DIPARTIMENTO DI ELETTRONICA E INFORMAZIONE

POLITECNICO
DI MILANO





Coding Matrix Factorization

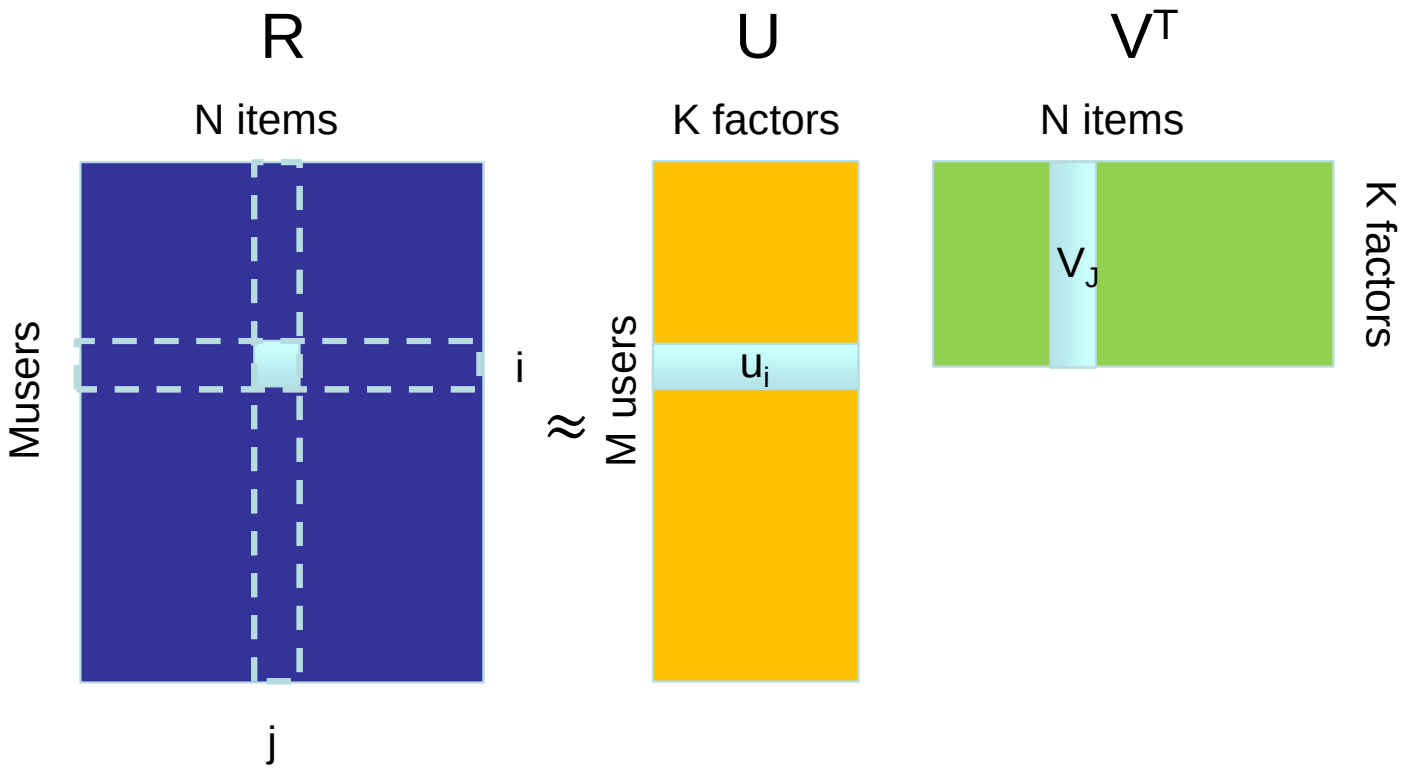
Massimo Quadrana

Maurizio Ferrari Dacrema

Outline

- FunkSVD (rating prediction)
- AsymmetricSVD (rating prediction)
- IALS or WRMF (implicit feedback)

Matrix Factorization in RecSys



$R \in \mathbb{R}_{MXN}$: rating matrix
$U \in \mathbb{R}_{MXK}$: user factors
$V \in \mathbb{R}_{NXK}$: item factors

M: number of users
N: number of items
K: number of latent factors


CAUTION: MATH

FunkSVD

From Simon Funk's post:

http://sifter.org/~simon/journal/20061211.html

Matrix Factorization in RecSys

- Optimization

$$\underset{\theta}{\operatorname{argmin}}\, J(\theta) = \underset{U,V}{\operatorname{argmin}}\, \frac{1}{2}||R - UV^T||_2^2 + \frac{\lambda}{2}(||U||_F^2 + ||V||_F^2)$$

Matrix Factorization in RecSys

- Optimization

$$\underset{\theta}{\operatorname{argmin}}\, J(\theta) = \underset{U,V}{\operatorname{argmin}} \frac{1}{2}||R - UV^T||_2^2 + \frac{\lambda}{2}(||U||_F^2 + ||V||_F^2)$$

Least-squares
(goodness of fit)

Regularization
(prevent overfitting)

Matrix Factorization in RecSys

- Optimization

$$\underset{\theta}{\operatorname{argmin}} J(\theta) = \underset{U,V}{\operatorname{argmin}} \frac{1}{2}||R - UV^T||_2^2 + \frac{\lambda}{2}(||U||_F^2 + ||V||_F^2)$$

Least-squares (goodness of fit)

Regularization (prevent overfitting)

$$= \underset{u*,v*}{\operatorname{argmin}} \frac{1}{2}\sum_{i,j}(r_{ij} - u_i^T v_j)^2 + \frac{\lambda}{2}(\sum_i ||u_i||^2 + \sum_j ||v_j||^2)$$

WARNING: For simplicity I won't include user and item biases in the models, but you can try with them at home

Gradient Descent Variants

- Batch Gradient Descent
  - Calculate the gradients for the **whole** dataset to perform just **one** update
  - η learning rate (or step size): controls speed of convergence (or divergence if too large)
  - Cons: Very slow, no online updates

$$\theta = \theta - \eta \nabla_\theta J(\theta)$$

- Stochastic Gradient Descent
  - Perform a parameter update for **each** user-item tuple (i,j)
  - Pros: cheap updates, online update, same practical convergence of Batch-GD
  - Cons: high variance (fluctuations), local minima

$$\theta = \theta - \eta \nabla_\theta J(\theta; (i, j))$$

- Minibatch SGD (not-covered)
  - Update every minibatch of **n** training tuples
  - Pros over SGD: less variance → stable convergence, fast matrix-ops

Gradient Descent Variants

- Batch Gradient Descent
  - Calculate the gradients for the **whole** dataset to perform just **one** update
  - learning rate (or step size): controls speed of convergence (or divergence if too large)
  - Cons: Very slow, no online updates

- Stochastic Gradient Descent

$$\theta = \theta - \eta \nabla_\theta J(\theta; (i, j))$$

  - Perform a parameter update for **each** user-item tuple (i,j)
  - Pros: cheap updates, online update, same practical convergence of Batch-GD
  - Cons: high variance (fluctuations), local minima

- Minibatch SGD (not-covered)
  - Update every minibatch of **n** training tuples
  - Pros over SGD: less variance → stable convergence, fast matrix-ops

Matrix Factorization with SGD

- Compute the partial derivatives

$$\frac{\partial L(\theta)}{\partial u_i} = -(r_{ij} - u_i^T v_j)v_j^T + \lambda u_i \qquad \frac{\partial L(\theta)}{\partial v_j} = -(r_{ij} - u_i^T v_j)u_i + \lambda v_j$$

Matrix Factorization with SGD

- Compute the partial derivatives

$$\frac{\partial L(\theta)}{\partial u_i} = -(r_{ij} - u_i^T v_j)v_j^T + \lambda u_i \qquad \frac{\partial L(\theta)}{\partial v_j} = -(r_{ij} - u_i^T v_j)u_i + \lambda v_j$$

- Random shuffle **non-zero** user-item pairs in R
  - For each sampled pair (i,j)

$$e_{ij} \stackrel{def}{=} r_{ij} - u_i^T v_j$$

$$u_i = u_i + \eta(e_{ij} v_j - \lambda u_i)$$

$$v_j = v_j + \eta(e_{ij} u_i - \lambda v_j)$$

Matrix Factorization with SGD

- Compute the partial derivatives

$$\frac{\partial L(\theta)}{\partial u_i} = -(r_{ij} - u_i^T v_j)v_j^T + \lambda u_i \qquad \frac{\partial L(\theta)}{\partial v_j} = -(r_{ij} - u_i^T v_j)u_i + \lambda v_j$$

- Random shuffle **non-zero** user-item pairs in R
  - For each sampled pair (i,j)

$$e_{ij} \stackrel{def}{=} r_{ij} - u_i^T v_j$$

$$u_i = u_i + \eta(e_{ij} v_j - \lambda u_i)$$

$$v_j = v_j + \eta(e_{ij} u_i - \lambda v_j)$$

- Complexity Time: O(nnz(R)) << O(MN) – Space: O((N+M)K)

MF-SGD tips and tricks

- Tune the learning rate
  - Grid search with cross-validation
  - Reduce the learning rate over time with time decay
  - More sophisticated method (Adagrad, RmsProp,…)

- Regularization factors play a big role
  - Tune them as well

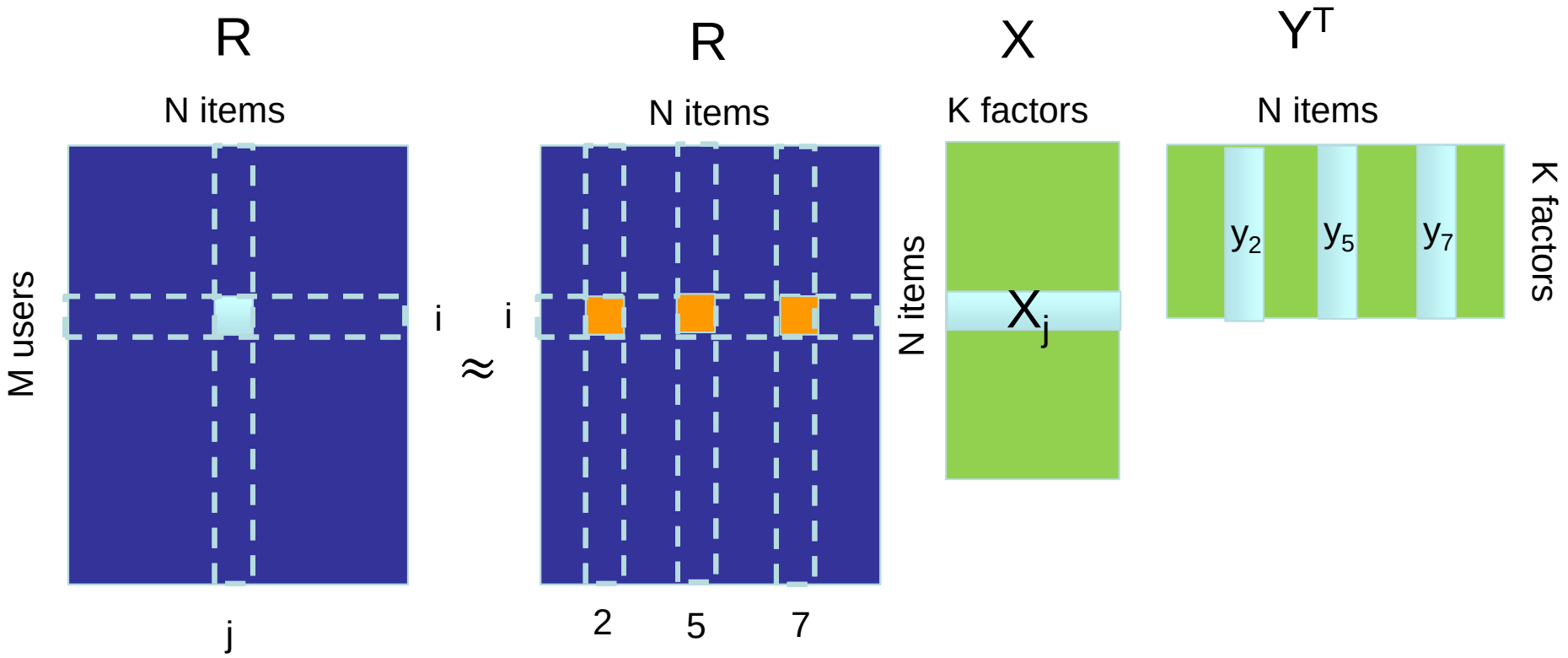
AsymmetricSVD

Modified version of SVD++

SVD++ is FunkSVD with global effects

AsymmetricSVD

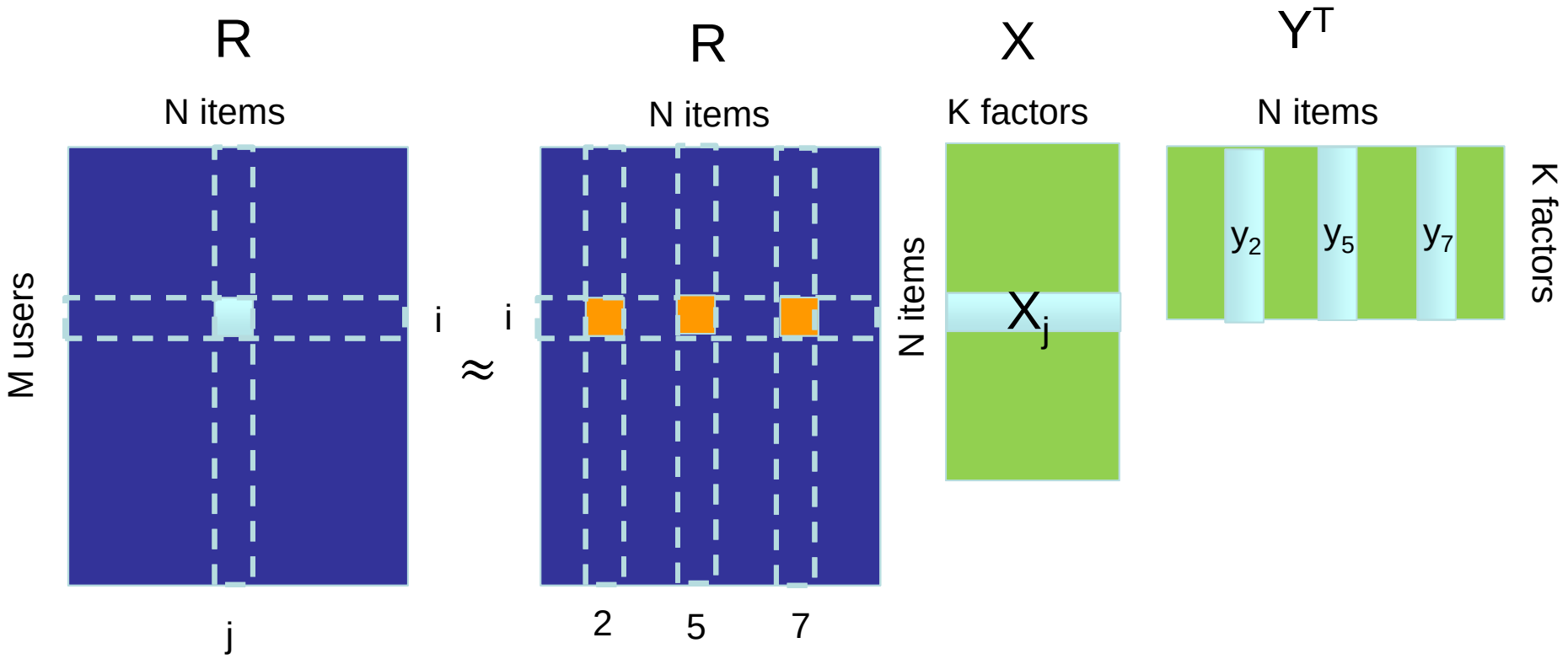
- Basic idea: represent users latent factors as the weighted average of their rated item latent factors



AsymmetricSVD

X and Y have the same shape but different meaning:

- Y - how much a feature is important for an item
- X - how much a feature is important for a user



AsymmetricSVD

- Prediction rule (again, biases are omitted):

$$\widehat{r_{ij}} = \frac{1}{\sqrt{|R(i)|}} \sum_{l \in R(i)} r_{il} x_j^T y_l = \frac{1}{\sqrt{|R(i)|}} x_j^T \boxed{\sum_{l \in R(i)} r_{il} y_l}$$

Rated items

  - R(i): rated by user i

AsymmetricSVD

- Prediction rule (again, biases are omitted):

$$\widehat{r_{ij}} = \frac{1}{\sqrt{|R(i)|}} \sum_{l \in R(i)} r_{il} x_j^T y_l = \frac{1}{\sqrt{|R(i)|}} x_j^T \sum_{l \in R(i)} r_{il} y_l$$

  - R(i): rated by user i

- Loss function

$$\underset{X,Y}{\operatorname{argmin}} J(\theta) = \underset{x*,y*}{\operatorname{argmin}} \frac{1}{2} \sum_{i,j \in R} \left(r_{ij} - \frac{1}{\sqrt{|R(i)|}} x_j^T \sum_{l \in R(i)} r_{il} y_l\right)^2 + \frac{\lambda}{2} \left(\sum_i ||x_i||^2 + \sum_j ||y_j||^2\right)$$

AsymmetricSVD

- Pros:
  - Fewer parameters, usually N (items) << M (users)
  - Handle new users as soon as they start providing ratings
- Cons:
  - Similar function but much slower training than simple FunkSVD, because of the summations

AsySVD-SGD updates

- Partial derivatives

$$\frac{\partial L(\theta)}{\partial x_j} = -e_{ij}\left(\sum_{l \in R(i)} r_{il} y_l\right) + \lambda x_j \qquad \frac{\partial L(\theta)}{\partial y_l} = -e_{ij} x_j + \lambda y_l$$

- Update x_j and y_l similar to FunkSVD

IALS or WRMF

Matrix Factorization for implicit feedbacks

IALS or WRMF

- Implicit Alternating Least Squares
  - Collaborative Filtering for Implicit Feedback Datasets (Hu et al, 2008)
- Also called Weighted Regularized Matrix Factorization (WRMF)

- Implicit feedback
  - Numerical values in explicit feedback express **preference** (1-totally dislike, 5-really like)
  - Numerical values in implicit feedback express **confidence** (e.g., how much time have I watched a movie?)
  - No negative feedback: missing as ‘not-interacted’

- Basic idea
  - Split the numerical score r_{ij} into
    - preference score p_{ij} (p_{ij} = 1 iif r_{ij} > 0 else 0)
    - confidence score c_{ij}, for example

1. Linear scaling $c_{ij} = 1 + \alpha r_{ij}$
2. Log scaling $c_{ij} = 1 + \alpha \log(1 + r_{ij}/\epsilon)$

The higher number of interactions, the higher the confidence

- Basic idea
  - Matrix factorization over preference and confidence scores

Elementwise (Hadamard) product

$$\underset{\theta}{\operatorname{argmin}} J(\theta) = \underset{X,Y}{\operatorname{argmin}} \frac{1}{2}||C \odot (P - XY^T)||_F^2 + \frac{\lambda}{2}(||X||^2 + ||Y||^2)$$

$$= \underset{x*,y*}{\operatorname{argmin}} \frac{1}{2} \sum_{i,j} c_{ij}(p_{ij} - x_i^T y_j) + \lambda(\sum_i ||x_i||^2 + \sum_j ||y_j||^2)$$

  - Trained with Alternating Least Squares

1. Fixed Y, optimize for X
2. Fixed X, optimize for Y

Learning IALS

- Alternating Least Squares
  1. Fixed Y, optimize for X

$$L(x_i) = \frac{1}{2}\sum_j c_{ij}(r_{ij} - x_i^T y_j)^2 + \frac{\lambda}{2}||x_i||^2$$

Equivalent to

$$L(x_i) = \frac{1}{2}||C^i(p(i) - Yx_i)||_F^2 + \frac{\lambda}{2}||x_i||^2$$

where C^i NxN diagonal matrix with $C^i_{jj} = c_{ij}$
and $p(i)$ is the vector of preferences of i

Learning IALS

- Alternating Least Squares
  1. Fixed Y, optimize for X

Partial derivative

$$\frac{\partial L(x_i)}{\partial x_i} = -Y^T C^i (p(i) - Y x_i) + \lambda x_i$$
$$= -Y^T C^i p(i) + (Y^T C^i Y + \lambda I) x_i = 0$$

Update Rule

$$x_i = (Y^T C^i Y + \lambda I)^{-1} Y^T C^i p(i)$$

Learning IALS

- Update rule - a trick

$$Y^T C^i Y = \boxed{Y^T Y} + \boxed{Y^T (C^i - I) Y}$$

Independent from i
→ Precompute

Depends only on the **non-zeros** in p(i)!!!
(if one of the previous scaling functions is used)

- Back to ALS
  1. Fixed X, optimize for Y
- Implementation
  1. Slow iterative code with numpy (uses _lsq_solver)
  2. Fast numpy only (uses _lsq_solver_fast)
  3. Faster parallelized version at this link: https://github.com/benfred/implicit