Recommender Systems

Silin DU

Department of Management Science and Engineering, Tsinghua University
dsl21@mails.tsinghua.edu.cn

June 6, 2023

清華大學
Tsinghua University

Contents

1 Data Preprocessing

In this report, we focus on the recommendation task on a subset of the famous Netflix competition. The selected dataset contains $10,000$ users and $10,000$ movies (items). There are several files in our dataset, including

- users.txt: stores the ID of each user.
- netflix_train.txt: training data of our recommendation task which contains around $6,890,000$ ratings from users to items. Ratings are from 1 to 5.
- netflix_train.txt: testing data which contains around $1,700,000$ ratings from users to items.
- movie_titles.txt: additional information about the titles and the time of the movies.

Now we know that the number of users m and items n are both $10,000$. The training and testing data are originally organized like (user_id, item_id, rating). We need to transform them into two matrices $X_{\text{train}}, X_{\text{test}} \in \mathbb{R}^{m \times n}$.

After loading the data by the `Pandas` library, we can leverage the `DataFrame.pivot()` method to generate the required matrices. Concretely,

1. We first reassign IDs to each user, from 1 to $10,000$.
2. Then we merge the training and testing data with the new user IDs.
3. Finally, we call the `DataFrame.pivot()` method to generate X_{train} and X_{test}, and fill the null values by 0.

Besides, due to the sparsity of the interaction matrices, we can also leverage the sparse matrix `coo_matrix` provided in `scipy.sparse`. We first create a `coo_matrix` object and then transform it into dense `numpy.ndarray`. We provide sample codes below.

```
1  train_coo = coo_matrix((train_data, (train_row, train_column)),
       shape=(num_user, num_movie))
2  X_train = train_coo.toarray()
```

Note that both methods take no more than 1 min to finish the transformation and we do not drop any data in this step. The codes for both methods are provided.

2 Collaborative Filtering

In this part, we leverage user-based collaborative filtering (user-based CF) to predict the rating scores in the testing data. Assume that X_{train} uses 0 for non-exist ratings. For user i and item j, we estimate the rating score from i to j based on the following equation:

$$\widehat{r_{ij}} = \frac{\sum_{k=1}^{K} \text{sim}\left(X_{i\cdot}, X_{k\cdot}\right) \cdot r_{kj}}{\sum_{k=1}^{K} \text{sim}\left(X_{i\cdot}, X_{k\cdot}\right)} \tag{1}$$

where $r_{kj} \in X_{\text{train}}$ is the known rating score from user k to item j, $X_{i\cdot} \in \mathbb{R}^n$ is the ratings from user i to all items, and $\text{sim}\left(X_{i\cdot}, X_{k\cdot}\right)$ is the similarity score between user i and k. We

use cosine similarity score here, which is defined by

$$\text{sim}\left(X_{i\cdot}, X_{k\cdot}\right) = \frac{X_{i\cdot}^T X_{k\cdot}}{\left|X_{i\cdot}\right| \cdot \left|X_{k\cdot}\right|}$$

In the implementation, we use the `cosine_similarity` function in `sklearn.metrics` to calculate the similarity scores between every two users, and get the similarity matrix. To estimate the rating score from user i to user j, we then find the top-K similar user to i according to the similarity matrix. Here K is a predefined hyperparameter. Finally, we predict $\widehat{r_{ij}}$ via Equation (1).

However, we find that this method cannot make desired predictions. Therefore, we make several adjustments.

1. Again, consider estimating $\widehat{r_{ij}}$. When selecting the top-K user, we only consider users that have rating records to item j, which are named as valid users for item j. Let q_j be the number of valid users for item j. If $q_j < K$, we only leverage q_j users for predictions. In our dataset, $q \geq 2$.
2. It's common that different users will have diverse rating habits. For some users, 3 might be the borderline score, which means the item is not particularly good or bad, while for other users, it might represent very negative score. Similar patterns can be observed among items. Therefore, we decompose each rating into two parts, i.e., the baseline score and the preference score.

$$r_{ij} = \underbrace{\mu + b_i + b_j}_{b_{ij}\text{: baseline score}} + \underbrace{p_{ij}}_{\text{preference score}}$$

where μ is the average of all rating scores, b_i is the rating deviation of user i (b_i = average rating scores of user $i - \mu$), b_j is the rating deviation of item j (b_j = average rating scores of item $j - \mu$). Note that all the averages are calculated on X_{train}.

After these two adjustments, we modify Equation (1) as follows.

$$\widehat{r_{ij}} = b_{ij} + \frac{\sum_{k=1}^{\min(q_j,K)} \text{sim}\left(X_{i\cdot}, X_{k\cdot}\right) \cdot \left(r_{kj} - b_{kj}\right)}{\sum_{k=1}^{\min(q_j,K)} \text{sim}\left(X_{i\cdot}, X_{k\cdot}\right)} \tag{2}$$

When $q_j = 0$, we directly predict $\widehat{r_{ij}} = b_{ij}$. But there are no such items in our dataset.

To implement Equation (2), we first calculate the μ and b_i, b_j for each user and item. Then we demean X_{train}, i.e., use $X_{i\cdot}$ to minus the average rating score of user i (equal to $\mu + b_i$) for user $i = 1, ..., m$. We denote the new matrix as X'_{train}, which is further used to calculate the similarity matrix. To predict $\widehat{r_{ij}}$, we find the top-K (or q_j) similar valid users (corresponding to item j) to i based on the similarity matrix. Finally, we predict $\widehat{r_{ij}}$ via Equation (2).

To evaluate our method, we leverage the Root Mean Square Error (RMSE), defined by

$$\text{RMSE} = \sqrt{\frac{1}{\text{non-zero}\left(X_{\text{test}}\right)} \sum_{\substack{i,j \\ r_{ij} \in X_{\text{test}}, r_{ij} \neq 0}} \left(r_{ij} - \widehat{r_{ij}}\right)^2}$$

where non-zero(X_{test}) calculates the number of non-zero elements in X_{test}.

We highlight several details and conclusions in this part as follows.

1. Assume that all 5 ratings have equal probability. Then if we 'guess' an all-3 prediction for all samples, the RMSE of this random guess is $\sqrt{2} \approx 1.414$. While in our testing data, the RMSE of an all-3 prediction is 1.1715. We use the latter number as our baseline hereafter.
2. In our experiments, if we consider **all users**, we can leverage matrix multiplication to speed up the calculation. Particularly, let $S \in \mathbb{R}^{m \times m}$ be the similarity matrix, A be a 0-1 indicator matrix with $A_{ij} = 1$ if $r_{ij} \in X_{\text{train}}, r_{ij} \neq 0$, then we can predict X_{test} through

   $$\widehat{X_{\text{test}}} = \left(SX_{\text{train}}\right) / \left(SA\right)$$

   where / denotes the element-wise division. Once we use this matrix form formula, it only takes no more than 1 min to make all predictions. Note that if we replace the indicator matrix A with an all-1 matrix, it means we do not select the valid users.
   If we only consider the top-K users, it's a little bit complicate to derive the matrix form formula. Thus, we make predictions through the `for` loop. Our codes take around 20 mins to predict around $1,700,000$ samples in the testing data.
3. In the implementation of Equation (2), we find that in some rare cases, the similarity scores between user i and its valid users are quite small (very close to 0). We directly predict $\widehat{r_{ij}} = b_{ij}$ in those cases.
4. When using Equation (1) for predictions, we get a quite large RMSE, reaching over 2.2. The most important reason is that we do not select valid users for each item. After selecting the valid users (Adjustment 1), we can get a RMSE of 1.0184, which is 13.07% lower than the baseline.
5. When using Equation (2) with $K = 3$, the RMSE is around 0.9375, which is 19.97% lower than the baseline.

3 Matrix Factorization

3.1 Implementation Details

In this part, we implement the Matrix Factorization method (MF) to generate recommendations. In MF, we believe that the preferences of each user i can be represented by

a low dimensional vector $u_i \in \mathbb{R}^k$, where k is the dimension of latent vectors and is predefined. Note it's different from k and K in Section 2. Similarly, we can also use a latent vector $v_j \in \mathbb{R}^k$ to represent the information of item j. Then we can make predictions via

$$\widehat{r_{ij}} = u_i^T v_j \tag{3}$$

Equivalently, it can be viewed as the interaction matrix $X \in \mathbb{R}^{m \times n}$ is factorized into two low dimensional matrices, i.e.,

$$X = UV^T$$

where $U \in \mathbb{R}^{m \times k}$ contains latent vectors for all users, and $V \in \mathbb{R}^{n \times k}$ contains latent vectors for all items.

Our goal now is to learn two matrices U and V, such that they can recover the interaction matrix well. To achieve this goal, we can define the following loss function during training

$$J = \frac{1}{2}\left\| A \odot \left(X_{\text{train}} - UV^T \right) \right\|_F^2 + \lambda \left(\|U\|_F^2 + \|V\|_F^2 \right) \tag{4}$$

where A is a 0-1 indicator matrix with $A_{ij} = 1$ if $r_{ij} \in X_{\text{train}}, r_{ij} \neq 0$, $\odot$ is the element-wise multiplication, $\|\cdot\|_F$ is the Frobenius norm, and λ is a predefined hyperparameter to normalized the latent vectors of users and items. Taking derivative with respect to U and V yields

$$\begin{aligned} \frac{\partial J}{\partial U} &= \left(A \odot \left(UV^T - X \right) \right) V + 2\lambda U \\ \frac{\partial J}{\partial V} &= \left(A \odot \left(UV^T - X \right) \right)^T U + 2\lambda V \end{aligned}$$

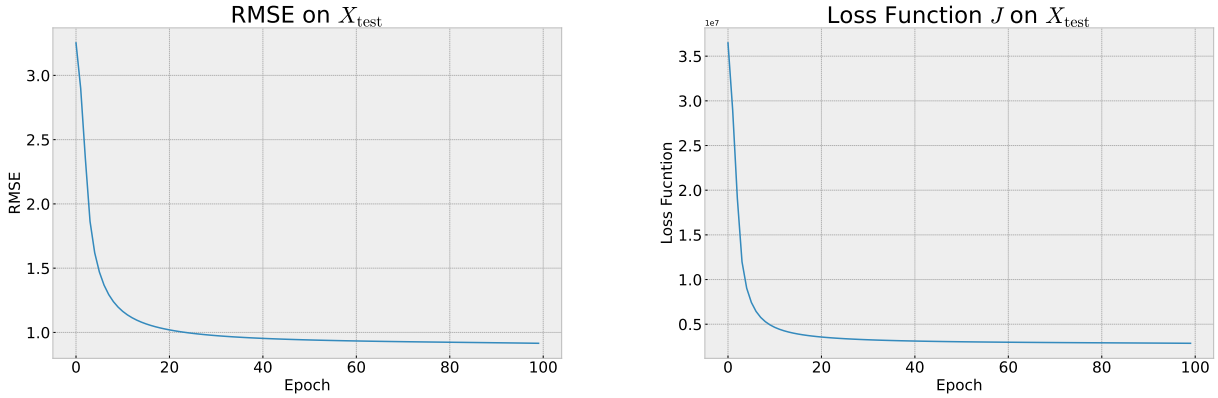
Then we can leverage gradient descent to minimize J in Equation (4), i.e., update U and V in each iteration by

$$U = U - \alpha \frac{\partial J}{\partial U}, \quad V = V - \alpha \frac{\partial J}{\partial V} \tag{5}$$

where α is a predefined hyperparameter and is called the learning rate.

In the implementation, we first randomize two matrices $U \in \mathbb{R}^{m \times k}, V \in \mathbb{R}^{n \times k}$ using `numpy.random.randn`, i.e., each element in these two matrices is drawn from a standard normal distribution. To avoid too large normalization term ($\|U\|_F^2 + \|V\|_F^2$), we multiply each element in U and V by 0.1. Then we set the learning rate as $\alpha = 10^{-4}$ and the number of iteration (epoch) as 100. Here we follow the standard gradient descent algorithm, i.e., update U and V via Equation (5) using the whole training data. Meanwhile, we will record the loss function J and the RMSE on X_{test} during each iteration. At last, once we get well-trained U and V, we can make predictions for each non-zero entry in X_{test} via Equation (3).

When the dimension of latent vectors k is 50, and the weight of normalization term λ is 0.01, we plot the change of J and RMSE on X_{test} in Figure 3.1. We can observe that the loss function and RMSE both decrease as the number of iteration increases. And they both converge after about 60 iterations. At last, we can get a RMSE on X_{test} of 0.9155, which is 21.85% lower than the baseline.



(a) RMSE on X_{test}.

(b) J on X_{test}.

Figure 3.1: The performance of the MF method when $k = 50, \lambda = 0.01, \alpha = 1e-4$.

Moreover, we also conduct experiments on more hyperparameter combinations, where $k \in \{10, 50, 100\}$ and $\lambda \in \{1, 0.1, 0.01, 0.001\}$. All results are shown in Table 3.1. All models are trained 100 epochs with $\alpha = 1e-4$. They both converge after around 80 iterations. We can observe that all MF models outperform the CF method in Section 2. And when k is small, choosing a small λ will provide better performance (see the result when $k = 10, \lambda = 0.01$). But when k is large, we need a large λ to prevent the models from overfitting. Finally, we find that the performances of our models are stable and not sensitive to λ and k.

Table 3.1: Experimental results of different hyperparameter combinations

RMSE λ / k	1	0.1	0.01	0.001
Matrix Factorization				
10	0.9230	0.9223	0.9229	0.9208
50	0.9156	0.9152	0.9155	0.9156
100	**0.9127**	0.9202	0.9220	0.9217
Other Methods	RMSE			
User-based CF	1.0184			
User-based CF with Decomposition	0.9375			
Baseline (All-3 Predictions)	1.1715			

We conclude this subsection with the following points.

1. A lot of matrix multiplications are involved during the training of U and V. We run our codes on a CPU, and it takes about 7 seconds for each iteration when $k = 50$. It might be better if we transfer the calculation to GPUs, which can leverage CUDA for faster matrix multiplication. To achieve this, we can utilize several famous open-source frameworks, such as `Pytorch`.
2. Since the size of training data is acceptable to some extend, we use the whole data in each update of the parameters. If the dataset is larger, we might need a batch-style training algorithm, which only selects a subset of the training data in each iteration.
3. Due to the limitation of the computational resources, we do not systematically fine-tune the learning rate α, which can be chosen from 10^{-1} to 10^{-5} in our context. But, after some informal explorations, we find that too large α would hurt the final RMSE.

3.2 Comparison with CF

We summary the comparison between MF in this section and CF in Section 2.

1. The CF method is a kind of 'lazy' model, which takes no cost for building and storing a model, but it has relatively high cost in model usage and prediction. In our implementation, it's fast to get the similarity matrix, but it's quite slow to make predictions for the testing data if we need to select the top-K users.
   The MF method takes a long time to train, but it's quite effective in model usage and prediction. We need more than 10 mins to get well-trained U and V, but only no more than 1 min to make all predictions for the testing data. It also requires necessary spaces to store the latent vectors for users and items.
2. The CF method has two variants, i.e., the user-based CF and the item-based CF. The user-based CF in Section 2 requires a large space to store the user-item iteration matrix, which is not suitable if the number of users changes frequently.
   However, at the end of the last subsection, we mention that we can use a batch-style training method for MF. It means MF does not require a large space to store the user-item iteration matrix, and an MF model can be trained directly via the records, just like our original data. Therefore, the MF method is much more scalable.
3. The user-based CF suffers from the sparsity problem of the interaction matrix and it's difficult to find similar users in some cases. We already mention this problem in the details of our implementation at the end of Section 2.
   The item-based CF, although not used here, suffers from the problem that popular items which receive many interactions will be similar to a large amount of items, while it might be difficult to find similar items for some unpopular ones. This can lead to over-recommending popular items and reduce the diversity of recommendations, which is known as popularity bias.
   In the situation where it's hard to find similar users or items, MF can still make reasonable predictions, since it models the latent preference of users and information

of items. Note that the MF method also has the popularity bias problem. From Table 3.1, we can observe that all MF methods can outperform the CF methods.

4. Since the CF method is a kind of 'lazy' model, it can only leverage the interaction matrix. In many contexts, we have additional information about users and items. But they cannot be utilized by the CF method.
   In contrast, the MF method is more flexible. For example, we can use additional information about users and items to initialize the latent vectors, which might provide better predictions. Thus, the MF method is more flexible and extendable.

4 Conclusion

In this report, we focus on the recommendation tasks on the Netflix dataset. We first prepare the data for modeling in Section 1. Then, we implement a user-based CF method to make predictions in Section 2. During this part, we find that some necessary adjustments will bring significant improvements, such as the selection of valid users and the decomposition of the rating scores. After that, we build an matrix factorization model in Section 3, and compare the performances and convergence rate when the key hyperparameters take different values. We find that the MF methods with different hyperparameters can all outperform the CF methods.