and as many movie types as there and number of movie types will equal to the number of cell types.

One common and easy-to-use metric is the *Bayesian information criterion* (BIC), also known as the *Schwarz information criterion* (SIC). The formula for the BIC is

$$-2 \cdot \log L\left(R = \mathbf{R} \,\middle|\, \hat{\theta}\right) + k \log n \tag{34}$$

where k is the dimensionality of the model space and $n = |I|$ is the number of rating observations. The model with the lower value of BIC is the one to be preferred: increasing the log-likelihood decreases the BIC, but increasing the number of parameters increases (penalizes) the BIC.

In our model, we have

$$l\left(R = \mathbf{R} \,|\, \theta\right) = \sum_{i,j \in I} \begin{bmatrix} \log \Pr\left(R_{i,j} = r_{i,j} \,\middle|\, U_{i,j} = u_{i,J}, M_{i,j} = m_{i,j}, \theta^R\right) + \\ \log \Pr\left(U_{i,j} = u_{i,j} \,\middle|\, \theta_i^U\right) + \\ \log \Pr\left(M_{i,j} = m_{i,j} \,\middle|\, \theta_j^M\right) \end{bmatrix} \tag{35}$$

$$= \sum_{i,j \in I} \left[\log \theta^R_{u_{i,j}, m_{i,j}}(r_{i,j}) + \log \theta_i^U(u_{i,j}) + \log \theta_j^M(m_{i,j})\right] \tag{36}$$

$$\tag{37}$$

Here is a table of our final results. "sparsity" is the sparsity ratio; "best LL" is the best log-likelihood we saw across all runs; and "RMSE (ML)" is the RMSE when considering the most likely rating, whereas "RMSE (WM)" is the RMSE when considering the weighted average rating (weighted by their likelihoods).

sparsity	k_U	k_M	best LL	RMSE (ML)	RMSE (WM)	BIC penalty	BIC score
0.20000	2	2	−11682.49021	0.94946	0.94838	8676.09233	−20358.58254
0.20000	4	4	−11712.12555	0.94946	0.94898	17534.26424	−29246.38979
0.20000	8	8	−11714.34409	0.94946	0.94902	35796.84683	−47511.19092
0.20000	16	16	−11714.61355	0.94946	0.94904	74506.96705	−86221.58060
0.20000	32	32	−11714.65863	0.94946	0.94904	160667.02767	−172381.68631
0.50000	2	2	−29003.86587	0.95235	0.94102	9559.75354	−38563.61941
0.50000	4	4	−29259.87174	0.95198	0.95034	19320.13149	−48580.00323
0.50000	8	8	−29277.39835	0.95198	0.95070	39442.76065	−68720.15900
0.50000	16	16	−29279.63470	0.95198	0.95075	82095.51200	−111375.14670
0.50000	32	32	−29279.98302	0.95198	0.95076	177030.98678	−206310.96980
0.70000	2	2	−39549.00019	0.92600	0.91623	9879.84227	−49428.84246
0.70000	4	4	−40926.65390	0.95017	0.94795	19967.02646	−60893.68036
0.70000	8	8	−40990.95403	0.95017	0.94867	40763.42059	−81754.37462
0.70000	16	16	−41000.51714	0.95017	0.94884	84844.31182	−125844.82896
0.70000	32	32	−41001.29407	0.95017	0.94885	182958.50623	−223959.80030

Our results show that the EM algorithm on the whole does not perform so well, always achieving roughly the same RMSE and likelihood on the test dataset, regardless of the number of user types and movie types. The results seemed suspicious, so we re-implemented the algorithm multiple times (in search of both correctness and speed), but always we found the same results, so we now understand that the problem lies in the approach.

We investigated these results in detail and extensively, and found a number of phenomena. The first thing we noticed was that the final parameters (in any of our runs) almost always causes our algorithm to predict the rating 4 (and occasionally some 5s), regardless of user and cell. This is likely caused by the fact that our parameters/distributions $\Pr\left(U \,\middle|\, \theta_i^U\right)$ and $\Pr\left(M \,\middle|\, \theta_j^M\right)$ are always nearly uniform, and our distribution $\Pr\left(R \,|\, U, M\right)$ is a more "natural" distribution that more often than not gives the most probabily to 4, followed by 5. It turns out that over 70% of the ratings in our matrix are either 4 or 5, which explains this.

In our results, the BIC penalty dominates the final BIC score; this is because the likelihood changes only negligibly compared to the BIC penalty. The penalty is high because there are many parameters to be estimated; the complexity of this model is high.

3 Low-rank Approximation

3.1 Overview

The goal of the learning algorithm presented in this section is to find a low-rank approximation of the sparse rating matrix

$$R \approx MU \tag{38}$$

where M is a n_M-by-d movie property matrix, U is a d-by-n_U user property matrix, and d is the dimension of the low-rank approximation. Due to the missing entries of R, M and U cannot be obtained from singular value decomposition. Instead, the algorithm learns M and U using linear regression and an EM-like alternate update algorithm.

The time and space complexity of the algorithm is proportional to the *sum* of the number of users and movies $(n_U + n_M)$ (not the product!), so it is tractable even for the large data set of the Netflix Prize. However, it strongly overfits to the training data especially when the rating matrix is sparse.

3.2 Learning Algorithm

At the beginning of the algorithm, M is randomly initialized and fixed. Then the i-th column of U is learned by regular linear regression as follows:

$$\boldsymbol{u}_i = (\check{M}^T \check{M})^{-1} \check{M}^T \check{\boldsymbol{r}}_i \tag{39}$$

where $\check{\boldsymbol{r}}_i$ is a column vector constructed from the ith column of the rating matrix R, by excluding the missing elements from it. For example, if the ith column of R is $[2 \quad \circ \quad 4 \quad \circ \quad \circ \quad 5 \quad \circ]^T$ where"$\circ$" means missing element, then $\check{r}_i = [2 \quad 4 \quad 5]^T$. $\check{M}$ is a matrix with the corresponding rows of M.

Then in the next step, U is fixed and each row of M is updated in the exactly same manner. These steps are repeated and U and M are alternately updated until the training RMSE converges, in a similar fashion as the EM algorithm.

The important fact is that, by repeating this iteration, the training RMSE monotonically decreases, and eventually converges to the (local) optimum. Linear regression finds the parameter which minimizes the log-likelihood given the following normal distribution with arbitrary fixed variance σ^2;

$$r_{i,j} \sim N(\vec{m}_i \cdot \boldsymbol{u}_j, \sigma^2) \tag{40}$$

where $\vec{m}_i$ is the i-th row of M and $\boldsymbol{u}_j$ is j-th column of U.

Thus, just as in the EM algorithm, the log-likelihood of the training data monotonically decreases in each iteration. With $\sigma = 1/\sqrt{2}$, the RMSE is exactly the same as the log likelihood. Thus the training RMSE monotonically decreases by iterations.

3.3 Estimation

Given the trained matrices U and M, the maximum likelihood estimation of the ratings can be easily obtained as follows:

$$\hat{r}_{i,j} = \vec{m}_i \cdot \boldsymbol{u}_j. \tag{41}$$

Although the actual ratings are given as integer numbers from one to five, the estimations by this algorithm are real numbers. We use the real number outputs without rounding them. However, since ratings are from one to five, the estimations above five are turned into five, and those below one are turned into one.

3.4 Result

We implemented this algorithm in MATLAB as well, and found that it worked successfully.

Since the initial value of M is randomly chosen, the result is stochastic. In fact it appears that the result is very sensitive to the initial value of M. Figure 2 shows the typical result of the training RMSE and the test RMSE as well as the RMSE when the estimation is simply the average rating of each movie (i.e. zero-order estimation). The training RMSE monotonically decreases over iterations as we expected, but the test RMSE does not necessarily monotonically decrease. There is a large gap between the training RMSE and test RMSE, indicating that the algorithm strongly overfits to the training data.

Table 1 compares the average RMSE of ten runs with different dimensions of the low-rank approximation d and different sparsity ratios. Looking at the table column-wise, we see that the algorithm works better given training data with fewer missing ratings, which is an obvious result. An interesting result can be seen by looking at the table row-wise: a larger dimension results in a larger RMSE when the rating matrix is sparse, while it results in a smaller RMSE when the rating matrix is mostly filled. This is because the algorithm strongly overfits to the training data when the number of free parameters is large and the number of the available training data is small.

3.5 Complexity Analysis

Time complexity The most time-consuming part of the algorithm is the d-by-d matrix inversion $(\check{M}^T \check{M})^{-1}$ in Eq. 39. In practice, Gaussian elimination, which has a complexity of $\mathrm{O}(n^3)$, would be used to find the solution instead of complete matrix inversion. The inverse of a matrix is computed $(n_U + n_M)$ times in each iteration. Thus the time complexity of the algorithm is

$$\mathrm{O}\big(d^3 N(n_U + n_M)\big) \tag{42}$$

where N is the number of iterations.

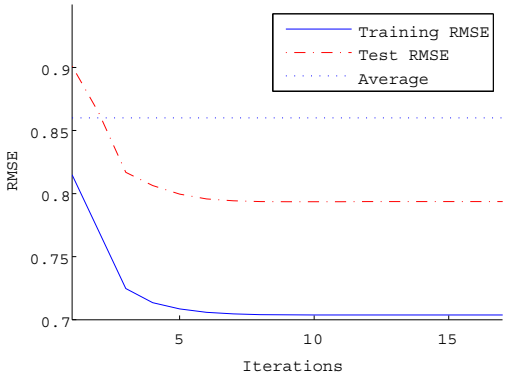

Figure 2: Typical result when $d = 2$ and (sparsity ratio)=0.5. The dotted line shows the RMSE when the estimation is simply the average rating of each movie (i.e. zero-order estimation)

Table 1: RMSE with different dimensions d and different sparsity ratio. The values are the average of ten runs.

		Sparsity ratio		
		0.2 (dense)	0.5	0.7 (sparse)
d	1	0.789	0.795	0.808
	2	0.772	0.790	0.859
	3	0.772	0.808	0.933

Space complexity Only U and M are stored during computation. Thus the space complexity is

$$\mathrm{O}\big(d(n_U + n_M)\big). \tag{43}$$

3.6 Estimated performance on the full Netflix Prize data

Computational cost It took about 15 seconds for 100 iterations with $n_U = 892$, $n_M = 51$, and $d = 3$ on an Intel Celeron 2.0 GHz CPU. For the full data set of the Netflix Prize, where $n_U = 480,189$ and $n_M = 17,770$, the computation for 100 iterations would take about 2 hours with $d = 3$. The required memory space would be about 11 MB. Thus, this is a tractable algorithm for the full Netflix Prize data.

RMSE In the actual Netflix prize data, 99% of the elements in the rating matrix are missing. With the small data set where $n_U = 892$, $n_M = 51$, the algorithm cannot run with 99% sparsity ratio since $(\check{M}^T\check{M})^{-1}$ in Eq. 39 becomes singular for most of the rows due to the lack of data. One thing we can say for sure from Table 1 is that the RMSE would be worse than 0.808 for the full Netflix Prize data set. Trying the algorithm on the full data is future work.

3.7 Principle Component Analysis

Low-rank approximation is equal to the Principle Component Analysis (PCA) when the rating matrix is packed. Thus, by plotting the trained matrix U and M with $d = 2$ on 2-D plain, the distribution of user types and movie types can be visualized as Figure 3.

In the user domain only one cluster is observed, although there are several outliers. On the other hand, three clear clusters can be observed in the movie domain.

Although low-rank approximation algorithm does not explicitly classify users and movies into discrete types as mixture model, it can captures the latent data structures.

3.8 Introducing Prior

The undesirable behavior of overfitting to the training data may be fixed by introducing the prior belief (in other words, regularization term). As the algorithm overfits to the training data, it tends to give stupid estimations such as 100 and -100, which is reduced to 5 or 1 in the algorithm eventually. If we introduce the prior that forces the predictions for the missing data to stay in the rage $1 \leq r \leq 5$, result would be improved. This is a future work.

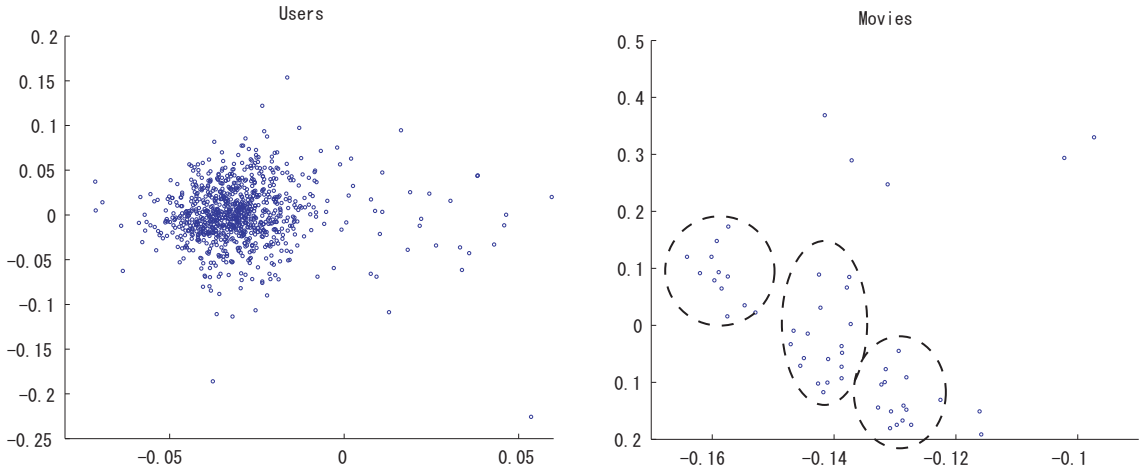

Figure 3: Principle Component Analysis of users and movies, obtained from singular value decomposition of packed rating matrix. Three clusters can be observed in movie domain.

4 Comparison

We have implemented two different methods for collaborative filtering in our final project.

The mixture model approach uses discrete multinomial distributions for modeling $\Pr(R \mid U, M)$, which is a natural way to handle the discrete movie ratings. Its shortcoming is that it regards the ratings as just "labels", which simply means different classes without any relation each other. However, "ratings" is a linear concept (*i.e.*, in the rating domain, 5 is closer to 4 than to 1), which cannot be captured by the multinomial distribution. Thus, seeing many ratings of 5 does not encourage the rating of 4 any more than the rating of 1.

On the other hand, low-rank approximation works in a continuous domain by handling discrete ratings as real numbers. Thus, it may make predictions out of the range $\{1, \ldots, 5\}$, such as 1000, 2.3, or -1. However, since it is a linear model, it can capture the linear concept of the ratings.

These differences between the two models leads to dramatically different results and performance, although both of them use similar optimization method (EM and alternating updates).

4.1 RMSE

The RMSE of the mixture model is worse than the low-rank approximation by large, for all scarcity ratio we have tried. Due to the inability of the mixture model to capture the linear concept of rating, it almost always predicts the rating 4, which appears most frequently in the training data. The resulting user-type and movie-type priors are all nearly uniform.

The overall disappointing performance of the EM algorithm was one of the motivations for looking into an alternative approach, namely low-rank approximation. It is interesting to note that most of the leading teams in the Netflix Prize competition appear to use a low-rank approximation approach.

4.2 Scalability

The computation time of both algorithms is on the same order for the small dataset we used, where $n_M = 51$ and $n_U = 892$. However, the expected computation time on the full dataset of 500K users and 17K movies is roughly 10 days for mixture versus 2 hours for low-rank approximation. The computation time is proportional to $n_U \cdot n_M$ in the mixture approach, whereas it is proportional to $n_U + n_M$ in the low-rank approximation approach. This is another reason why most of the leading teams in the Netflix Prize competition do not appear to use the mixture model.

5 Conclusion

We successfully implemented two approaches for collaborative filtering: an EM algorithm for estimating mixture model parameters, and a low-rank approximation algorithm. We analyze these approaches and evaluate them on a subset of the Netflix Prize dataset. We also included the theoretical basis of each approach and the empirical results of our experimental evaluation as well as complexity analyses and discussion of their feasibility for larger-scale problems.

The source code for our work is publicly available at:

http://code.google.com/p/netflix6867/

6 Appendix

6.1 Mixture Models

Here are the variables we're using for the mixture models:

- θ: the entire set of parameters.
  - $\theta_i^U(u)$: the probability distribution over user types u for a user i (think of this as a table). This is shared by all user-movie ratings for user i.
  - $\theta_j^M(m)$: the probability distribution over movie types m for a movie j. This is shared by all user-movie ratings for movie j.
  - $\theta_{u,m}^R(r)$: the probability distributions over ratings r for user type u and movie type m. This is shared by all user-movie ratings i, j.
- Dimensions and sets
  - n_U: number of users
  - n_M: number of movies
  - k_U: number of user types
  - k_M: number of movie types
  - I: the set of indices i, j of the ratings in the training set
  - I_T: the set of indices i, j of the ratings in the test set
- Indexes. This allows us to omit ranges in sums and quantifiers.
  - i: generally used to index over users
  - j: generally used to index over movies
  - u: generally used to index over user types
  - m: generally used to index over movie types
- Random variables
  - U_i: the user type of user i
  - M_j: the movie type of movie j
  - $R_{i,j}$: the rating user i gave for movie j

References

[1] C. Z. Rong Jin, Luo Si. A study of mixture models for collaborative filtering. URL http://www.cs.purdue.edu/homes/lsi/publications.htm.

[2] L. Si and R. Jin. A flexible mixture model for collaborative filtering, 2003. URL citeseer.ist.psu.edu/si03flexible.html.