Master IASD
Data Science Lab
Explicit collaborative filtering

lecun-team:
Linghao Zeng, Zhe Huang, Joseph Amigo

1 Introduction to collaborative filtering

Collaborative filtering (CF) is a method used to make automatic predictions about the preferences of a user by collecting preferences from many users. One of the prominent techniques for CF is *Matrix Factorization* (MF).

1.1 Problem Formulation

Given a user-item interaction matrix, $R \in \mathbb{R}^{m \times n}$, where m is the number of users and n is the number of items, the goal of matrix factorization is to approximate R as a product of two lower-rank matrices, $I \in \mathbb{R}^{m \times k}$ and $U \in \mathbb{R}^{k \times n}$:

$$R \approx I \times U = \hat{R}$$

The objective is to minimize the difference between R and its approximation $\hat{R}$:

$$\min_{I,U} \|R - IU\|_{\mathcal{F}}^2 + \lambda \|I\|_{\mathcal{F}}^2 + \mu \|U\|_{\mathcal{F}}^2$$

where λ and μ are regularization parameters.

1.2 Experimental Setup

1.2.1 Dataset

We were provided with a dataset capturing interactions between 610 users and 4,980 movies. Ideally, if every user rated every movie, it would result in 3,037,800 ratings. However, the sparsity is retained in both the training and testing datasets, each comprising only 31,598 ratings. There is no overlap observed between the datasets, preventing any data leakage.

Missing values in the dataset, represented as 'NaN', were replaced with 0. It's important to highlight that this 0 doesn't indicate a rating of zero but signifies a lack of rating.

1.2.2 Evaluation Metrics

For our matrix factorization model, we produce an output matrix $\hat{R}$ using the original ratings matrix R and the matrices I and U resulting from the factorization:

$$\hat{R}_{i,j} = \begin{cases} R_{i,j} & \text{if } R_{i,j} > 0 \\ (I \times U)_{i,j} & \text{otherwise} \end{cases}$$

With this, to assess the performance on the test set, we utilized the Root Mean Squared Error (RMSE):

$$\text{RMSE} = \sqrt{\frac{\sum_{i,j}(R_{\text{test},i,j} - \hat{R}_{i,j} \times \mathbf{1}((R_{\text{test},i,j} > 0))^2)}{\sum_{i,j} \mathbf{1}((R_{\text{test},i,j} > 0))}}$$

where R_{test} is the test dataset, and $\mathbf{1}(\cdot)$ is the indicator function.

2 Algorithms

To ensure optimization occurs only over existing ratings, instead of employing Eq.1.1, we incorporated an indicator function $\mathbf{1}(R > 0)$ to mask and consider only the positions where ratings exist. The modified loss function is defined as:

$$\mathcal{L}(I, U) = \|R - (I \times U) \odot \mathbf{1}(R > 0)\|_{\mathcal{F}}^2 + \lambda \|I\|_{\mathcal{F}}^2 + \mu \|U\|_{\mathcal{F}}^2$$

2.1 Gradient Descent (GD)

Initialization Strategy Inspired by the good performance of an average-based model, we initialized matrices I and U based on the average ratings from the matrix R:

For matrices $I \in \mathbb{R}^{m \times 1}$ and $U \in \mathbb{R}^{1 \times n}$:

$$I_{i,\cdot} = \sqrt{\frac{\sum_j R_{i,j}}{\sum_j \mathbf{1}(R_{i,j} > 0)}} \quad \text{and} \quad U_{\cdot,j} = \sqrt{\frac{\sum_i R_{i,j}}{\sum_i \mathbf{1}(R_{i,j} > 0)}}$$

This data-driven initialization aims to expedite the convergence of the gradient descent algorithm.

Optimization using GD Given our loss function, we employ the gradient descent algorithm for optimization. The steps of the algorithm are as follows:

Algorithm 1 Gradient Descent for Matrix Factorization

```
1: Initialize matrices $I$ and $U$.
2: Set learning rate $\alpha$ and regularization parameters $\lambda$ and $\mu$.
3: while not converged do
4:     Compute the loss: $\mathcal{L}(I, U)$.
5:     Compute gradient with respect to $U$ and $I$:
           $\nabla_U = 2\left((R - I \times U) \odot \mathbf{1}(R > 0)\right)^\top I + 2\mu U$
           $\nabla_I = 2\left((R - I \times U) \odot \mathbf{1}(R > 0)\right) U^\top + 2\lambda I$
6:     Update $U$ and $I$:
           $U \leftarrow U - \alpha \nabla_U$
           $I \leftarrow I - \alpha \nabla_I$
7:     Check for convergence.
```

2.2 Alternated Least Squares (ALS)

Here the same initialization strategy is used as in GD. The steps of ALS are as follows:

Algorithm 2 Alternated Least Squares for Matrix Factorization

```
1:  **Initialize** matrices $I$ and $U$.
2:  **Set** regularization parameters $\lambda$ and $\mu$.
3:  **while** not converged **do**
4:      Compute the loss: $\mathcal{L}(I, U)$.
5:      **for** each item $i$ **do**
6:          $mask \leftarrow R_{i,\cdot} > 0$
7:          Formulate the effective user-factor matrix $U_{mask}$
8:          Solve for $I_{i,\cdot}$: $I_{i,\cdot} = \left(U_{mask}U_{mask}^T + \lambda I\right)^{-1} U_{mask}R_{i,mask}$
9:      **for** each user $j$ **do**
10:         $mask \leftarrow R_{\cdot,j} > 0$
11:         Formulate the effective item-factor matrix $I_{mask}$
12:         Solve for $U_{\cdot,j}$: $U_{\cdot,j} = \left(I_{mask}^T I_{mask} + \mu I\right)^{-1} I_{mask}^T R_{mask,j}$
13:     Check for convergence.
```

2.3 Deep matrix factorization (DMF)[1]

In our deep matrix factorization approach, we employ two neural networks, each taking a user or movie vector (corresponding to a row or column of matrix R) as input and generating embeddings:

$$\mathrm{NN}_{\theta_1}^{\text{movie}} : \mathbb{R}^n \to \mathbb{R}^{256}, \quad \mathrm{NN}_{\theta_2}^{\text{user}} : \mathbb{R}^m \to \mathbb{R}^{256}$$

This enables us to compute cosine similarity to extract collaborative information from these embeddings:

$$\delta_{\theta_1,\theta_2}(\text{user}_i, \text{movie}_j) = \frac{\langle \mathrm{NN}_{\theta_1}^{\text{movie}}(\text{movie}_j), \mathrm{NN}_{\theta_2}^{\text{user}}(\text{user}_i)\rangle}{\|\mathrm{NN}_{\theta_1}^{\text{movie}}(\text{movie}_j)\| \|\mathrm{NN}_{\theta_2}^{\text{user}}(\text{user}_i)\|} \in [-1, 1]$$

After rescaling $\delta_{\theta_1,\theta_2}(\text{user}_i, \text{movie}_j)$ from $[-1, 1]$ to $[0, 1]$, and similarly for r_{ij} from $[0, 5]$ to $[0, 1]$:
The binary cross entropy guides the network towards converging to the actual distribution:

$$\mathrm{BCE}(\delta_{\theta_1,\theta_2}(\text{user}_i, \text{movie}_j), r_{ij})$$

3 Improved DMF

3.1 Reducing the approximation error

Definition 3.1: Approximation and estimation errors

Let's define the following three prediction functions:

1. $f^* = \operatorname{argmin}_f \mathbb{E}_{X,Y}[l(f(X), Y)]$, the best prediction function over all possible prediction functions.

2. $f_{\mathcal{F}} = \operatorname{argmin}_{f \in \mathcal{F}} \mathbb{E}_{X,Y}[l(f(X), Y)]$, the best prediction function over the function class $\mathcal{F}$.

3. $\hat{f}_n = \operatorname{argmin}_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^n l(f(x_i), y_i)$ the function over the class $\mathcal{F}$ that maximizes the empirical risk over the dataset $S = ((x_1, y_1), \ldots, (x_n, y_n))$.

We then define the **approximation error** of $\mathcal{F}$ to be $R(f_{\mathcal{F}}) - R(f^*)$ and the **estimation error** of $\hat{f}_n$ in $\mathcal{F}$ to be $R(\hat{f}_n) - R(f_{\mathcal{F}})$.

The improvements we made upon the initial vanilla deep matrix factorization algorithm were initially aimed at reducing the approximation error.

The initial model was asked to project the users' and movies' vectors such that the more a user would rate a movie, the higher their Euclidean canonical cosine similarity. We conjectured that such a class of prediction functions was sub-optimal in terms of approximation error because it asked to project two different objects in the same area (with respect to the Euclidean L_2 norm).

We therefore experimented with a model that learned its own scalar product within the class $\{(X, Y) \to X^T S^T SY,\ S \text{ invertible}\}$. This learned scalar product was used to compute the cosine similarity. We learned from end to end a matrix S and conjectured that S would always end up invertible. We did that to allow the model to project vectors how it wanted (because it would then compare them how it wanted, not based on an imposed L_2 norm). So we basically gave the model more expressivity. We observed experimentally improvement in the results.

However after an analysis, which you'll find in the appendix, we ended up finding that the improvement in the model didn't come from all the properties of learning a new distance metric. It actually only came from the learning of an affine transformation Aff_{θ_3} (a linear layer without activation) applied to the latent of the user's vector before taking the Euclidean cosine similarity:

$$\frac{\langle \text{Aff}_{\theta_3}(\text{NN}^{\text{movie}}_{\theta_1}(\text{movie}_j)), \text{NN}^{\text{user}}_{\theta_2}(\text{user}_i)\rangle_{L_2}}{\|\text{Aff}_{\theta_3}(\text{NN}^{\text{movie}}_{\theta_1}(\text{movie}_j))\|_{L_2}\|\text{NN}^{\text{user}}_{\theta_2}(\text{user}_i)\|_{L_2}}$$

Replacing our learning of a scalar product with the learning of an affine transformation yielded even better results. We tried learning a more complex class of non-linear transformations, but it decreased results (maybe not enough data).

Our conjecture about how the new model works is as follows: the model learns to project the vectors of movies and users into separate areas that are more suitable for them, where two movies (or two users) are collinear if they are similar, like in figure 6. Then the model uses the affine transformation on the user vectors to give a vector that is collinear with a movie if the rating is high, like in figure 9.

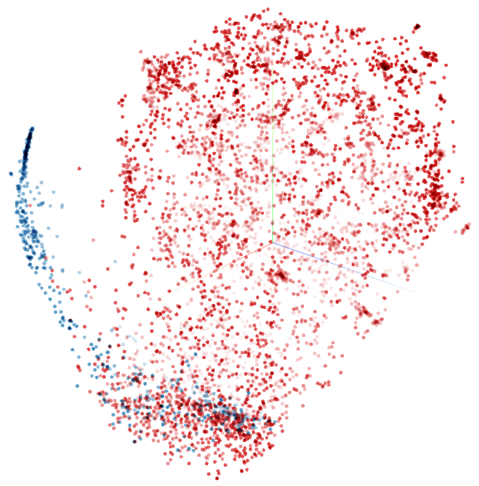
Figure 1: T-SNE of projections, view 1

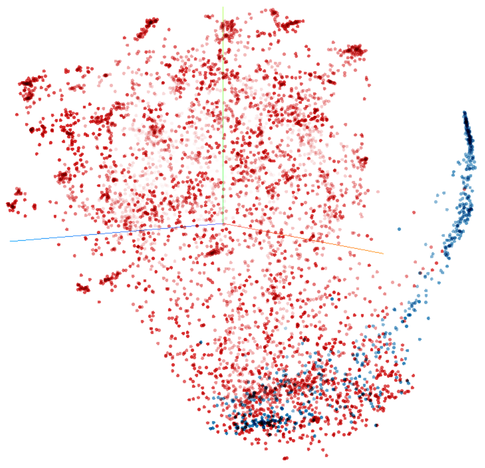
Figure 2: T-SNE of projections, view 2

Figure 3: Projections of the latents of the initial vanilla deep matrix factorization model. Movies are red, users are blue.

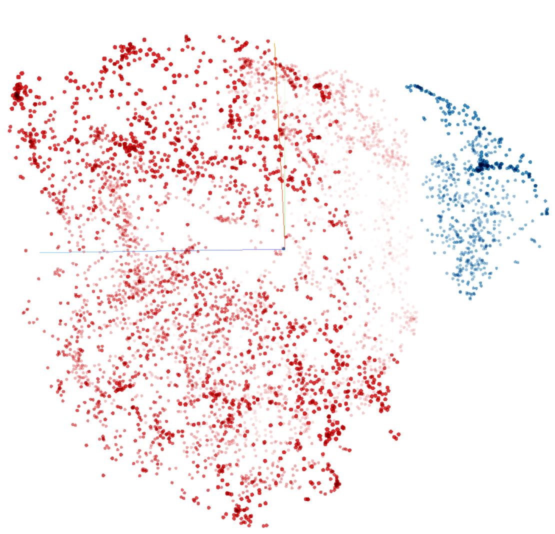

Figure 4: T-SNE of projections, view 1

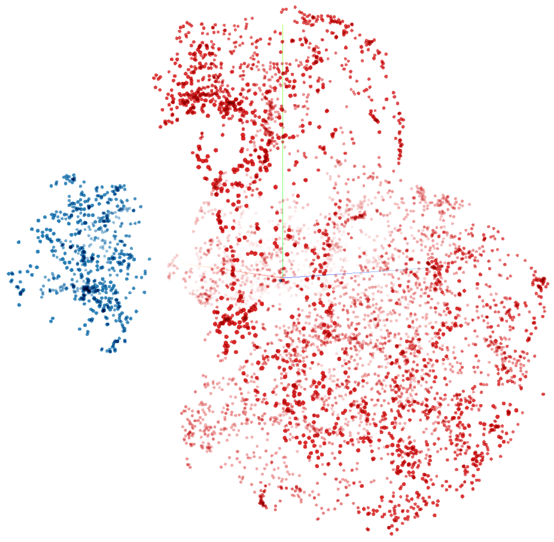

Figure 5: T-SNE of projections, view 2

Figure 6: Projections of the latents of our latest model, before the affine transformation is applied to the users' latents. Movies are red, users are blue.

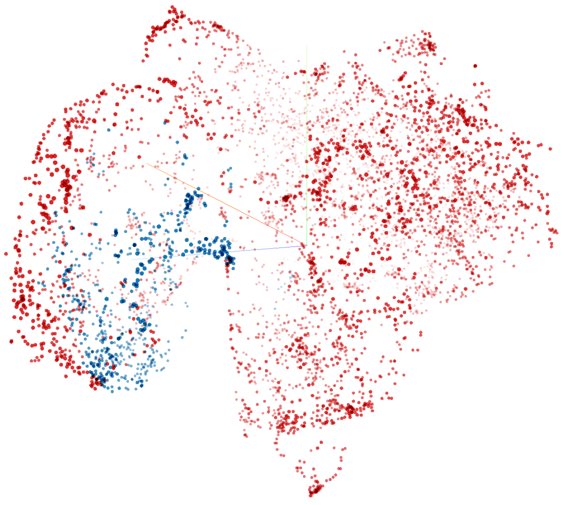

Figure 7: T-SNE of projections, view 1

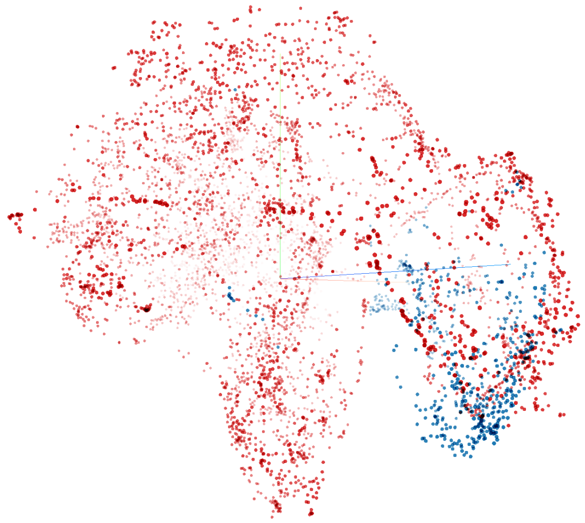

Figure 8: T-SNE of projections, view 2

Figure 9: Projections of the latents of our latest model, after the affine transformation is applied to the users' latents. Movies are red, users are blue.

We can see in figure 3 that with the initial vanilla deep matrix factorization algorithm many users' vectors are not well-projected into the area of the movies' vectors. So they don't really have meaningful similarities with movies' vectors. However with our latest model, as we can see in figure 9, all the users' vectors are well-mixed with the movies. Many movies don't mix with users but it's already better than with the initial vanilla deep matrix factorization where many movies' vectors didn't mix with users' vectors at all and many users' vectors didn't mix with movies' vectors at all.

3.2 Reducing the estimation error

The improvements we made upon the vanilla deep matrix factorization algorithm concern implicit regularization as well. The general idea was to modify the original model to inject into it constraints that would not increase the approximation error. We conjecture that these constraints would improve the estimation error (regularization) while not increasing the approximation error.

The constraints we are talking about is the adjunction of an auxiliary loss (reconstruction loss). The new structure is a division of the model into the following two parts:

- Two auto-encoders (that we conjecture would learn meaningful latents of the initial movies' vectors and users' vectors) optimized to reconstruct the input movies' vectors and the input users' vectors.
- We learn an affine transformation (thus, in particular, a transformation that preserves collinearity) that remaps the users' latents so that the euclidean canonical cosine similarity between the remapped user's latent and a movie is proportional to the ratings of this user for this movie.

The initial vanilla deep matrix factorization model wasn't divided into these two parts.

We optimized these two parts conjointly. We didn't test a separate optimization scheme.

3.3 Partial ablation study

We only ran a partial ablation study, as we thought we would have one more week of time after the end of the project to hand over the report and only learned on Monday that the report was due on Tuesday. We report mean RMSE obtained over 20 folds:

Everything	scalar product	w/t product/affine	w/t reconstruction	w/t metadata
0.8135	0.8171	0.8244	0.8234	0.8143

Everything means our final model with affine transformation learning instead of scalar product learning.
Scalar product means we learned a scalar product instead of an affine transformation.
w/t product/affine means we learned neither a scalar product nor an affine transformation.
w/t reconstruction means we didn't use a decoder and a reconstruction regularization loss.
w/t metadata means we didn't add metadata to the movies' vectors.

4 Appendix

4.1 Analysis of the contribution of unknown values to the gradient

Intuitively, as the settings with the unknown values are wrong (an unknown value is not the same object as a zero rating, yet we must encode them somehow into scalars), we'd like to remove the influence of the zero encoding of unknown values on the gradient update. We can formally investigate their influence, remove it, and compare experimentally the results.

We investigate this influence with the following mathematical derivations:

$$
\begin{aligned}
Wx + b = \big([\sum_k w_{lk} x_k] + b_l\big)_l \\
\frac{\partial f(Wx+b)}{\partial w_{ij}} &= \sum_l \frac{\partial f}{\partial y_l} \frac{\partial [\sum_k w_{lk} x_k] + b_l}{w_{ij}} \\
&= \frac{\partial f}{\partial y_i} \frac{\partial w_{ij} x_j + b_i}{\partial w_{ij}} \\
&= \frac{\partial f}{\partial y_i} x_j
\end{aligned}
$$

Meaning that if $x_j = 0$:

$$
\forall i, \; \frac{\partial f(Wx+b)}{\partial w_{ij}} = 0
$$

This means that the parameters of the first layer associated with an unknown value aren't updated *when using vanilla stochastic gradient descent* (not using batches). This indicates that a zero encoding for unknown values limits their influence on gradient updates.

When we use batched stochastic gradient descent with a gradient averaging approach this is what happens:

$$
\begin{aligned}
\forall i, \; \frac{\frac{\sum_m^{\text{batch size}} \partial f(Wx^m+b)}{\text{batch size}}}{\partial w_{ij}} &= \frac{1}{\text{batch size}} \sum_m^{\text{batch size}} \frac{\partial f}{\partial y_i} x_j^m \\
&\leq \frac{1}{\#\{x_j^m, x_j^m \neq 0\}} \sum_m^{\text{batch size}} \frac{\partial f}{\partial y_i} x_j^m
\end{aligned}
$$

With this gradient-averaging strategy, we can see the influence of zero values on their associated weights: they diminished non-uniformly the value of the update. We therefore tried a summing strategy of the gradients (instead of an averaging strategy) and computed manually the averaging coefficient for each parameter with the goal of removing the influence of zero values. It decreased results.

4.2 From scalar product learning to affine transformation learning

Warning: $(X, Y) \to X^T S^T S Y$ is a scalar product only if S is invertible. We consider that is it very unlikely to learn a perfectly non-invertible matrix via SGD and thus consider $(X, Y) \to X^T S^T S Y$ to be a scalar product.

$$
\begin{aligned}
\langle X, Y \rangle_2 = X^T Y \\
X^T S^T S Y = (SX)^T S Y = \langle SX, SY \rangle_2 \\
X^T S^T S Y = (S^T S X)^T Y = \langle S^T S X, Y \rangle_2 = \langle X, S^T S Y \rangle_2
\end{aligned}
$$

Thus learning and using this new scalar product ($\langle \cdot, \cdot \rangle : (X, Y) \to X^T S^T S Y$) is tantamount to :

- Transforming X with $S^T S$ and applying canonical cosine similarity between the transformed vector and Y.
- Transforming X and Y with the same matrix and applying canonical cosine similarity to the transformed vector.

But what exactly in the learning of the scalar product yields better results? Here are some examples of what we could do that is somehow similar to the above:

- Learn a linear transformation L (a Linear layer without bias and activation) and minimize $\frac{\langle LX,Y\rangle_2}{\|LX\|_2\|Y\|_2}$, it's the same as above but we neither impose L to be definite positive nor symmetric.
- Learn an upper-triangular matrix T, create a symmetric matrix M by replicating the upper-triangle of T above and under. Minimize $\frac{\langle MX,Y\rangle_2}{\|MX\|_2\|Y\|_2}$. This is almost what we are doing with the learning of a scalar product, except that M is just symmetric, not positive-definite.

We tested several somehow similar strategies and it appeared the learning of a linear/affine transformation was what yielded better results. So we conjectured that it's this part of learning a scalar product that yielded better results.

5 Results on the platform

After several trials on the platform, the result of different approaches is the following:

Method	RMSE test set	RMSE platform	Time	Rank platform
GD MF	0.88	0.86	20s	1
ALS MF	0.91	0.87	15s	1
iDMF	**0.85**	**0.80**	250s*	1

GD MF: Gradient descent matrix factorization
ALS MF: Alternating least squares matrix factorization
iDMF: improved deep matrix factorization
*: improvable by parallelization

References

[1] H.-J. Xue, X. Dai, J. Zhang, S. Huang, and J. Chen, "Deep matrix factorization models for recommender systems.," in *IJCAI*, Melbourne, Australia, vol. 17, 2017, pp. 3203–3209.