Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.S064 Introduction to Machine Learning

Phase 2: recommender problems (lecture 10)

Collaborative filtering

Recommender problems are ubiquitous. We are interested recommending books, movies, or other products to users based on what we know about their history, what they do and don't like, and so on. Not surprisingly, recommender systems are in active use in companies such as Amazon.com (books and other products) or Netflix (movies). We will use the Netflix movie recommendation problem as a guiding example to illustrate these type of problems and how we can solve them.

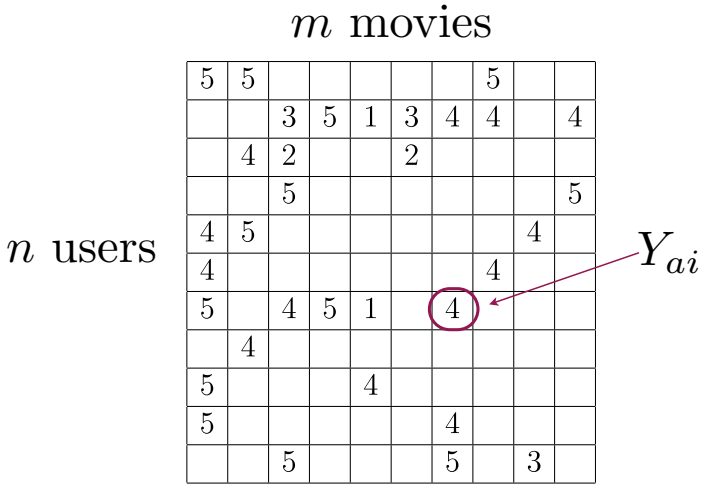

Figure 1: Data matrix Y for a collaborative filtering problem with users and movies.

Suppose we have n users and m movies as in Figure 1. We assume that each user has already rated some movies, i.e., assigned 1-5 stars to movies they have seen. These ratings can be collected into a data matrix Y. We will use $Y_{ai} \in \{1, \ldots, 5\}$ to denote the rating that user $a \in \{1, \ldots, n\}$ has given to movie $i \in \{1, \ldots, m\}$. For simplicity, we omit the rating scale and instead simply treat the matrix entries as real numbers, i.e., $Y_{ai} \in \mathcal{R}$. Our learning task is to complete the data matrix, i.e., predict ratings for all the unobserved entries of the matrix. Recommender problems that can be reduced to matrix (or tensor) completion problems are known as collaborative filtering problems. What makes this problem interesting from a learning point of view is that there are typically many users (e.g., $400,000$) and movies (e.g., $17,000$) but very few available

ratings (e.g., the matrix may be only 1% filled). The key idea behind collaborative filtering (CF) solutions is that we can borrow experience from other similar users to better predict the ratings for any particular user. The implication of this is that we should solve the problem as a matrix problem rather than as multiple independent prediction problems (one for each user).

We will consider here two common (and complementary) approaches to collaborative filtering: nearest-neighbor prediction and matrix factorization.

Nearest-neighbor prediction

Let's begin with a simple version of the problem where we have to make a prediction on a single missing entry Y_{ai}, i.e., how user a would rate movie i. We cannot solve this problem by looking at the past ratings of only user a. Our setup provides no information about movies (e.g., in terms of features). They are merely symbols associated with ratings. As a result, ratings for previous symbols provide no information about which rating to give to a new (so far unseen) symbol. The only way to solve the problem is to borrow experience from others.

A natural approach to exploiting the experience of others is to (somehow) combine ratings of users who have rated similarly to a in the past *and* have also rated movie i. To this end, we need to quantitatively measure how similar any two users are. A popular way to do this is via correlation, i.e., how their common ratings co-vary (with normalization). Let $CR(a,b)$ be the set of movies that both a and b have rated. Note that $CR(a,b)$ does not include i (or other items that a hasn't rated). We define the similarity as

$$\text{sim}(a,b) = \text{corr}(a,b) = \frac{\sum_{j \in CR(a,b)} (Y_{aj} - \tilde{Y}_a)(Y_{bj} - \tilde{Y}_b)}{\sqrt{\sum_{j \in CR(a,b)} (Y_{aj} - \tilde{Y}_a)^2} \sqrt{\sum_{j \in CR(a,b)} (Y_{bj} - \tilde{Y}_b)^2}} \in [-1,1] \quad (1)$$

where the mean ratings $\tilde{Y}_a = (1/|CR(a,b)|) \sum_{j \in CR(a,b)} Y_{aj}$ have to be computed anew for each comparison as they are based on shared ratings.

Similarity comparisons are often accurate or useful only when users are quite close. For example, it is unadvisable to use ratings from nearly uncorrelated users. For this reason (and for efficiency), we only find a small set of k other users who are most similar and who have also rated movie i. Let $kNN(a,i)$ denote this set of other users. The resulting nearest-neighbor prediction is a weighted combination of these users' ratings:

$$\hat{Y}_{ai} = \bar{Y}_a + \frac{\sum_{b \in kNN(a,i)} \text{sim}(a,b)(Y_{bi} - \bar{Y}_b)}{\sum_{b \in kNN(a,i)} |\text{sim}(a,b)|} \quad (2)$$

where, e.g., $\bar{Y}_a$ is now the mean across all the available ratings for a. Note that the prediction rule combines in a weighted manner how much other users' ratings for i deviate from their overall means, i.e., it uses $(Y_{bi} - \bar{Y}_b)$, and adds these deviations to

the user a's mean $\bar{Y}_a$. As a result, the rule is less sensitive to how the rating style changes from one user to another. For example, some users may give only 1s and 2s while others are more positive, primarily using 4s and 5s. The overall differences in the level (positiveness) will not affect the prediction. Whether the ratings are spread across the 1-5 scale, or vary only little around the mean, does have an affect, however.

Matrix factorization

We can think of collaborative filtering problem as a constrained matrix estimation problem. Constraints are necessary since we observe only a small fraction of entries of Y. Let X denote the matrix we use to approximate Y, in particular the missing entries. To begin with, let's see what happens in the absence of any constraints on X. We setup the problem as a regression problem, trying to force X to match the observed ratings. With regularization, we minimize

$$\sum_{ai \in D} (Y_{ai} - X_{ai})^2/2 + \frac{\lambda}{2} \sum_{ai} X_{ai}^2 \tag{3}$$

where D is the index set of observed entries. $\{Y_{ai},\, (a,i) \notin D\}$ are missing entries that we would like to use X to fill in. What is the solution $\hat{X}$? Unfortunately, it is not very useful

$$\hat{X}_{ai} \;=\; \begin{cases} \frac{1}{1+\lambda} Y_{ai}, & (a,i) \in D \\ 0, & (a,i) \notin D \end{cases} \tag{4}$$

In other words, we would predict zero for all unobserved entries!

A natural constraint to impose on a matrix is its rank. In other words, we try to approximate the $n \times m$ matrix Y with a matrix of low rank $X = UV^T$ where U is $n \times d$, V is $m \times d$, and $d \leq \min\{m, n\}$. For small d, the entries of $X = UV^T$ cannot be chosen arbitrarily (completely independently of each other). Indeed, setting a subset of the entries of X has consequences on how we can set the others. This is necessary for learning anything from such a sparse rating matrix Y.

Let's explore the factorization a little more. Let $u^{(a)} \in \mathcal{R}^d$ be the a^{th} row of U interpreted as a column vector, so that $u^{(a)}$ is $d \times 1$. Similarly, let $v^{(i)}$ be the i^{th} row of V, turned into a column vector. Put another way, $U = [u^{(1)}, \ldots, u^{(n)}]^T$ and $V = [v^{(1)}, \ldots, v^{(m)}]^T$. In this notation, $[UV^T]_{ai} = u^{(a)} \cdot v^{(i)}$. In other words, we can interpret the low rank factorization as means of introducing feature vectors for movies and users. Each user as a feature vector $u^{(a)}$ that hopefully captures user's preferences in this lower d-dimensional space. Similarly, each movie has a feature vector $v^{(i)}$ that (again hopefully) captures the relevant properties of movies as they pertain to ratings. The predicted rating X_{ai} is then the inner product (or similarity) between the corresponding feature vectors: $X_{ai} = u^{(a)} \cdot v^{(i)}$. The difficulty in finding such a factorization is that neither type of feature vector is known a priori but have to be estimated from the ratings

alone. Clearly, how we choose these feature vectors are inter-dependent problems. A good feature vector for a movie depends on what the user feature vectors are, and vice versa.

Our goal now is to find a low rank matrix $X = UV^T$, i.e., set the parameters (matrices) U and V, so that we match the observed ratings in Y

$$Y_{ai} \approx [UV^T]_{ai}, \quad (a,i) \in D \tag{5}$$

It is often advantageous to avoid satisfying all the constraints, e.g., if some of the ratings are mistakes. We might also suspect that we gain something by turning the estimating problem into a regularized least squares regression problem. More formally, we minimize

$$J(U,V) = \sum_{(a,i)\in D} (Y_{ai} - [UV^T]_{ai})^2/2 + \frac{\lambda}{2}\sum_{a=1}^{n}\sum_{k=1}^{d} U_{ak}^2 + \frac{\lambda}{2}\sum_{i=1}^{m}\sum_{k=1}^{d} V_{ik}^2 \tag{6}$$

$$= \sum_{(a,i)\in D} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\sum_{a=1}^{n} \|u^{(a)}\|^2 + \frac{\lambda}{2}\sum_{i=1}^{m} \|v^{(i)}\|^2 \tag{7}$$

Note that this problem reduces to the trivial example when $X = UV^T$ has full rank, i.e., when $d = \min\{n, m\}$ (each element can be chosen independently of each other). The smaller d we choose, the fewer adjustable parameters we have in U and V. d and the regularization parameter λ together constrain our solution.

How do we solve this regression problem? Here is a simple alternating minimization approach.

(0) Initialize the movie feature vectors $v^{(1)}, \ldots, v^{(m)}$ (e.g., randomly)

(1) Fix $v^{(1)}, \ldots, v^{(m)}$ and solve for $u^{(1)}, \ldots, u^{(n)}$. This is a regression problem and can be solved separately for each user a because the movie features are fixed. Let $D_{a,\cdot} = \{i \text{ s.t. } (a,i) \in D\}$ denote the observed entries for user a. Then, to solve for $u^{(a)}$, we minimize

$$\sum_{i\in D_{a,\cdot}} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|u^{(a)}\|^2 \tag{8}$$

This is the only part of $J(U,V)$ that depends on $u^{(a)}$. Note that this is a standard linear regression problem.

(2) Fix $u^{(1)}, \ldots, u^{(n)}$ and solve for $v^{(1)}, \ldots, v^{(m)}$. This is again a regression problem (note the symmetry), and can be solved separately for each movie i. Let $D_{\cdot,i} = \{a \text{ s.t. } (a,i) \in D\}$ be the set of users who have rated i. Then we minimize

$$\sum_{a\in D_{\cdot,i}} (Y_{ai} - u^{(a)} \cdot v^{(i)})^2/2 + \frac{\lambda}{2}\|v^{(i)}\|^2 \tag{9}$$

which (as expected) is again a standard linear regression problem.

Given that the feature vectors in U and V are inter-dependent, we have to iterate over steps (1) and (2). This procedure will monotonically decrease $J(U, V)$ and will converge to a locally optimal solution. But it may not necessarily find the best solution (cf. k-means). The initialization in step (0) matters quite a bit. For example, consider what will happen if we initialized all $v^{(i)}$ to zero (left as an exercise).