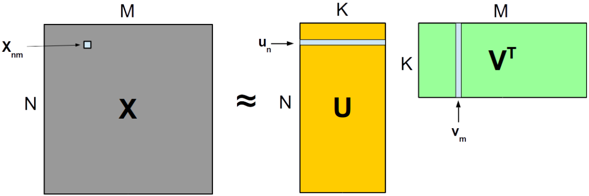
Matrix Factorization

- Given a matrix $\mathbf{X}$ of size $N \times M$, approximate it as a product of two matrices

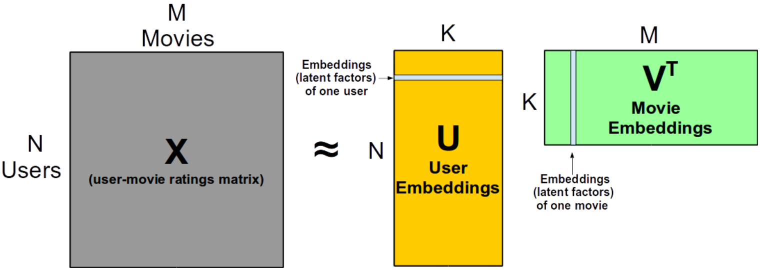
$$\mathbf{X} \approx \mathbf{U}\mathbf{V}^\top$$



- $\mathbf{U}$: $N \times K$ latent factor matrix
  - Each row of $\mathbf{U}$ represents a K-dim latent factor $\boldsymbol{u}_n$
- $\mathbf{V}$: $M \times K$ latent factor matrix
  - Each row of $\mathbf{V}$ represents a K-dim latent factor $\boldsymbol{v}_n$
- Each entry of $\mathbf{X}$ can be written as: $X_{nm} \approx \boldsymbol{u}_n^\top \boldsymbol{v}_m = \sum_{k=1}^K u_{nk} v_{mk}$
- If X_{nm} is large (small) then $\boldsymbol{u}_n$ and $\boldsymbol{v}_m$ should be similar (dissimilar)

Why Matrix Factorization?

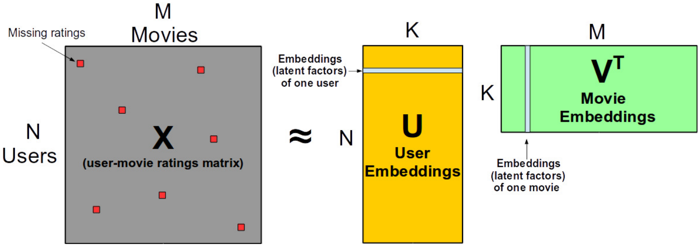
- The latent factors can be used/interpreted as "embeddings" or "features"



- Especially useful for learning good features for "dyadic" or relational data
  - Examples: Users-Movies ratings, Users-Products purchases, etc.
- If $K \ll \min\{M, N\} \Rightarrow$ then can also be seen as dimensionality reduction or a "low-rank factorization" of the matrix $\mathbf{X}$

Why Matrix Factorization?

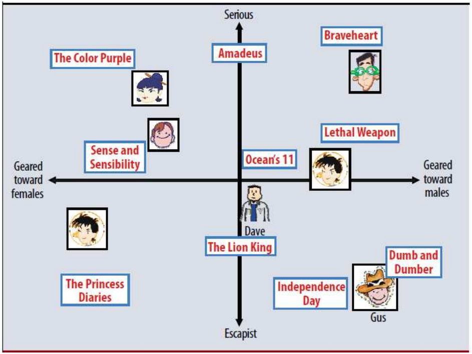
- Can also predict the missing/unknown entries in the original matrix



- Note: The latent factor matrices **U** and **V** can be learned even when the matrix **X** is only partially observed (as we will see shortly)
- After learning **U** and **V**, any missing X_{nm} can be approximated by $\boldsymbol{u}_n^\top \boldsymbol{v}_m$
- $\mathbf{U}\mathbf{V}^\top$ is the best low-rank matrix that approximates the full **X**
- Note: The "Netflix Challenge" was won by a matrix factorization method

Interpreting the Embeddings/Latent Factors

- Embeddings/latent factors can often be interpreted. E.g., as "genres" if $\mathbf{X}$ represents a user-movie rating matrix. A cartoon with $K = 2$ shown below

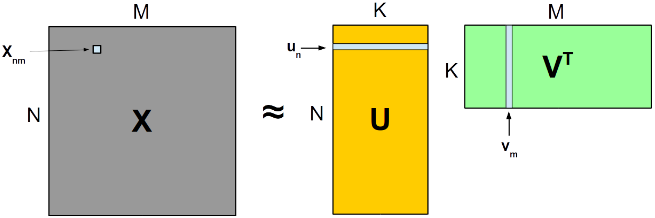


- Similar things (users/movies) get embedded nearby in the embedding space (two things will be deemed similar if their embeddings are similar). Thus useful for computing similarities and/or making recommendations

Picture courtesy: Matrix Factorization Techniques for Recommender Systems: Koren *et al*, 2009

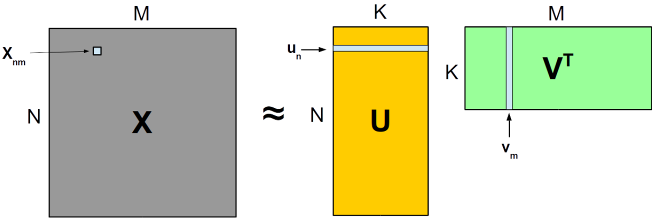
Matrix Factorization

- Recall our matrix factorization model: $\mathbf{X} \approx \mathbf{U}\mathbf{V}^\top$
- Goal: learn $\mathbf{U}$ and $\mathbf{V}$, given a subset Ω of entries in $\mathbf{X}$ (let's call it $\mathbf{X}_\Omega$)
- Some notations:
  - $\Omega = \{(n, m)\}$: X_{nm} is observed
  - Ω_{r_n}: column indices of observed entries in row n of $\mathbf{X}$
  - Ω_{c_m}: row indices of observed entries in column m of $\mathbf{X}$



Matrix Factorization

- We want $\mathbf{X}$ to be as close to $\mathbf{UV}^\top$ as possible



- Let's define a squared "loss function" over the observed entries in $\mathbf{X}$

$$\mathcal{L} = \sum_{(n,m)\in\Omega} (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)^2$$

- Here the latent factors $\{\boldsymbol{u}_n\}_{n=1}^N$ and $\{\boldsymbol{v}_m\}_{m=1}^M$ are the unknown parameters
- Squared loss chosen only for simplicity; other loss functions can be used
- How do we learn $\{\boldsymbol{u}_n\}_{n=1}^N$ and $\{\boldsymbol{v}_m\}_{m=1}^M$?

Alternating Optimization

- We will use an ℓ_2 regularized version of the squared loss function

$$\mathcal{L} = \sum_{(n,m)\in\Omega} (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)^2 + \sum_{n=1}^{N} \lambda_U ||\boldsymbol{u}_n||^2 + \sum_{m=1}^{M} \lambda_V ||\boldsymbol{v}_m||^2$$

- A **non-convex** problem. Difficult to optimize w.r.t. $\boldsymbol{u}_n$ and $\boldsymbol{v}_m$ jointly.
- One way is to solve for $\boldsymbol{u}_n$ and $\boldsymbol{v}_m$ in an alternating fashion, e.g.,
  - $\forall n$, fix all variables except $\boldsymbol{u}_n$ and solve the optim. problem w.r.t. $\boldsymbol{u}_n$

$$\arg\min_{\boldsymbol{u}_n} \sum_{m\in\Omega_{r_n}} (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)^2 + \lambda_U ||\boldsymbol{u}_n||^2$$

  - $\forall m$, fix all variables except $\boldsymbol{v}_m$ and solve the optim. problem w.r.t. $\boldsymbol{v}_m$

$$\arg\min_{\boldsymbol{v}_m} \sum_{n\in\Omega_{c_m}} (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)^2 + \lambda_V ||\boldsymbol{v}_m||^2$$

  - Iterate until not converged
- Each of these subproblems has a simple, convex objective function
- Convergence properties of such methods have been studied extensively

The Solutions

- Easy to show that the problem

$$\arg\min_{\boldsymbol{u}_n} \sum_{m \in \Omega_{r_n}} (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)^2 + \lambda_U ||\boldsymbol{u}_n||^2$$

.. has the solution

$$\boldsymbol{u}_n = \left(\sum_{m \in \Omega_{r_n}} \boldsymbol{v}_m \boldsymbol{v}_m^\top + \lambda_U \mathbf{I}_K \right)^{-1} \left(\sum_{m \in \Omega_{r_n}} X_{nm} \boldsymbol{v}_m \right)$$

- Likewise, the problem

$$\arg\min_{\boldsymbol{v}_m} \sum_{n \in \Omega_{c_m}} (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)^2 + \lambda_V ||\boldsymbol{v}_m||^2$$

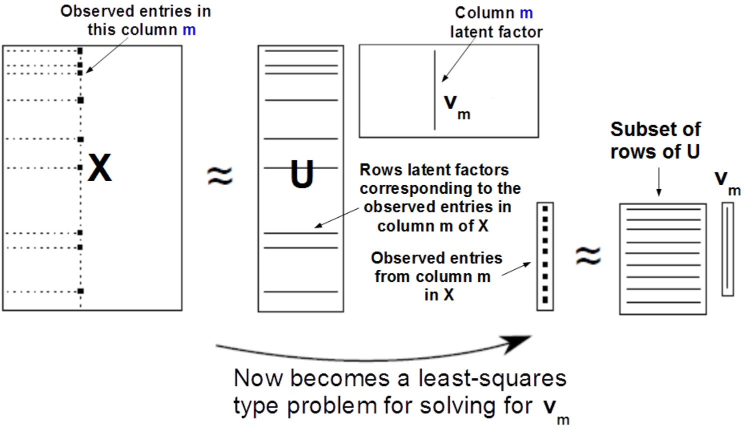
.. has the solution

$$\boldsymbol{v}_m = \left(\sum_{n \in \Omega_{c_m}} \boldsymbol{u}_n \boldsymbol{u}_n^\top + \lambda_V \mathbf{I}_K \right)^{-1} \left(\sum_{n \in \Omega_{c_m}} X_{nm} \boldsymbol{u}_n \right)$$

- Note that this is very similar to (regularized) least squares regression
- Thus matrix factorization can be also seen as a sequence of regression problems (one for each latent factor)

Matrix Factorization as Regression

Suppose we are solving for $\boldsymbol{v}_m$ (with $\mathbf{U}$ and all other $\boldsymbol{v}_m$'s fixed)



Can think of solving for $\boldsymbol{u}_n$ (with $\mathbf{V}$ and all other $\boldsymbol{u}_n$'s fixed) in the same way

Matrix Factorization as Regression

- A very useful way to understand matrix factorization
- Can modify the regularized least-squares like objective

$$\arg\min_{\boldsymbol{u}_n} \sum_{m \in \Omega_{r_n}} (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)^2 + \lambda_U \boldsymbol{u}_n^\top \boldsymbol{u}_n$$

.. using other loss functions and regularizers

- Some possible modifications:
    - If entries in the matrix $\mathbf{X}$ are binary, counts, etc. then we can replace the squared loss function by some other loss function (e.g., logistic or Poisson)
    - Can impose other constraints on the latent factors, e.g., non-negativity, sparsity, etc. (by changing the regularizer)
    - Can think of this also as a probabilistic model (a likelihood function on X_{nm} and priors on the latent factors $\boldsymbol{u}_n$, $\boldsymbol{v}_m$) and do MLE/MAP

Matrix Factorization: The Complete Algorithm

- Input: Partially complete matrix $\mathbf{X}_\Omega$
- Initialize the latent factors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_M$ randomly
- Iterate until converge
  - Update each row latent factor $\boldsymbol{u}_n$, $n = 1, \ldots, N$ (can be in parallel)

$$\boldsymbol{u}_n = \left(\sum_{m \in \Omega_{r_n}} \boldsymbol{v}_m \boldsymbol{v}_m^\top + \lambda_U \mathbf{I}_K \right)^{-1} \left(\sum_{m \in \Omega_{r_n}} X_{nm} \boldsymbol{v}_m \right)$$

  - Update each column latent factor $\boldsymbol{v}_m$, $m = 1, \ldots, M$ (can be in parallel)

$$\boldsymbol{v}_m = \left(\sum_{n \in \Omega_{c_m}} \boldsymbol{u}_n \boldsymbol{u}_n^\top + \lambda_V \mathbf{I}_K \right)^{-1} \left(\sum_{n \in \Omega_{c_m}} X_{nm} \boldsymbol{u}_n \right)$$

- Final prediction for any entry: $X_{nm} = \boldsymbol{u}_n^\top \boldsymbol{v}_m$

A Faster Algorithm via SGD

- Alternating optimization is nice but can be slow (note that it requires matrix inversion with cost $O(K^3)$ for updating each latent factor $\boldsymbol{u}_n, \boldsymbol{v}_m$)
- An alternative is to use stochastic gradient descent (SGD). In each round, select a randomly chosen entry X_{nm} with $(n, m) \in \Omega$
- Consider updating $\boldsymbol{u}_n$. For loss function $\sum_{m \in \Omega_{r_n}} (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)^2 + \lambda_U ||\boldsymbol{u}_n||^2$, the stochastic gradient w.r.t. $\boldsymbol{u}_n$ using this randomly chosen entry X_{nm} is

$$-(X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)\boldsymbol{v}_m + \lambda_U \boldsymbol{u}_n$$

- Thus the SGD update for $\boldsymbol{u}_n$ will be

$$\boldsymbol{u}_n = \boldsymbol{u}_n - \eta(\lambda_U \boldsymbol{u}_n - (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)\boldsymbol{v}_m)$$

- Likewise, the SGD update for $\boldsymbol{v}_m$ will be

$$\boldsymbol{v}_m = \boldsymbol{v}_m - \eta(\lambda_V \boldsymbol{v}_m - (X_{nm} - \boldsymbol{u}_n^\top \boldsymbol{v}_m)\boldsymbol{u}_n)$$

- The SGD algorithm chooses a random entry X_{nm} in each iteration, updates $\boldsymbol{u}_n, \boldsymbol{v}_m$, and repeats until convergece ($\boldsymbol{u}_n$'s,$\boldsymbol{v}_m$'s randomly initialized).