9
Matrix Factorization and Recommender Systems

9.1 Recommender Systems

Cataloging and recommender systems have always been an essential asset for consumers who find it difficult to choose from the vast scale of available goods. As early as 1876, the Dewey decimal system was invented to organize libraries. In 1892, Sears released their famed catalog to keep subscribers up to date with the latest products and trends, which amounted to 322 pages. Shopping assistants at department stores or radio disc jockeys in the 1940s are also examples of recommndations via human curation. In more contemporary times, bestseller lists at bookstores, or Billboard Hits list aim to capture what is popular among people. The modern recommender system paradigm now focuses on recommending products based on what is liked by people "similar" to you. In this long history of recommender systems, the common theme is that *people like to follow trends*, and recommender systems can help catalyze this process.

9.1.1 Movie Recommendation via Human Curation

Suppose we want to design a recommender system for movies. A human curator identifies r binary attributes that they think are important for a movie (*e.g.*, is a romance movie, is directed by Steven Spielberg, etc.) Then they assign each movie an r-dimensional *attribute vector*, where each element represents whether the movie has the corresponding attribute (*e.g.*, coordinate 2 will have value 1 if a movie is a "thriller" and 0 otherwise).

Now, using a list of movies that a particular user likes, the curator assigns an r-dimensional *taste vector* to a given user in a similar manner (*e.g.*, coordinate w will have value 1 if a user likes "thrillers" and 0 otherwise). With these concepts in mind, we can start with defining the affinity of a user for a particular movie:

Definition 9.1.1 (User Affinity). *Given a taste vector* $\mathbf{A}_i = (a_{i,1}, a_{i,2}, \ldots, a_{i,r})$ *for user* i *and the genre vector* $\mathbf{B}_j = (b_{1,j}, b_{2,j}, \ldots, b_{r,j})$ *for movie* j*, we define the* ***affinity of user*** i ***for movie*** j *as*

$$\mathbf{A}_i \cdot \mathbf{B}_j = \sum_{k=1}^{r} a_{i,k} b_{k,j} \tag{9.1}$$

Intuitively, this metric counts the number of attributes which are 1 in both vectors, or equivalently how many of the user's boxes are "checked off" by the movie. Mathematically, the affinity is defined as a dot product, which can be extended to matrix multiplication. Thus if we have a matrix $\mathbf{A} \in \mathbb{R}^{m \times r}$ where each of m rows is a taste vector for a user and a matrix $\mathbf{B} \in \mathbb{R}^{r \times n}$ where each of n columns is a genre vector for a movie, the (i, j) entry of the matrix product $\mathbf{M} = \mathbf{AB}$ represents the affinity score of user i for movie j.

We can also define an additional similarity metric:

Definition 9.1.2 (Similarity Metric). *Given taste vector* A_i *for user* i *and taste vector* $\mathbf{A}_j$ *for user* j*, we define the* ***similarity of user*** i ***and user*** j *as*

$$\sum_{k=1}^{r} a_{i,k} a_{j,k} \tag{9.2}$$

Similarly, the ***similarity of movie*** i ***and movie*** j *is defined as*

$$\sum_{k=1}^{r} b_{k,i} b_{k,j} \tag{9.3}$$

Finally, in practice, each individual is unique and has a different average level of affinity for movies (for example, some users like everything while others are very critical). This means that directly comparing the affinity of one user to another might not be helpful. One way to circumvent this problem is to augment (9.1) in Definition 9.1.1 as

$$\sum_{k=1}^{r} a_{i,k} b_{k,j} + a_{i,0} \tag{9.4}$$

with a bias term $a_{i,0}$.

Based on the affinity scores or similarity scores, the human curator will be able to recommend movies to users. This model design seems like it does the job as a recommender system. In practice, developing such models through human curation comes with a set of pros and cons:

- *Pros:* Using human curation allows domain expertise to be leveraged and this intuition can be critical in the development of a good model (*i.e.*, which attribute is important). In addition, a human curated model will naturally be interpretable.

- *Cons:* The process is tedious and expensive; thus it is difficult to scale. In addition, it can be difficult to account for niche demographics and genres and this becomes a problem for companies with global reach.

We conclude that while human curated models can certainly be useful, the associated effort is often too great.

9.2 *Recommender Systems via Matrix Factorization*

In this section, we provide another technique that can be used for recommender systems — matrix factorization. This method started to become popular since 2005.

9.2.1 *Matrix Factorization*

Matrix factorizations are a common theme throughout linear algebra. Some common techniques include LU and QR decomposition, Rank Factorization, Cholesky Decomposition, and Singular Value Decomposition.

Definition 9.2.1 (Matrix Factorization)**.** *Suppose we have some matrix* $\mathbf{M} \in \mathbb{R}^{m\times n}$*. A* ***matrix factorization*** *is the process of finding matrices* $\mathbf{A} \in \mathbb{R}^{m\times r}, \mathbf{B} \in \mathbb{R}^{r\times n}$ *such that* $\mathbf{M} = \mathbf{AB}$ *for some* $r < m, n$*.*

Unfortunately, these techniques become less directly applicable once we consider the case where most of the entries of $\mathbf{M}$ are missing (*i.e.*, a missing-data setting). As we saw in Section 9.1.1, this is very common in real-world applications — for example, if the (m, n) entry of M represents the rating of user m for movie n, most entries in M are missing because not everyone has seen every movie. What can we do in such a case?

In turns out, if we assume that M is a low-rank matrix (which is true for many high-dimensional datasets, as noted in Chapter 7), then we can consider an approximate factorization $\mathbf{M} \approx \mathbf{AB}$ on the known entries. We express this as the following optimization problem:

Definition 9.2.2 (Approximate Matrix Factorization)**.** *Suppose we have some matrix* $\mathbf{M} \in \mathbb{R}^{m\times n}$ *where* $\Omega \subset [m] \times [n]$ *is the subset of* (i, j) *where* M_{ij} *is known. An* ***approximate matrix factorization*** *is the process of finding matrices* $\mathbf{A} \in \mathbb{R}^{m\times r}, \mathbf{B} \in \mathbb{R}^{r\times n}$ *for some* $r < m, n$ *that minimize the loss function:*

$$L(\mathbf{A}, \mathbf{B}) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} (M_{ij} - (AB)_{ij})^2 \tag{9.5}$$

We denote the approximation as $\mathbf{M} \approx \mathbf{AB}$*.*

Notice this form is familiar: we are effectively trying to find optimal matrices $\mathbf{A}, \mathbf{B}$ which will minimize the MSE between known entries of M and corresponding entries in the matrix product $\mathbf{AB}$! One thing to note is that by calculating the matrix product $\mathbf{AB}$, we can "predict" entries of $\mathbf{M}$ that are unknown.

You can take the following result from linear algebra as granted.

Theorem 9.2.3. *Given $\mathbf{M} \in \mathbb{R}^{m \times n}$, we can find the matrix factorization $\mathbf{M} = \mathbf{AB}$, with $\mathbf{A} \in \mathbb{R}^{m \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times n}$ if and only if M has rank at most r. Also, we can find the approximate matrix factorization $\mathbf{M} \approx \mathbf{AB}$, with $\mathbf{A} \in \mathbb{R}^{m \times r}, \mathbf{B} \in \mathbb{R}^{r \times n}$ if and only if $\mathbf{M}$ is "close to" rank r.*

9.2.2 *Matrix Factorization as Semantic Embeddings*

Recall the setup in Section 9.1.1. But instead of calculating the affinity matrix $\mathbf{M}$ as the product of the matrices $\mathbf{A}, \mathbf{B}$, we will approach from the opposite direction. We will start with an affinity matrix $\mathbf{M} \in \mathbb{R}^{m \times n}$ (which is only partially known) and find its approximate matrix factorization $\mathbf{M} \approx \mathbf{AB}$. We can understand that $\mathbf{A} \in \mathbb{R}^{m \times r}$ represents a set of users and that $\mathbf{B} \in \mathbb{R}^{r \times n}$ represents a set of movies.

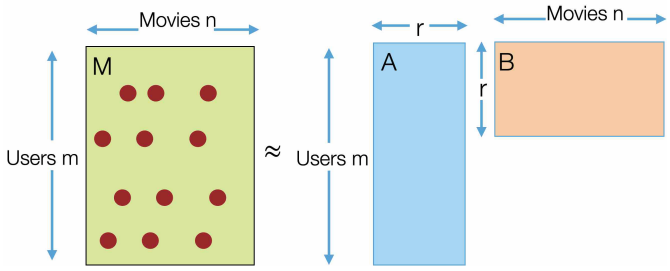

Figure 9.1: Matrix factorization on movie recommendations. Usually the inner dimension r would be much smaller than m, n.

Specifically, if we let A_{i*} denote the i-th row of $\mathbf{A}$ and B_{*j} denote the j-th column of $\mathbf{B}$, then A_{i*} can be understood as the *taste vector* of user i and B_{*j} can be understood as the *attribute vector* of movie j. One difference to note is that the output of a matrix factorization is real-valued, unlike the the $0/1$ valued matrices $\mathbf{A}, \mathbf{B}$ from Section 9.1.1. We can then use the vectors A_{i*} and B_{*j} to find similar users or movies and make recommendations.

Example 9.2.4. *Assume all columns of $\mathbf{B}$ have ℓ_2 norm 1. That is, $\|B_{*j}\|_2 = 1$ for all j. When the inner product $B_{*j} \cdot B_{*j'}$ of two movie vectors is actually 1, the two vectors are exactly the same! They have the same inner product with every user vector A_{i*} — in other words these movies have the same appeal to all users. Now suppose $B_{*j} \cdot B_{*j'}$ is not quite 1 but close to 1, say 0.9. This means the movie vectors are quite close but not the same. Still, their inner product with typical user vectors will not be too different. We*

conclude that two movies j, j' with inner product $B_{*j} \cdot B_{*j'}$ close to 1 tend to get recommended together to users. One can similarly conclude that high value of inner product between two user vectors is suggestive that the users have similar tastes.

9.2.3 *Netflix Prize Competition: A Case Study*

During 2006-09, DVDs were all the rage. Companies like Netflix were quite interested in recommending movies as accurately as possible in order to retain clients. At the time, Netflix was using an algorithm which had stagnated around $RMSE = 0.95$. [1] Seeking fresh ideas, Netflix curated an anonymized database of $100M$ ratings (each rating was on a $1-5$ scale) of $0.5M$ users for $18K$ movies. Adding a cash incentive of $\$1,000,000$, Netflix challenged the world to come up with a model that could achieve a much lower RMSE! [2] It turned out that matrix factorization would be the key to achieving lower scores. In this example, $m = 0.5M$, $n = 18k$, and Ω corresponds to the $100M$ ratings out of $m \cdot n = 10B$ affinities. [3]

After a lengthy competition, [4] the power of matrix factorization is on full display when we consider the final numbers:

- Netflix's algorithm: $RMSE = 0.95$
- Plain matrix factorization: $RMSE = 0.905$
- Matrix factorization and bias: $RMSE = 0.9$
- Final winner (an ensemble of many methods) : $RMSE = 0.856$

[1] $RMSE$ is shorthand for $\sqrt{MSE}$.

[2] This was an influential competition, and is an inspiration for today's hackathons, Kaggle, etc.

[3] Less than 1% of possible elements are accounted for by Ω.

[4] Amazingly, a group of Princeton undergraduates managed to achieve the second place!

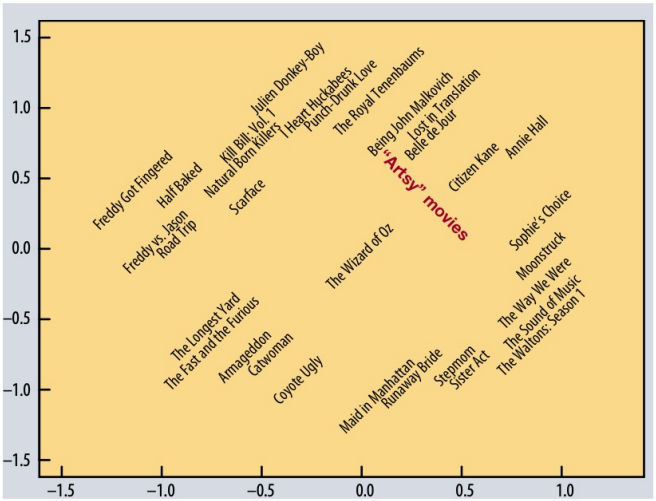

Figure 9.2: 2D visualization of embeddings of film vectors. Note that you see clusters of "artsy" films on top right, and romantic films on the bottom. *Credit: Koren et al.,* Matrix Factorization Techniques for Recommender Systems, *IEEE Computer* 2009.

9.2.4 *Why Does Matrix Factorization Work?*

In general, we need mn entries to completely describe a $m \times n$ matrix $\mathbf{M}$. However, if we find factor $\mathbf{M}$ into the product $\mathbf{M} = \mathbf{AB}$ of $m \times r$ matrix $\mathbf{A}$ and $r \times n$ matrix $\mathbf{B}$, then we can describe $\mathbf{M}$ with essentially only $(m+n)r$ entries. When r is small enough such that $(m+n)r \ll mn$, some entries of $\mathbf{M}$ (including the missing entries) are not truly "required" to understand $\mathbf{M}$.

Example 9.2.5. *Consider the matrix*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & * & 2 \\ 1 & 1 & * & * \\ 4 & * & 8 & * \\ 4 & * & * & * \end{bmatrix}$$

Is it possible to fill in the missing elements such that the rank of $\mathbf{M}$ *is* 1*? Since* $r = 1$*, it means that all the rows/columns of* $\mathbf{M}$ *are the same up to scaling. By observing the known entries, the second row should be equal to the first row, and the third and the fourth row should be equal to the the first row multiplied by* 4*. Therefore, we can fill in the missing entries as*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 4 & 4 & 8 & 8 \\ 4 & 4 & 8 & 8 \end{bmatrix}$$

It is not hard to infer that $\mathbf{M} = \mathbf{AB}$ *where* $\mathbf{A} = (1, 1, 4, 4)^T$ *and* $\mathbf{B} = (1, 1, 2, 2)$

Example 9.2.6. *Consider another matrix*

$$\mathbf{M} = \begin{bmatrix} 1 & 1 & * & * \\ 1 & 7 & * & * \\ 4 & * & * & 2 \\ * & 4 & * & * \end{bmatrix}$$

Is it possible to fill in the missing elements such that the rank of $\mathbf{M}$ *is* 1*? This time, the answer is no. Following a similar logic from Example 9.2.5, the second row should be equal to the first row multiplied by a constant. This is not feasible since* $M_{2,1}/M_{1,1} = 1$ *and* $M_{2,2}/M_{1,2} = 7$*.*

9.3 *Implementation of Matrix Factorization*

In this section, we look more deeply into implementing matrix factorization in an ML setting. As suggested in Definition 9.2.2, we can consider the process of approximating a matrix factorization to be an optimization problem. Therefore, we can use gradient descent.

9.3.1 Calculating the Gradient of Full Loss

Recall that for an approximate matrix factorization of a matrix $\mathbf{M}$, we want to find matrices $\mathbf{A}, \mathbf{B}$ that minimize the following loss:

$$L(\mathbf{A}, \mathbf{B}) = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} (M_{ij} - (AB)_{ij})^2 \qquad \text{(9.5 revisited)}$$

Here $(AB)_{ij} = A_{i*} \cdot B_{*j}$. Now we find the gradient of the loss $L(\mathbf{A}, \mathbf{B})$ by first finding the derivatives of L with respect to elements of $\mathbf{A}$ (a total of mr derivatives), then finding the derivatives of L with respect to elements of $\mathbf{B}$ (a total of nr derivatives).

First, consider an arbitrary element $A_{i'k'}$:

$$\begin{aligned}
\frac{\partial}{\partial A_{i'k'}} L(\mathbf{A}, \mathbf{B}) &= \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} 2(M_{ij} - (AB)_{ij}) \frac{\partial}{\partial A_{i'k'}} (-(AB)_{ij}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} 2(M_{i'j} - (AB)_{i'j}) \frac{\partial}{\partial A_{i'k'}} (-(AB)_{i'j}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} 2(M_{i'j} - (AB)_{i'j}) \cdot (-B_{k'j}) \\
&= \frac{1}{|\Omega|} \sum_{j:\,(i',j)\in\Omega} -2B_{k'j}(M_{i'j} - (AB)_{i'j}) \qquad (9.6)
\end{aligned}$$

Note that the second step is derived because $(AB)_{ij} = \sum_k A_{ik}B_{kj}$ and if $i \neq i'$, then $\frac{\partial (AB)_{ij}}{\partial A_{i'k'}} = 0$. Enumerating $(i,j) \in \Omega$ can be changed to only enumerating $(i',j) \in \Omega$. Similarly, we can consider an arbitrary element $B_{k'j'}$:

$$\begin{aligned}
\frac{\partial}{\partial B_{k'j'}} L(\mathbf{A}, \mathbf{B}) &= \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} 2(M_{ij} - (AB)_{ij}) \frac{\partial}{\partial B_{k'j'}} (-(AB)_{ij}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} 2(M_{ij'} - (AB)_{ij'}) \frac{\partial}{\partial B_{k'j'}} (-(AB)_{ij'}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} 2(M_{ij'} - (AB)_{ij'}) \cdot (-A_{ik'}) \\
&= \frac{1}{|\Omega|} \sum_{i:\,(i,j')\in\Omega} -2A_{ik'}(M_{ij'} - (AB)_{ij'}) \qquad (9.7)
\end{aligned}$$

Whew! That's a lot of derivatives, but we now have $\nabla L(\mathbf{A}, \mathbf{B})$ at our disposal.

9.3.2 Stochastic Gradient Descent for Matrix Factorization

Of course, we could use $\nabla L(\mathbf{A}, \mathbf{B})$ for a plain gradient descent as shown in Chapter 3. However, given that each derivative in the gradient involves a sum over a large number of indices, it would be

worthwhile to use stochastic gradient descent in order to estimate the overall gradient via a small random sample (as shown in Section 3.2).

If we take a sample $S \subset \Omega$ of the known entries at each iteration, the loss becomes

$$\widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{i,j \in S} (M_{ij} - (AB)_{ij})^2 \tag{9.8}$$

and the gradient becomes

$$\frac{\partial}{\partial A_{i'k'}} \widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{j:\ (i',j) \in S} -2B_{k'j}(M_{i'j} - (AB)_{i'j}) \tag{9.9}$$

$$\frac{\partial}{\partial B_{k'j'}} \widehat{L}(\mathbf{A}, \mathbf{B}) = \frac{1}{|S|} \sum_{i:\ (i,j') \in S} -2A_{ik'}(M_{ij'} - (AB)_{ij'}) \tag{9.10}$$

However, if we take a uniform sample S of Ω, the computation will not become much cheaper, since $(i, j) \in S$ can spread into many different rows and columns. One clever (and common) way to do so is to select a set of columns C by sampling k out of the overall n columns. This method is called *column sampling*. We then only need to consider entries $(i, j) \in \Omega$ where $j \in C$ and compute gradients only for the entries $B_{k,j}$ where $j \in C$. We can also perform row sampling in a very similar manner. In practice, whether we should use column sampling or row sampling, or gradient descent of full loss, depends on the actual sizes of m and n.