Matrix Factorization and Factorization Machines for Recommender Systems

Chih-Jen Lin
Department of Computer Science
National Taiwan University

Talk at SDM workshop on Machine Learning Methods on Recommender Systems, May 2, 2015
Outline

1. Matrix factorization
2. Factorization machines
3. Conclusions
In this talk I will briefly discuss two related topics

- Fast matrix factorization (MF) in shared-memory systems
- Factorization machines (FM) for recommender systems and classification/regression

Note that MF is a special case of FM
Matrix factorization

1. Introduction and issues for parallelization
2. Our approach in the package LIBMF

Factorization machines

Conclusions
Outline

1. Matrix factorization
   - Introduction and issues for parallelization
   - Our approach in the package LIBMF

2. Factorization machines

3. Conclusions
Matrix Factorization is an effective method for recommender systems (e.g., Netflix Prize and KDD Cup 2011)

But training is slow.

We developed a parallel MF package LIBMF for shared-memory systems

http://www.csie.ntu.edu.tw/~cjlin/libmf

Best paper award at ACM RecSys 2013
Matrix Factorization (Cont’d)

- For recommender systems: a group of users give ratings to some items

<table>
<thead>
<tr>
<th>User</th>
<th>Item</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>80</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

The information can be represented by a rating matrix $R$
Matrix Factorization (Cont’d)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>v</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>?2,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>r_{u,v}</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $m, n$: numbers of users and items
- $u, v$: index for $u_{th}$ user and $v_{th}$ item
- $r_{u,v}$: $u_{th}$ user gives a rating $r_{u,v}$ to $v_{th}$ item
Matrix Factorization (Cont’d)

\[ R \approx P^T Q \]

\[ m \times n \approx m \times k \times k \times n \]

- \( k \) : number of latent dimensions
- \( r_{u,v} = p_u^T q_v \)
- \( r_{2,2} = p_2^T q_2 \)
Matrix Factorization (Cont’d)

- A non-convex optimization problem:

\[
\min_{P,Q} \sum_{(u,v) \in R} \left( (r_{u,v} - p_u^T q_v)^2 + \lambda_P \|p_u\|_F^2 + \lambda_Q \|q_v\|_F^2 \right)
\]

\(\lambda_P\) and \(\lambda_Q\) are regularization parameters

- SG (Stochastic Gradient) is now a popular optimization method for MF

- It loops over ratings in the training set.
Matrix Factorization (Cont’d)

- **SG update rule:**

\[
\begin{align*}
\mathbf{p}_u &\leftarrow \mathbf{p}_u + \gamma (\mathbf{e}_{u,v}\mathbf{q}_v - \lambda_P \mathbf{p}_u), \\
\mathbf{q}_v &\leftarrow \mathbf{q}_v + \gamma (\mathbf{e}_{u,v}\mathbf{p}_u - \lambda_Q \mathbf{q}_v)
\end{align*}
\]

where

\[
\mathbf{e}_{u,v} \equiv r_{u,v} - \mathbf{p}_u^T \mathbf{q}_v
\]

- **SG is inherently sequential**
SG for Parallel MF

After $r_{3,3}$ is selected, ratings in gray blocks cannot be updated.

But $r_{6,6}$ can be used.

\[ r_{3,1} = p_3^T q_1 \]
\[ r_{3,2} = p_3^T q_2 \]
\[ \ldots \]
\[ r_{3,6} = p_3^T q_6 \]
\[ r_{6,6} = p_6^T q_6 \]
SG for Parallel MF (Cont’d)

We can split the matrix to blocks. Then use threads to update the blocks where ratings in different blocks don’t share $p$ or $q$. 

![Matrix Factorization Diagram]

- Blocks 1 and 2 share $p$.
- Blocks 3 and 4 share $q$.
- Blocks 5 and 6 do not share $p$ or $q$.
SG for Parallel MF (Cont’d)

- This concept of splitting data to independent blocks seems to work.
- However, there are many issues to have a right implementation under the given architecture.
Outline

1. Matrix factorization
   - Introduction and issues for parallelization
   - Our approach in the package LIBMF

2. Factorization machines

3. Conclusions
Our approach in the package LIBMF

- Parallelization (Zhuang et al., 2013; Chin et al., 2015a)
  - Effective block splitting to avoid synchronization time
  - Partial random method for the order of SG updates
- Adaptive learning rate for SG updates (Chin et al., 2015b)
  Details omitted due to time constraint
Block Splitting and Synchronization

- A naive way for $T$ nodes is to split the matrix to $T \times T$ blocks

- This is used in DSGD (Gemulla et al., 2011) for distributed systems. The setting is reasonable because communication cost is the main concern

- In distributed systems, it is difficult to move data or model
Block Splitting and Synchronization (Cont’d)

• However, for shared memory systems, synchronization is a concern.

<table>
<thead>
<tr>
<th>Thread</th>
<th>0→10</th>
<th>10→20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Busy</td>
<td>Busy</td>
</tr>
<tr>
<td>2</td>
<td>Busy</td>
<td>Idle</td>
</tr>
<tr>
<td>3</td>
<td>Busy</td>
<td>Busy</td>
</tr>
</tbody>
</table>

10s wasted!!
Lock-Free Scheduling

We split the matrix to enough blocks. For example, with two threads, we split the matrix to $4 \times 4$ blocks.

\[
\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]

0 is the updated counter recording the number of updated times for each block.
Lock-Free Scheduling (Cont’d)

Firstly, $T_1$ selects a block randomly

\[
\begin{array}{cccccc}
T_1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Lock-Free Scheduling (Cont’d)

For $T_2$, it selects a block neither green nor gray randomly.

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$T_2$ 0</td>
</tr>
</tbody>
</table>
Lock-Free Scheduling (Cont’d)

After $T_1$ finishes, the counter for the corresponding block is added by one.
**Lock-Free Scheduling (Cont’d)**

$T_1$ can select available blocks to update.

**Rule:** select one that is least updated.

\[
\begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & T_2 \end{array}
\]
Matrix factorization

Our approach in the package LIBMF

Lock-Free Scheduling (Cont’d)

**SG**: applying Lock-Free Scheduling

**SG**: applying DSGD-like Scheduling

![Graph showing RMSE vs Time for MovieLens 10M and Yahoo!Music](image)

- **MovieLens 10M**: 18.71s → 9.72s (RMSE: 0.835)
- **Yahoo!Music**: 728.23s → 462.55s (RMSE: 21.985)
Memory Discontinuity

Discontinuous memory access can dramatically increase the training time. For SG, two possible update orders are:

<table>
<thead>
<tr>
<th>Update order</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>Faster and stable</td>
<td>Memory discontinuity</td>
</tr>
<tr>
<td>Sequential</td>
<td>Memory continuity</td>
<td>Not stable</td>
</tr>
</tbody>
</table>

Our lock-free scheduling gives *randomness*, but the resulting code may not be cache friendly.
Partial Random Method

Our solution is that for each block, access both $\hat{R}$ and $\hat{P}$ continuously.

$$\hat{R} : \text{(one block)}$$

Partial: sequential in each block
Random: random when selecting block
Matrix factorization

Our approach in the package LIBMF

Partial Random Method (Cont’d)

MovieLens 10M

Yahoo!Music

The performance of Partial Random Method is better than that of Random Method
Experiments

State-of-the-art methods compared

- **LIBPMF**: a parallel coordinate descent method (Yu et al., 2012)
- **NOMAD**: an asynchronous SG method (Yun et al., 2014)
- **LIBMF**: earlier version of LIBMF (Zhuang et al., 2013; Chin et al., 2015a)
- **LIBMF++**: with adaptive learning rates for SG (Chin et al., 2015c)
### Experiments (Cont’d)

<table>
<thead>
<tr>
<th>Data Set</th>
<th>$m$</th>
<th>$n$</th>
<th>#ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netflix</td>
<td>2,649,429</td>
<td>17,770</td>
<td>99,072,112</td>
</tr>
<tr>
<td>Yahoo!Music</td>
<td>1,000,990</td>
<td>624,961</td>
<td>252,800,275</td>
</tr>
<tr>
<td>Webscope-R1</td>
<td>1,948,883</td>
<td>1,101,750</td>
<td>104,215,016</td>
</tr>
<tr>
<td>Hugewiki</td>
<td>39,706</td>
<td>25,000,000</td>
<td>1,703,429,136</td>
</tr>
</tbody>
</table>

- Due to machine capacity, Hugewiki here is about half of the original
- $k = 100$
Our method has been extended to solve NMF

\[
\min_{P, Q} \sum_{(u,v) \in R} \left( (r_{u,v} - p_u^T q_v)^2 + \lambda_P \|p_u\|^2_F + \lambda_Q \|q_v\|^2_F \right)
\]

subject to \(P_{i,u} \geq 0, Q_{i,v} \geq 0, \forall i, u, v\)
Outline

1 Matrix factorization
2 Factorization machines
3 Conclusions
MF and Classification/Regression

- MF solves

\[
\min_{P, Q} \sum_{(u, v) \in R} (r_{u,v} - p_u^T q_v)^2
\]

Note that I omit the regularization term

- Ratings are the only given information
- This doesn't sound like a classification or regression problem
- In the second part of this talk we will make a connection and introduce FM (Factorization Machines)
Handling User/Item Features

- What if instead of user/item IDs we are given user and item features?
- Assume user $u$ and item $v$ have feature vectors $f_u$ and $g_v$
- How to use these features to build a model?
We can consider a regression problem where data
instances are

\[
\begin{align*}
\text{value features} \\
\vdots \\
\vdots \\
r_{uv} & \quad \begin{bmatrix} f_u^T & g_v^T \end{bmatrix} \\
\vdots & \quad \vdots \\
\end{align*}
\]

and solve

\[
\min_w \sum_{u,v \in R} \left( R_{u,v} - w^T \begin{bmatrix} f_u \\ g_v \end{bmatrix} \right)^2
\]
However, this does not take the interaction between users and items into account.

Note that we are approximating the rating $r_{u,v}$ of user $u$ and item $v$.

Let

\[ U \equiv \text{number of user features} \]
\[ V \equiv \text{number of item features} \]

Then

\[ f_u \in R^U, \ u = 1, \ldots, m, \]
\[ g_v \in R^V, \ v = 1, \ldots, n \]
Following the concept of degree-2 polynomial mappings in SVM, we can generate new features

\[(f_u)_t (g_v)_s, t = 1, \ldots, U, s = 1, \ldots, V\]

and solve

\[
\min_{t,s, \forall t,s} \sum_{u,v \in R} (r_{u,v} - \sum_{t'=1}^{U} \sum_{s'=1}^{V} w_{t',s'} (f_u)_t (g_v)_s)^2
\]
This is equivalent to

$$\min_W \sum_{u,v \in R} (r_{u,v} - f_u^T W g_v)^2,$$

where

$$W \in R^{U \times V}$$

is a matrix.

If we have \( \text{vec}(W) \) by concatenating \( W \)'s columns, another form is

$$\min_W \sum_{u,v \in R} \left( r_{u,v} - \text{vec}(W)^T \begin{bmatrix} \vdots \ (f_u)_t (g_v)_s \vdots \end{bmatrix} \right)^2,$$
However, this setting **fails for extremely sparse features**

Consider the most extreme situation. Assume we have

user ID and item ID as features

Then

\[ U = m, J = n, \]

\[ \mathbf{f}_i = [0, \ldots, 0, 1, 0, \ldots, 0]^T \]

\[ i - \underbrace{1}_{i-1} \]
The optimal solution is

\[ W_{u,v} = \begin{cases} 
  r_{u,v}, & \text{if } u, v \in R \\
  0, & \text{if } u, v \notin R 
\end{cases} \]

We can never predict

\[ r_{u,v}, u, v \notin R \]
The reason why we cannot predict unseen data is because in the optimization problem

$$\# \text{ variables} = mn \gg \# \text{ instances} = |R|$$

- Overfitting occurs
- Remedy: we can let

$$W \approx P^TQ,$$

where $P$ and $Q$ are low-rank matrices. This becomes matrix factorization
This can be generalized to sparse user and item features

$$\min_{u,v \in R} (R_{u,v} - f_u^T P^T Q g_v)^2$$

That is, we think

$$Pf_u \text{ and } Qg_v$$

are latent representations of user $u$ and item $v$, respectively

This becomes factorization machines (Rendle, 2010)
Factorization Machines (Cont’d)

- Similar ideas have been used in other places such as Stern, Herbrich, and Graepel (2009)
- In summary, we connect MF and classification/regression by the following settings
  - We need combination of different feature types (e.g., user, item, etc)
  - However, overfitting occurs if features are very sparse
  - We use product of low-rank matrices to avoid overfitting
We see that such ideas can be used for not only recommender systems.

They may be useful for any classification problems with very sparse features.
Field-aware Factorization Machines

- We have seen that FM is useful to handle highly sparse features such as user IDs.
- What if we have more than two ID fields?
- For example, in CTR prediction for computational advertising, we may have:

<table>
<thead>
<tr>
<th>value</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>CTR</td>
<td>user ID, Ad ID, site ID</td>
</tr>
</tbody>
</table>
Field-aware Factorization Machines (Cont’d)

- FM can be generalized to handle different interactions between fields
  
  Two latent matrices for user ID and Ad ID
  Two latent matrices for user ID and site ID

- This becomes FFM: field-aware factorization machines (Rendle and Schmidt-Thieme, 2010)
Factorization machines

FFM for CTR Prediction

- It’s used by Jahrer et al. (2012) to win the 2nd prize of KDD Cup 2012
- Recently my students used FFM to win two Kaggle competitions
- After we used FFM to win the first, in the second competition all top teams use FFM
- Note that for CTR prediction, logistic rather than squared loss is used
How to decide which field interactions to use?
If features are not extremely sparse, can the result still be better than degree-2 polynomial mappings?
Note that we lose the convexity here.
We have a software LIBFFM for public use
http://www.csie.ntu.edu.tw/~cjlin/libffm
Experiments

- We see that
  \[ W \Rightarrow P^T Q \]
  reduces the number of variables

- What if we map
  \[
  \begin{bmatrix}
  \vdots \\
  (f_u)_t (g_v)_s \\
  \vdots 
  \end{bmatrix} \Rightarrow \text{a shorter vector}
  \]
  to reduce the number of features/variables
However, we may have something like

\[(r_{1,2} - W_{1,2})^2 \Rightarrow (r_{1,2} - \bar{w}_1)^2\]  
\[(r_{1,4} - W_{1,4})^2 \Rightarrow (r_{1,4} - \bar{w}_2)^2\]  
\[(r_{2,1} - W_{2,1})^2 \Rightarrow (r_{2,1} - \bar{w}_3)^2\]  
\[(r_{2,3} - W_{2,3})^2 \Rightarrow (r_{2,3} - \bar{w}_1)^2\]  

Clearly, there is no reason why (1) and (2) should share the same variable \(\bar{w}_1\)

In contrast, in MF, we connect \(r_{1,2}\) and \(r_{1,3}\) through \(p_1\)
Experiments (Cont’d)

- A simple comparison on MovieLens
  - # training: 9,301,274, # test: 698,780, # users: 71,567, # items: 65,133
- Results of MF: RMSE = 0.836
- Results of Poly-2 + Hashing:
  - RMSE = 1.14568 (10^6 bins), 3.62299 (10^8 bins), 3.76699 (all pairs)
- We can clearly see that MF is much better
Outline

1. Matrix factorization
2. Factorization machines
3. Conclusions
Conclusions

- In this talk we have talked about MF and FFM
- MF is a mature technique, so we investigate its fast training
- FFM is relatively new. We introduce its basic concepts and practical use
The following students have contributed to works mentioned in this talk

- Wei-Sheng Chin
- Yu-Chin Juan
- Bo-Wen Yuan
- Yong Zhuang