Factorization meets the neighborhood
: a multifaceted collaborative filtering model

&

Matrix Factorization Techniques for Recommender Systems

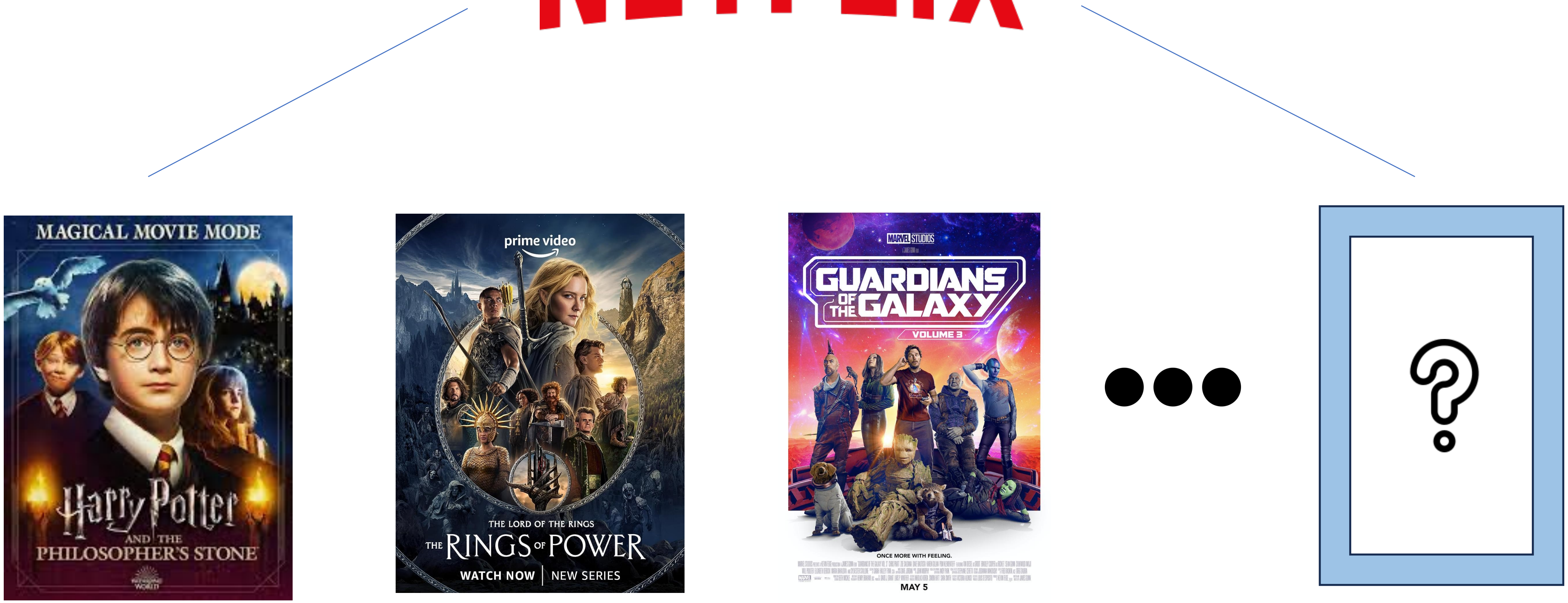
Yehuda Koren, Robert Bell and Chris Volinsky

JongGeun Lee

Overview

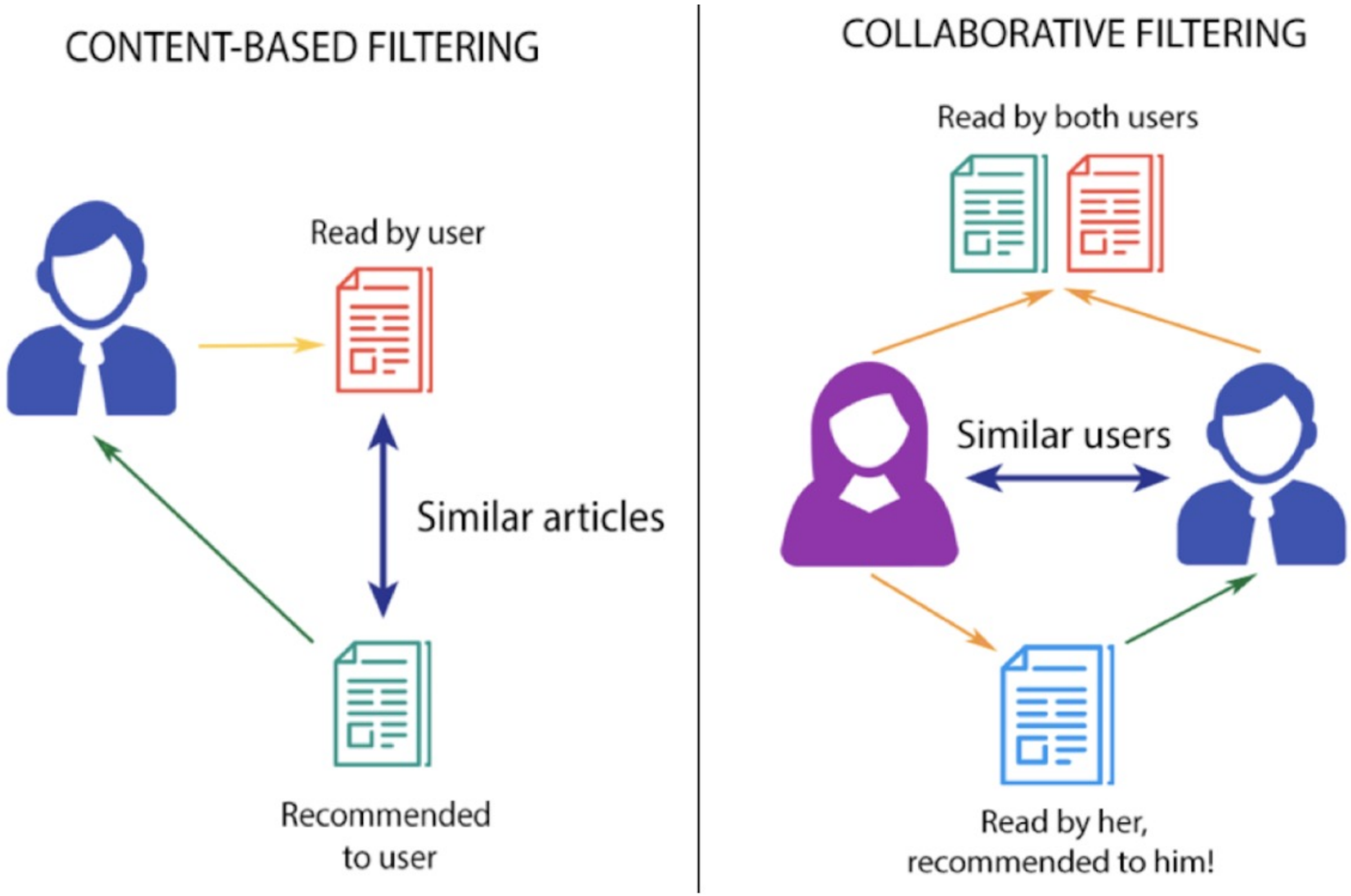
1. Introduction – What is RecSys?

NETFLIX



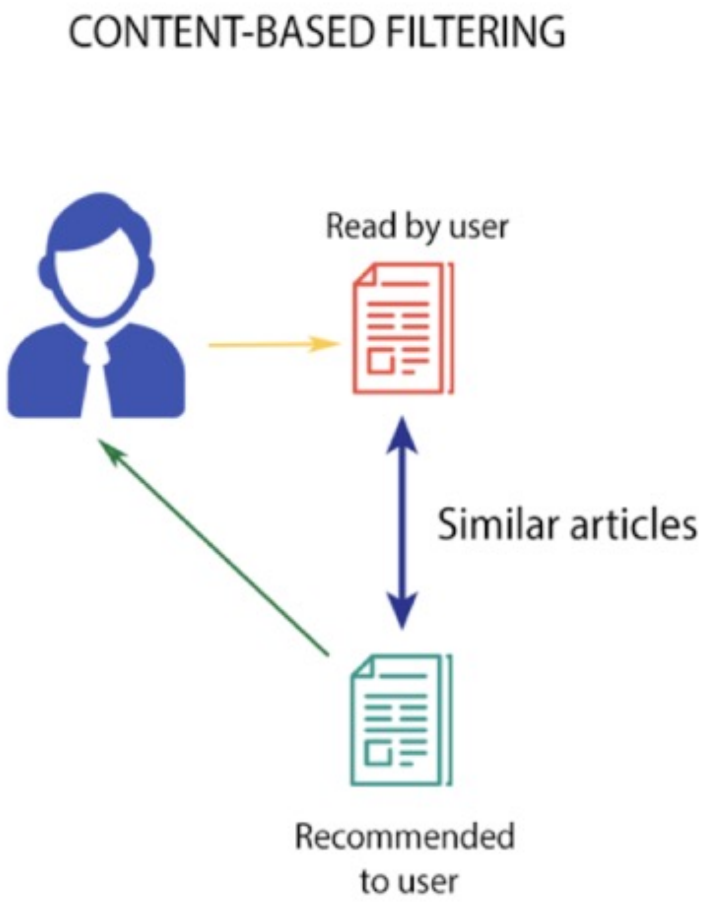
User Satisfaction & Loyalty

1. Introduction – What is RecSys?



- Towards Data Science, "Brief on Recommender Systems". (2019).

1. Introduction – Content-based RecSys



∗Content-based ?

A method of recommending products similar to those that each user has purchased or been satisfied with.

- ✓ same genre.
- ✓ same director or featuring the same actors.
- ✓ products in the same category.

∗How?

- ✓ Create User Profiles and Item Profiles
- ✓ Matching Users and Items based on similarity (Cosine-Similarity, Pearson-Coefficient)

- Towards Data Science, "Brief on Recommender Systems". (2019).

1. Introduction – Collaborative Filtering

∗Neighborhood Model Process

1. Find existing users with similar tastes to the user.
2. Find products favored by users with similar tastes.
   - ✓ Using such a correlation coefficient, find the top K users whose tastes are most similar to target user
   - ✓ Estimate the ratings through the weighted average of ratings, using the similarity of tastes as weights.

3.Recommend these products to the user.
   - ✓ Estimate ratings for all products not purchased by the target user X, then recommend the products with the highest ratings.

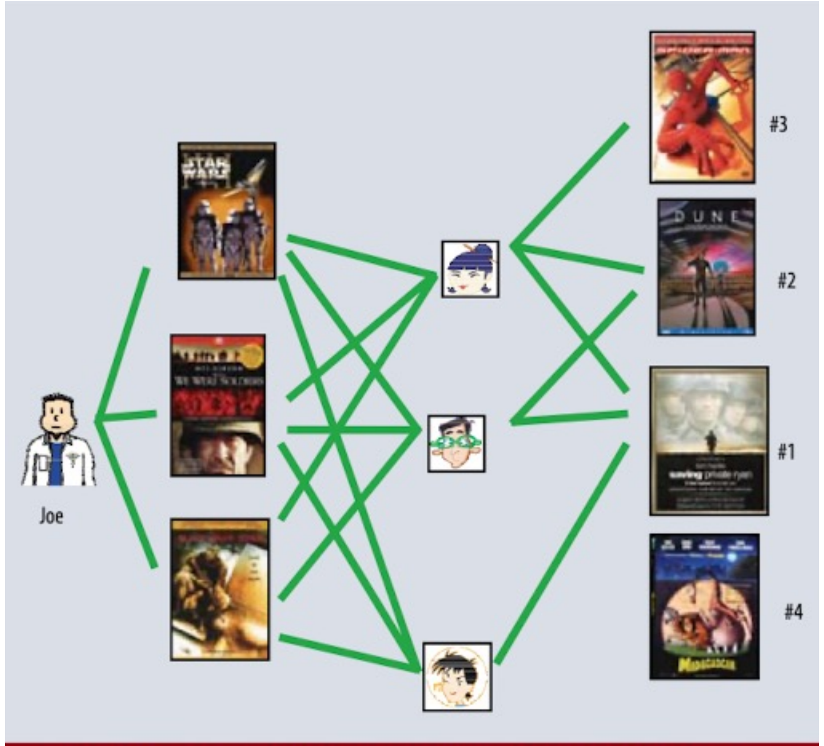


Figure 1. The user-oriented neighborhood method. Joe likes the three movies on the left. To make a prediction for him, the system finds similar users who also liked those movies, and then determines which other movies they liked. In this case, all three liked *Saving Private Ryan*, so that is the first recommendation. Two of them liked *Dune*, so that is next, and so on.

1. Introduction – Collaborative Filtering

***Latent Factor Model**

- ✓ The core of the Latent Factor Model is to represent users and products as vectors.

How to perform embedding?

  - ✓ Train so that the inner product of each user and product embedding vector is as similar as possible to the rating.
    - ✓ The embedding of user x is p_x, and the embedding of product i is q_i.
    - ✓ The rating of user x for product i is r_xi.
    - ✓ The goal of embedding is to train so that p_x^t * q_i is similar to r_xj.

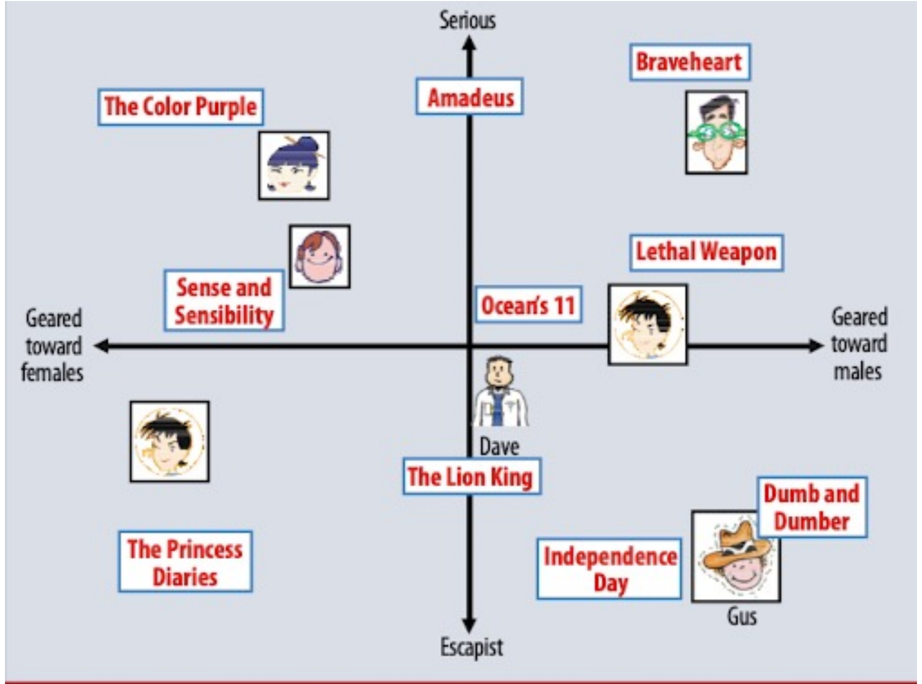


Figure 2. A simplified illustration of the latent factor approach, which characterizes both users and movies using two axes—male versus female and serious versus escapist.

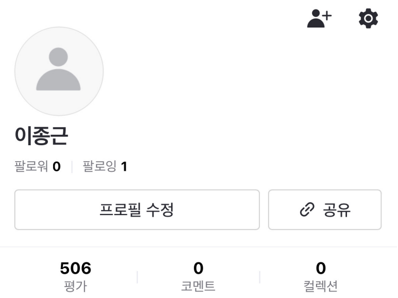
*Explicit Feedback

: includes explicit input by users regarding their interest in products

Star Ratings

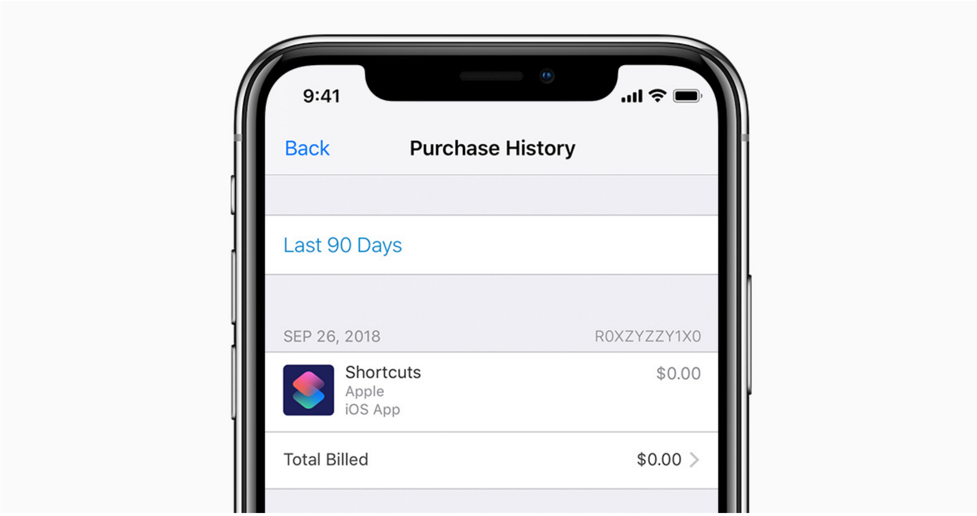
Thumbs-Up/Down

WATCHA PEDIA



*Implicit Feedback

- Purchase history
- Browsing history
- Search patterns
- Mouse movements



1. Introduction – Novelty

Neighborhood Model

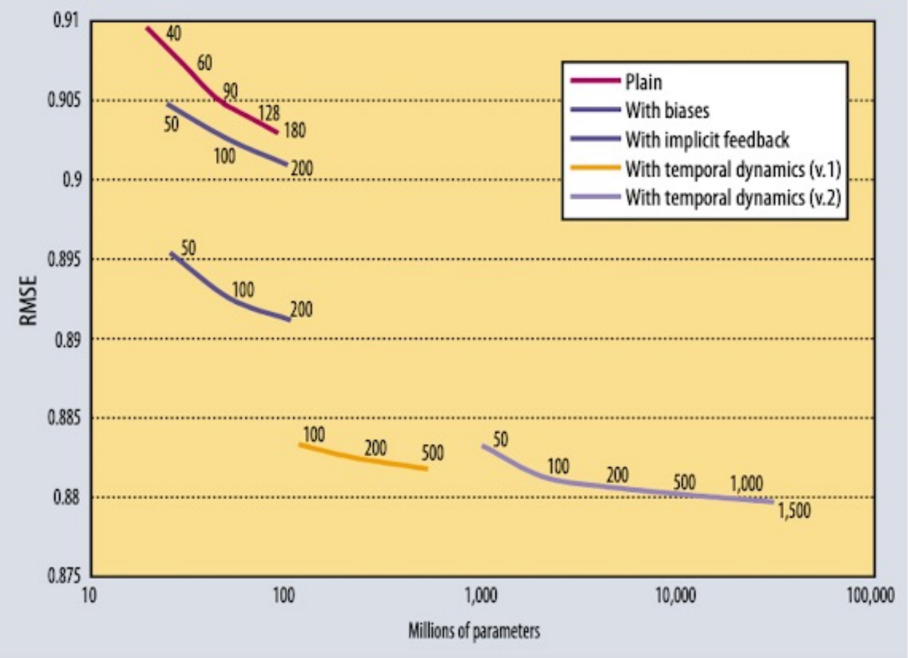
Latent Factor Model

+ Explicit Feedback

+ Implicit Feedback

"Integrated Model"

1. Introduction – Novelty



- Does achieving a slightly better RMSE result in a completely different and better recommendation?

"Top-K recommender"

2. Preliminaries – Baseline Model

$$b_{ui} = \mu + b_u + b_i$$

μ : *overall average rating*
b_u: *observed deviation of user u*
b_i : *observed deviation of item i*

$$\min_{b_*} \sum_{(u,i)\in\mathcal{K}} (r_{ui} - \mu - b_u - b_i)^2 + \underbrace{\lambda_1\left(\sum_u b_u^2 + \sum_i b_i^2\right)}$$

Regularizing term
: to avoid overfitting by penalizing the magnitudes of the parameters

2. Preliminaries – Neighborhood Model

Not just a Pearson Correlation Coefficient
: Shrunk Correlation Coefficient

ρ_{ij} : $Pearson\, Correlation\, Coefficient$

λ_2 : $100\ (typical\ value)$

η_{ij} : $the\ number\ of\ users\ that\ rated\ both\ i\ and\ j.$

$$s_{ij} \stackrel{\text{def}}{=} \frac{n_{ij}}{n_{ij} + \lambda_2} \rho_{ij}$$

In the case of a sparse matrix, two different items i,j may be reflected as close when they are not.

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{j \in \mathrm{S}^k(i;u)} s_{ij}(r_{uj} - b_{uj})}{\sum_{j \in \mathrm{S}^k(i;u)} s_{ij}}$$

Aka. CorNgbr

$S^k(i;u)$: $set\ of\ k-neighborhoods\ (for\ user\ u,\ item\ i)$

2. Preliminaries – Interpolation Weight

*Problems of K-Nearest Neighborhood

1. NN methods are not good at accounting for global effects.
2. Previous weighting methods fail to account for interdependencies between neighbors.
3. Previously, interpolation weights always sum to one. If there is no useful neighborhood information then it is better to ignore it.
4. NN methods work poorly if the number of ratings differs substantially.

$$\hat{r}_{ui} = b_{ui} + \sum_{j \in \mathrm{S}^k(i;u)} \theta_{ij}^u (r_{uj} - b_{uj})$$

Aka. WgtNgbr

2. Preliminaries – Latent Factor Model

*Standard SVD Model

$$\hat{r}_{ui} = b_{ui} + p_u^T q_i$$

$$p_u \rightarrow \frac{\sum_{j \in R(u)} x_j}{\sqrt{|R(u)|}}$$

: to avoid explicitly parameterizing each user, but rather models users based on the items that they rated

*Paterek's NSVD Model

$$b_{ui} + q_i^T \left(\sum_{j \in \mathrm{R}(u)} x_j \right) / \sqrt{|\mathrm{R}(u)|}.$$

$R(u)$: *the set of items rated by user* u

3. Revised Model – Neighborhood Model

*Interpolation Weights model

$$\hat{r}_{ui} = b_{ui} + \sum_{j \in \mathrm{S}^k(i;u)} \cancel{\theta_{ij}^{u}} (r_{uj} - b_{uj})$$

Instead of user-specific weights,
to facilitate global optimization, use global weights independent of a specific user

$$\hat{r}_{ui} = b_{ui} + \sum_{j \in \mathrm{R}(u)} (r_{uj} - b_{uj}) w_{ij} + \underline{\sum_{j \in \mathrm{N}(u)} c_{ij}}$$

→ Implicit feedback

$$R(u) : the\, set\, of\, items\, rated\, by\, user\, u$$

$$N(u) : the\, set\, of\, items\, for\, a\, implicit\, preference\, of\, user\, u$$

3. Revised Model – Neighborhood Model

$$\hat{r}_{ui} = \mu + b_u + b_i + |\mathrm{R}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{R}(u)} (r_{uj} - b_{uj}) w_{ij} + |\mathrm{N}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{N}(u)} c_{ij}$$

For Computation Efficiency,
Parameter Pruning

$$\hat{r}_{ui} = \mu + b_u + b_i + |\mathrm{R}^k(i;u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{R}^k(i;u)} (r_{uj} - b_{uj}) w_{ij} + |\mathrm{N}^k(i;u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{N}^k(i;u)} c_{ij} \quad (10)$$

$R^k(i;u)$: *the set of rated k – neighborhoods (for user u, item i)*

$N^k(i;u)$: *the set of k – neighborhoods for a implicit preference (for user u, item i)*

3. Revised Model – How to Solve ?

*Alternative Least Square Method

$\mathbf{X}:\ n \times (d+1)\ matrix,\ \mathbf{y}:\ n \times 1\ vector$

$\hat{\beta}: (d+1) \times 1\ vector$

$$\min E(\mathbf{X}) = \frac{1}{2}\left(\mathbf{y} - \mathbf{X}\hat{\beta}\right)^T\left(\mathbf{y} - \mathbf{X}\hat{\beta}\right)$$

$$\Rightarrow \frac{\partial E(\mathbf{X})}{\partial \hat{\beta}} = -\mathbf{X}^T\left(\mathbf{y} - \mathbf{X}\hat{\beta}\right) = 0$$

$$\Rightarrow -\mathbf{X}^T\mathbf{y} + \mathbf{X}^T\mathbf{X}\hat{\beta} = 0$$

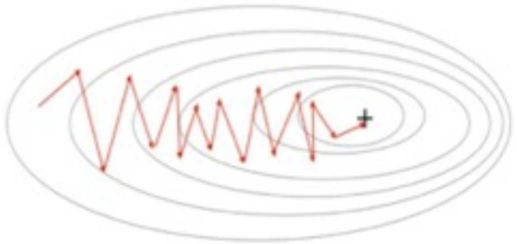
$$p_u = (Q^TQ + \lambda I)^{-1}Q^Tr_u$$

$$q_i = (P^TP + \lambda I)^{-1}P^Tr_i$$

*Stochastic Gradient Descent

- $b_u \leftarrow b_u + \gamma \cdot (e_{ui} - \lambda_4 \cdot b_u)$
- $b_i \leftarrow b_i + \gamma \cdot (e_{ui} - \lambda_4 \cdot b_i)$
- $\forall j \in \mathrm{R}^k(i;u):$
  $w_{ij} \leftarrow w_{ij} + \gamma \cdot \left(|\mathrm{R}^k(i;u)|^{-\frac{1}{2}} \cdot e_{ui} \cdot (r_{uj} - b_{uj}) - \lambda_4 \cdot w_{ij}\right)$
- $\forall j \in \mathrm{N}^k(i;u):$
  $c_{ij} \leftarrow c_{ij} + \gamma \cdot \left(|\mathrm{N}^k(i;u)|^{-\frac{1}{2}} \cdot e_{ui} - \lambda_4 \cdot c_{ij}\right)$

Stochastic Gradient Descent



3. Revised Model – Comparison of ngbrs

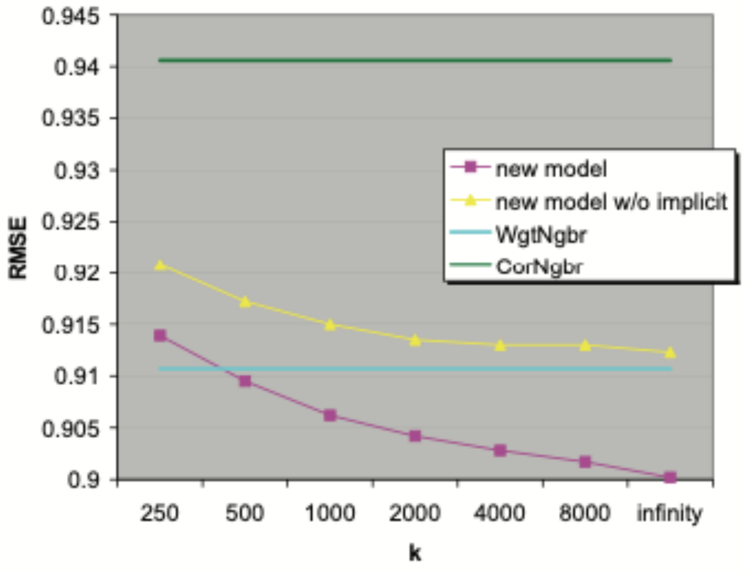


Figure 1: Comparison of neighborhood-based models. We measure the accuracy of the new model with and without implicit feedback. Accuracy is measured by RMSE on the Netflix test set, so lower values indicate better performance. RMSE is shown as a function of varying values of k, which dictates the neighborhood size. For reference, we present the accuracy of two prior models as two horizontal lines: the green line represents a popular method using Pearson correlations, and the cyan line represents a more recent neighborhood model.

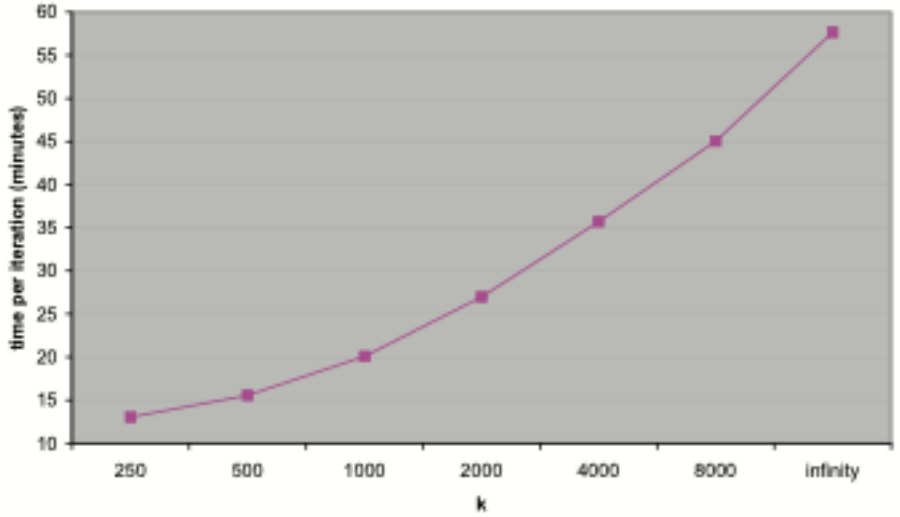


Figure 2: Running times (minutes) per iteration of the neighborhood model, as a function of the parameter k.

Trade-off between RMSE and Running Time

3. Revised Model – Asymmetric-SVD

*Improved model of Paterek's NSVD

$$\hat{r}_{ui} = b_{ui} + q_i^T \left(|\mathrm{R}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{R}(u)} (r_{uj} - b_{uj}) x_j + |\mathrm{N}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{N}(u)} y_j \right)$$

$$\min_{q_*, x_*, y_*, b_*} \sum_{(u,i) \in \mathcal{K}} \left(r_{ui} - \mu - b_u - b_i - q_i^T \left(|\mathrm{R}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{R}(u)} (r_{uj} - b_{uj}) x_j + |\mathrm{N}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{N}(u)} y_j \right) \right)^2 + \lambda_5 \left(b_u^2 + b_i^2 + \|q_i\|^2 + \sum_{j \in \mathrm{R}(u)} \|x_j\|^2 + \sum_{j \in \mathrm{N}(u)} \|y_j\|^2 \right) \tag{14}$$

Benefits
- Fewer parameters
- New users
- Explainability
- Efficient integration of implicit Feedback

3. Revised Model – SVD++

$$\hat{r}_{ui} = b_{ui} + q_i^T \left(p_u + |\mathrm{N}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{N}(u)} y_j \right)$$

Model	50 factors	100 factors	200 factors
SVD	0.9046	0.9025	0.9009
Asymmetric-SVD	0.9037	0.9013	0.9000
SVD++	0.8952	0.8924	0.8911

Table 1: Comparison of SVD-based models: prediction accuracy is measured by RMSE on the Netflix test set for varying number of factors (f). Asymmetric-SVD offers practical advantages over the known SVD model, while slightly improving accuracy. Best accuracy is achieved by SVD++, which directly incorporates implicit feedback into the SVD model.

Outperform

4. Integrated Model

SVD++: Interaction between the user profile and item profile

Baseline: general properties of the item and the user

$$\hat{r}_{ui} = \boxed{\mu + b_u + b_i} + \boxed{q_i^T \left(p_u + |\mathrm{N}(u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{N}(u)} y_j \right)}$$

$$\boxed{+ |\mathrm{R}^k(i;u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{R}^k(i;u)} (r_{uj} - b_{uj}) w_{ij} + |\mathrm{N}^k(i;u)|^{-\frac{1}{2}} \sum_{j \in \mathrm{N}^k(i;u)} c_{ij}}$$

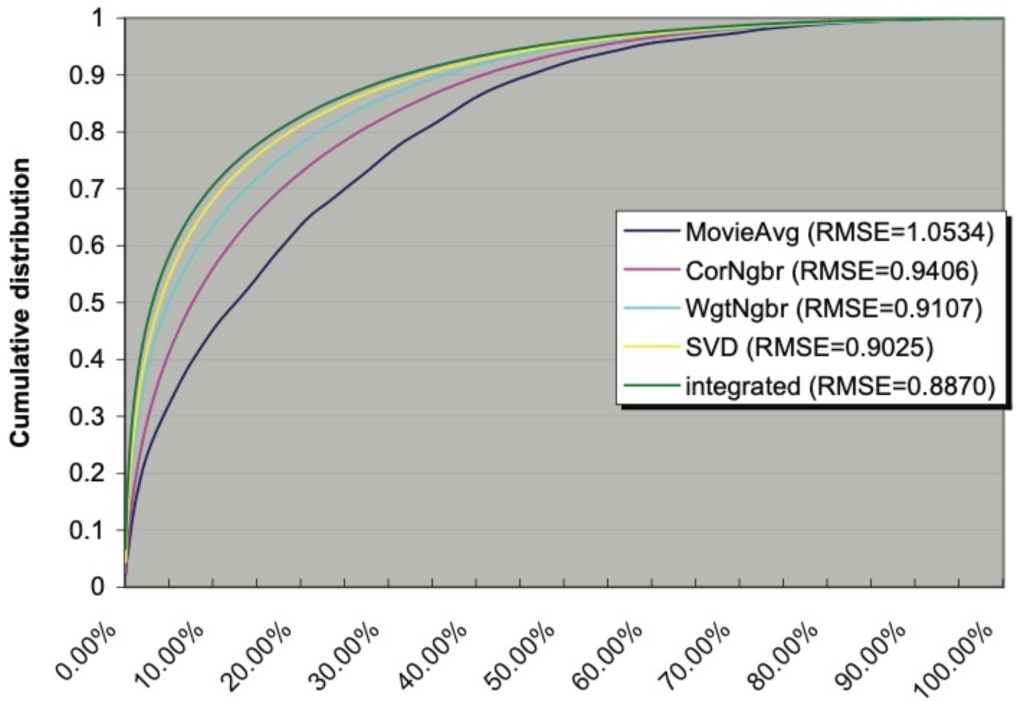
Revised Model: Neighborhood - tier

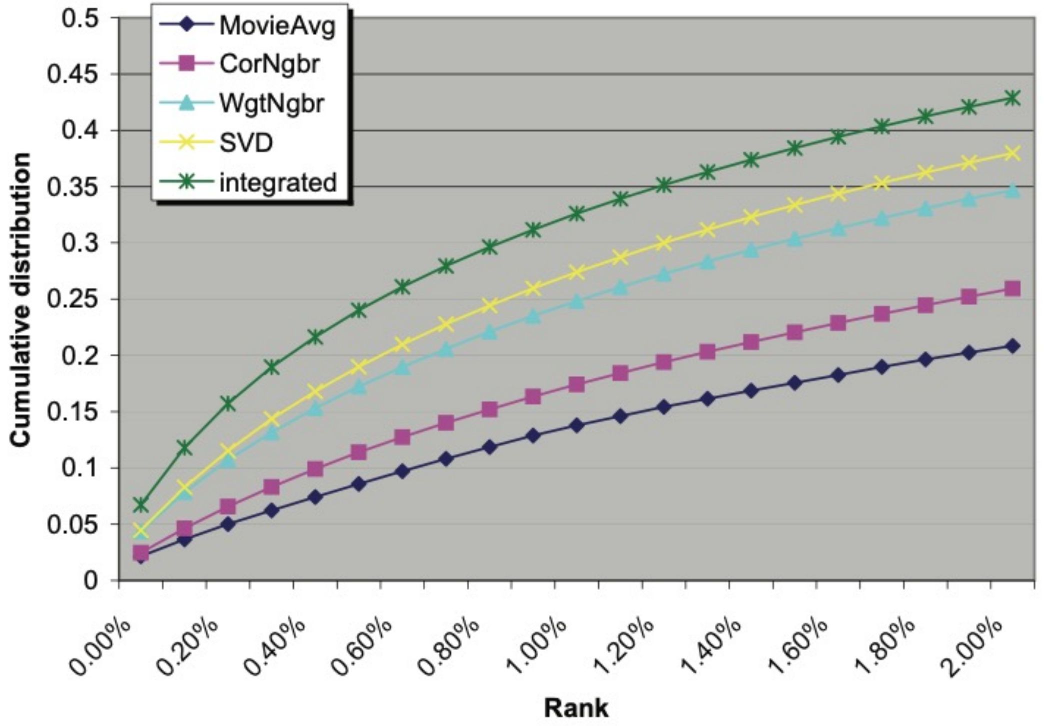
	50 factors	100 factors	200 factors
RMSE	0.8877	0.8870	0.8868
time/iteration	17min	20min	25min

Table 2: Performance of the integrated model. Prediction accuracy is improved by combining the complementing neighborhood and latent factor models. Increasing the number of factors contributes to accuracy, but also adds to running time.

5. Top-K Recommender

- Does achieving a slightly better RMSE result in a completely different and better recommendation?





6. Implementation – Dataset

grouplens about datasets publications blog

MovieLens

GroupLens Research has collected and made available rating data sets from the MovieLens web site (https://movielens.org). The data sets were collected over various periods of time, depending on the size of the set. Before using these data sets, please review their README files for the usage licenses and other details.

Seeking permission? If you are interested in obtaining permission to use MovieLens datasets, please first read the terms of use that are included in the README file. Then, please fill out this form to request use. We typically do not permit public redistribution (see Kaggle for an alternative download location if you are concerned about availability).

recommended for new research

MovieLens 25M Dataset

MovieLens 25M movie ratings. Stable benchmark dataset. 25 million ratings and one million tag applications applied to 62,000 movies by 162,000 users. Includes tag genome data with 15 million relevance scores across 1,129 tags. Released 12/2019

- README.txt
- ml-25m.zip (size: 250 MB, checksum)

Permalink: https://grouplens.org/datasets/movielens/25m/

Datasets

- MovieLens
- WikiLens
- Book-Crossing
- Book Genome Dataset
- Jester
- EachMovie
- Rating Disposition 2023
- Learning from Sets of Items 2019
- Personality 2018
- Serendipity 2018

MovieLens 1M Dataset

```
import random
random.seed(42)
def train_test_split(matrix, ratio):
    true_indices = np.argwhere(matrix)

    num_test = int(len(true_indices) * ratio)
    np.random.shuffle(true_indices)

    test_indices = true_indices[:num_test]
    train_indices = true_indices[num_test:]

    train_dataset, test_dataset = np.zeros_like(matrix), np.zeros_like(matrix

    for i in range(len(test_indices)):
        row_idx, column_idx = test_indices[i]
        test_dataset[row_idx, column_idx] = 1

    for i in range(len(train_indices)):
        row_idx, column_idx = train_indices[i]
        train_dataset[row_idx, column_idx] = 1

    return train_dataset, test_dataset

bin_train_data, bin_test_data = train_test_split(pivot_notna, 0.1)
```

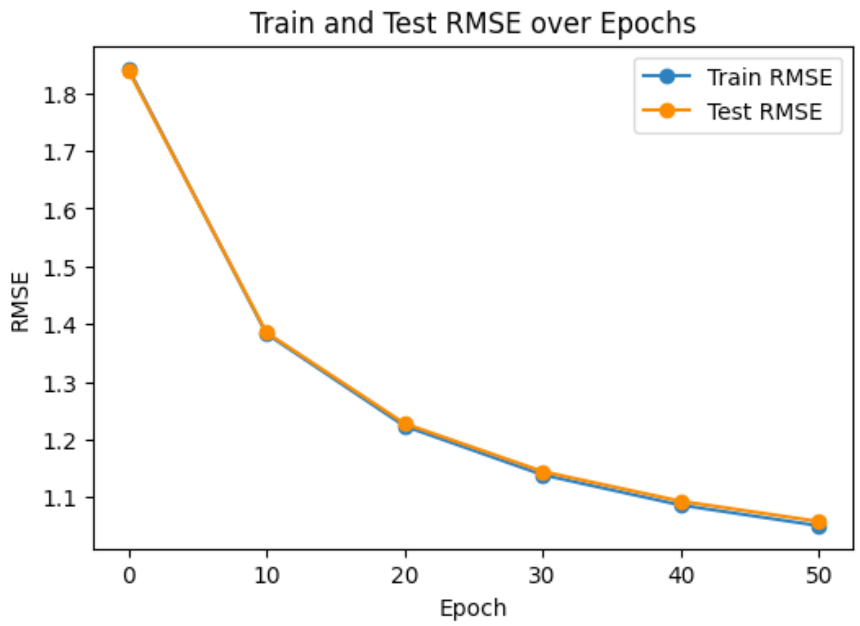
✓ Train-test-split Ratio = 10%

6. Implementation – Baseline

*Code

```python
class Baseline(nn.Module):
    def __init__(self, overall, matrix, n_users, n_movies, lam_u=2e-3):
        super().__init__()
        self.lam_u = lam_u
        self.overall_matrix = overall * torch.ones(n_users, n_movies)
        self.user_bias = nn.Parameter(torch.randn(n_users))
        self.movie_bias = nn.Parameter(torch.randn(n_movies))
        self.matrix = matrix
        self.non_zero_mask = (matrix != -1).type(torch.FloatTensor)

    def forward(self):
        user_bias_matrix = self.user_bias[:, None].expand(n_users, n_movies)
        movie_bias_matrix = self.movie_bias[None, :].expand(n_users, n_movies)
        baseline_matrix = self.overall_matrix + user_bias_matrix +
        movie_bias_matrix

        return baseline_matrix

    def loss(self, prediction):
        diff = (self.matrix - prediction)**2
        prediction_error = torch.sum(diff*self.non_zero_mask)
        l2_reg = self.lam_u * (self.user_bias.norm(p=2) + self.movie_bias.norm
        (p=2))
        total_loss = prediction_error + l2_reg

        return total_loss
```

*Loss



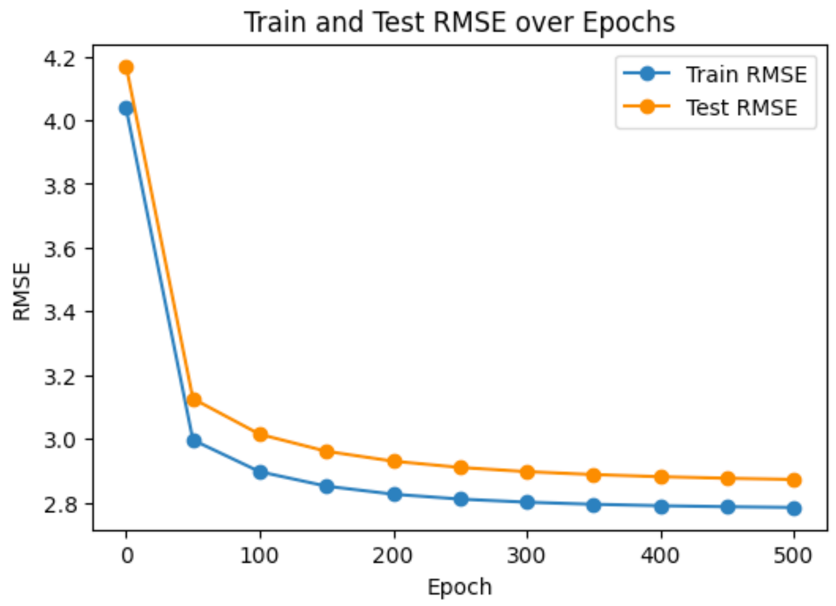
Hyperparameter	Value
Epoch	500
LR	2e-3
RMSE (Test-set)	0.9101
Time	46.3s

6. Implementation – CorNgbr

*Code

```
class Neighborhood(nn.Module):
    def __init__(self, overall, matrix, n_users, n_movies, lam_u=2e-3):
        super().__init__()
        self.baseline = Baseline(overall, matrix, n_users, n_movies)
        self.matrix = matrix
        self.lam_u = lam_u

    def forward(self):
        baseline_matrix = self.baseline()
        diff_matrix = self.matrix - baseline_matrix
        prediction = baseline_matrix + torch.matmul(diff_matrix.double(),
        topk_sigma.T)

        return prediction

    def loss(self, prediction):
        diff = (self.matrix - prediction)**2
        prediction_error = torch.sum(diff*self.baseline.non_zero_mask)
        l2_reg = self.lam_u * (self.baseline.user_bias.norm(p=2) + self.
        baseline.movie_bias.norm(p=2))
        total_loss = prediction_error + l2_reg

        return total_loss
```

*Loss



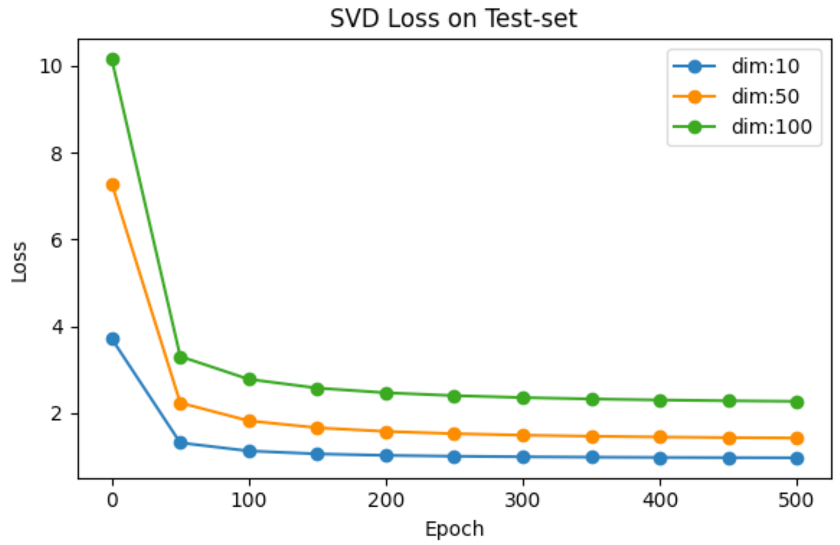
Hyperparameter	Value
Epoch	500
LR	2e-3
RMSE (Test-set)	2.8723
Time	17m 58s

6. Implementation – SVD

*Code

```
class SVD(nn.Module):
    def __init__(self, overall, matrix, n_users, n_movies, dim = 10,  lam_u=0.
    01):
        super().__init__()
        self.lam_u = lam_u
        self.matrix = matrix
        self.baseline = Baseline(overall, matrix, n_users, n_movies)
        self.user_features = nn.Parameter(torch.randn(n_users, dim))
        self.movie_features = nn.Parameter(torch.randn(n_movies, dim))
        self.non_zero_mask = (matrix != -1).type(torch.FloatTensor)

    def forward(self):
        baseline_matrix = self.baseline()
        feature_matrix = torch.matmul(self.user_features, self.movie_features.t
        ())
        prediction = baseline_matrix + feature_matrix

        return prediction

    def loss(self, prediction):
        diff = (self.matrix - prediction)**2
        prediction_error = torch.sum(diff*self.non_zero_mask)
        u_regularization = self.lam_u * (self.baseline.user_bias.norm(p=2) +
        self.baseline.movie_bias.norm(p=2) + torch.sum(self.user_features.norm
        (p=2)) + torch.sum(self.movie_features.norm(p=2)))

        return prediction_error + u_regularization
```

6. Implementation – SVD



Hyperparameter	SVD (dim = 10)	SVD (dim = 50)	SVD (dim = 100)
Epoch	500	500	500
LR	5e-5	5e-5	5e-5
RMSE (Test-set)	0.9639	1.4344	2.2652
Time	1m 3s	1m 8s	1m 12s

# of Parameters go up (relatively low LR & epochs)

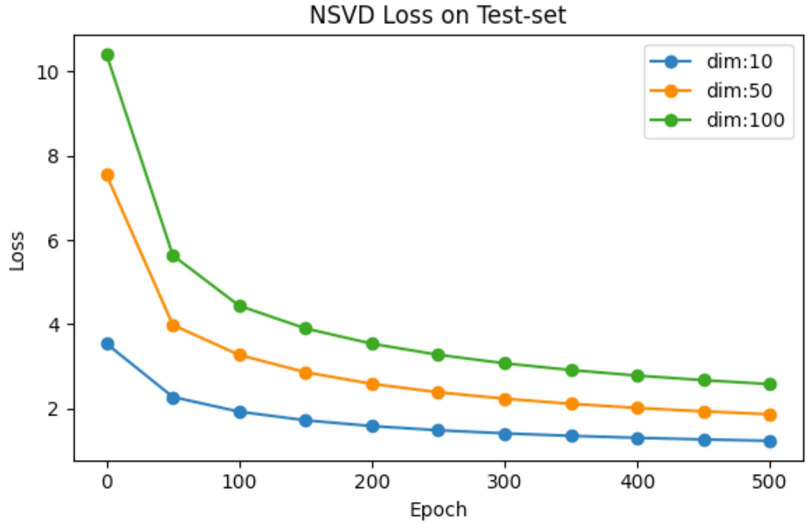
-> Underfitting ..?

6. Implementation – NSVD

*Code

```
class NSVD(nn.Module):
    def __init__(self, overall, matrix, n_users, n_movies, dim = 10, lam_u=0.01)
    :
        super().__init__()
        self.lam_u = lam_u
        self.matrix = matrix
        self.baseline = Baseline(overall, matrix, n_users, n_movies)
        self.movie_weight = nn.Parameter(torch.randn(n_movies, dim))
        self.movie_features = nn.Parameter(torch.randn(n_movies, dim))
        self.non_zero_mask = (matrix != -1).type(torch.FloatTensor)
        self.non_zero_sum = torch.sum(self.non_zero_mask, 1).unsqueeze(1)

    def forward(self, except_bl = False):
        baseline_matrix = self.baseline()
        user_rated_matrix = torch.matmul(self.non_zero_mask/self.non_zero_sum**
        (1/2), self.movie_weight)
        feature_matrix = torch.matmul(user_rated_matrix, self.movie_features.T)

        if except_bl :
            return feature_matrix

        prediction = baseline_matrix + feature_matrix
        return prediction

    def loss(self, prediction):
        diff = (self.matrix - prediction)**2
        prediction_error = torch.sum(diff*self.non_zero_mask)
        u_regularization = self.lam_u * (self.baseline.user_bias.norm(p=2) +
        self.baseline.movie_bias.norm(p=2) + torch.sum(self.movie_weight.norm
        (p=2)) + torch.sum(self.movie_features.norm(p=2)))

        return prediction_error + u_regularization
```

6. Implementation – NSVD



Hyperparameter	NSVD (dim = 10)	NSVD (dim = 50)	NSVD (dim = 100)
Epoch	500	500	500
LR	5e-6	5e-6	5e-6
RMSE (Test-set)	1.2396	1.8688	2.5841
Time	1m 24s	1m 42s	1m 31s

# of Parameters go up (relatively low LR & epochs)

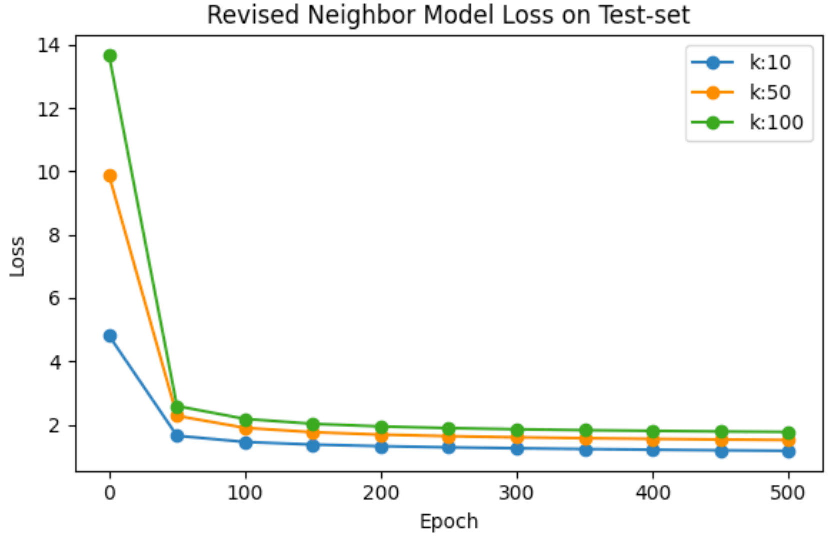
-> Underfitting ..?

6. Implementation – Revised-Ngbr

*Code

```
class Improved_Neighborhood(nn.Module):
    def __init__(self, overall, matrix, n_users, n_movies, pearson_coefficient,
    top_k=10, lam_u=0.01):
        super().__init__()
        self.matrix = matrix
        self.lam_u = lam_u
        self.baseline = Baseline(overall, matrix, n_users, n_movies)
        self.top_k = top_k
        self.pearson_coefficient = pearson_coefficient
        self.shrunk_coeff = get_shrunk_coefficient(self.pearson_coefficient,
        get_rated_both_items(matrix), 100) # default value
        self.topk_matrix, self.topk_sigma = get_neighborhood(self.shrunk_coeff,
        top_k)
        self.non_zero_mask = (matrix != -1).type(torch.FloatTensor)
        self.item_weight = nn.Parameter(torch.rand(n_movies, n_movies))
        self.item_implicit = nn.Parameter(torch.rand(n_movies, n_movies))

    def forward(self, except_bl = False):
        baseline_matrix = self.baseline()
        non_zero_sum = torch.sum(self.topk_matrix, 1).unsqueeze(1)

        diff_matrix = self.matrix - baseline_matrix # r_uj - b_uj
        feature_matrix = torch.matmul(diff_matrix, torch.mul(self.topk_matrix /
        non_zero_sum**(1/2), self.item_weight).T )  + torch.matmul(torch.ones
        (n_users, n_movies), torch.mul(self.topk_matrix/ non_zero_sum ** (1/2),
        self.item_implicit).T)

        if except_bl :
            return feature_matrix

        prediction = baseline_matrix + feature_matrix

        return prediction

    def loss(self, prediction):
        diff = (self.matrix - prediction)**2
        prediction_error = torch.sum(diff*self.non_zero_mask)
        l2_reg = self.lam_u * (self.baseline.user_bias.norm(p=2) + self.
        baseline.movie_bias.norm(p=2) + self.item_implicit.norm(p=2) + self.
        item_weight.norm(p=2))
        total_loss = prediction_error + l2_reg

        return total_loss
```

6. Implementation – Revised-Ngbr



Hyperparameter	Ngbr (K = 10)	Ngbr (K = 50)	Ngbr (K = 100)
Epoch	500	500	500
LR	5e-6	5e-6	5e-6
RMSE (Test-set)	1.1703	1.5108	1.7663
Time	10m 40s	11m 1s	12m 59s

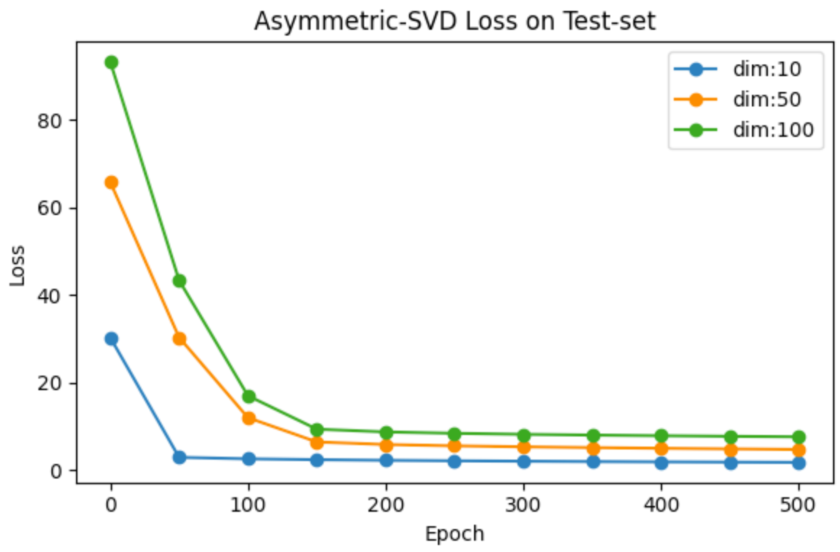
Better than the Neighborhood Model but …

6. Implementation – Asymmetric-SVD

*Code

```
class AsymmetricSVD(nn.Module):
    def __init__(self, overall, matrix, n_users, n_movies, dim=10, lam_u=0.01):
        super().__init__()
        self.lam_u = lam_u
        self.matrix = matrix
        self.baseline = Baseline(overall, matrix, n_users, n_movies)
        self.none_zero_mask = (self.matrix != -1).type(torch.FloatTensor)
        self.non_zero_sum = torch.sum(self.none_zero_mask, 1).unsqueeze(1)
        self.movie_weight = nn.Parameter(torch.randn(n_movies, dim))
        self.movie_features = nn.Parameter(torch.randn(n_movies, dim))
        self.implicit_feedback = nn.Parameter(torch.rand(n_movies, dim))

    def forward(self, except_bl = False):
        baseline_matrix = baseline()

        user_rated_matrix = torch.matmul(self.none_zero_mask, self.
        movie_weight) / self.non_zero_sum**(1/2) + torch.matmul(self.
        none_zero_mask, self.implicit_feedback) / self.non_zero_sum**(1/2)

        feature_matrix = torch.matmul(user_rated_matrix, self.movie_features.T)

        if except_bl:
            return feature_matrix

        prediction = baseline_matrix + feature_matrix
        return prediction

    def loss(self, prediction):
        diff = (self.matrix - prediction)**2
        prediction_error = torch.sum(diff*self.none_zero_mask)

        u_regularization = self.lam_u * (torch.sum(self.baseline.user_bias.norm
        (p=2)) + torch.sum(self.baseline.movie_bias.norm(p=2)) + torch.sum(self.
        movie_weight.norm(p=2)) + torch.sum(self.movie_features.norm(p=2)) +
        torch.sum(self.implicit_feedback.norm(p=2)))

        return prediction_error + u_regularization
```

6. Implementation – Asymmetric-SVD



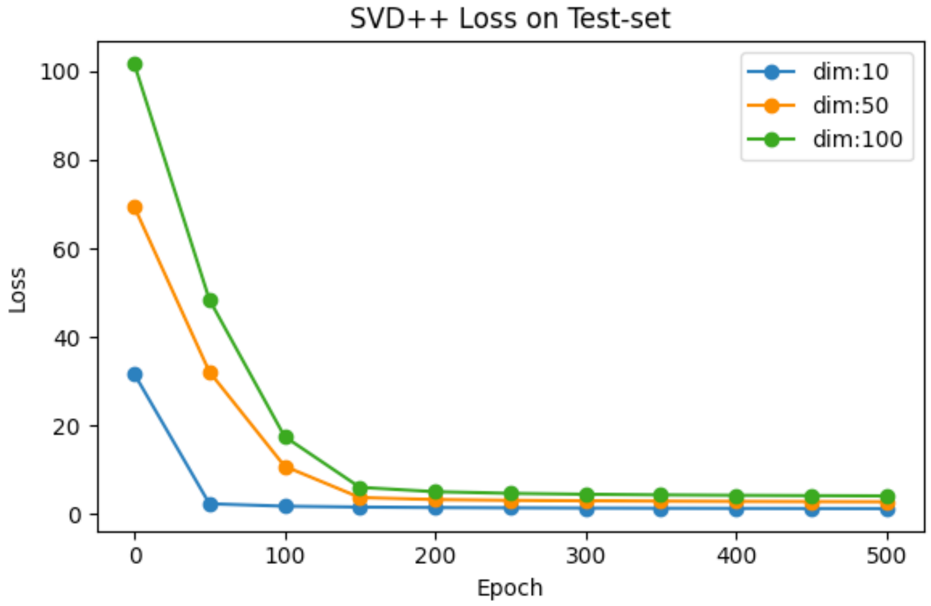
Hyperparameter	ASVD (dim = 10)	ASVD (dim = 50)	ASVD (dim = 100)
Epoch	500	500	500
LR	5e-7	5e-7	5e-7
RMSE (Test-set)	1.7786	4.7301	7.6140
Time	2m 23s	3m 9s	2m 15s

6. Implementation – SVD++

*Code

```
class SVDdoublePlus(nn.Module):
    def __init__(self, overall, matrix, n_users, n_movies, dim=10, lam_u=0.01):
        super().__init__()
        self.lam_u = lam_u
        self.matrix = matrix
        self.baseline = Baseline(overall, matrix, n_users, n_movies)
        self.none_zero_mask = (self.matrix != -1).type(torch.FloatTensor)
        self.non_zero_sum = torch.sum(self.none_zero_mask, 1).unsqueeze(1)
        self.user_features = nn.Parameter(torch.rand(n_users, dim))
        self.movie_features = nn.Parameter(torch.randn(n_movies, dim))
        self.implicit_feedback = nn.Parameter(torch.rand(n_movies, dim))

    def forward(self, except_bl = False):
        baseline_matrix = baseline()

        user_rated_matrix =  torch.matmul(self.none_zero_mask, self.
        implicit_feedback) / self.non_zero_sum**(1/2) + self.user_features

        feature_matrix = torch.matmul(user_rated_matrix, self.movie_features.T)

        if except_bl:
            return feature_matrix

        prediction = baseline_matrix + feature_matrix
        return prediction

    def loss(self, prediction):
        diff = (self.matrix - prediction)**2
        prediction_error = torch.sum(diff*self.none_zero_mask)

        u_regularization = self.lam_u * (torch.sum(self.baseline.user_bias.norm
        (p=2)) + torch.sum(self.baseline.movie_bias.norm(p=2)) + torch.sum(self.
        user_features.norm(p=2)) + torch.sum(self.movie_features.norm(p=2)) +
        torch.sum(self.implicit_feedback.norm(p=2)))

        return prediction_error + u_regularization
```

6. Implementation – SVD++



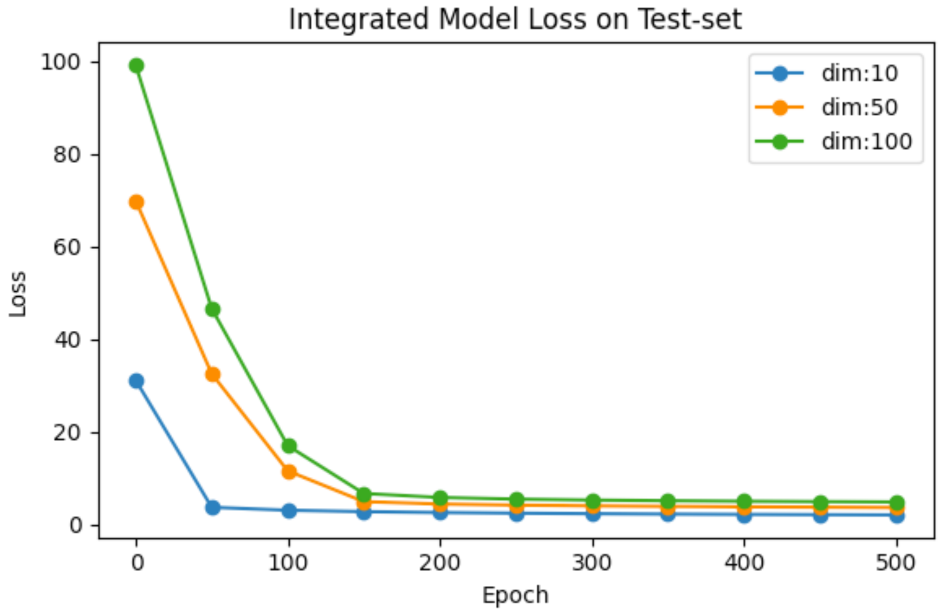
Hyperparameter	SVD++ (dim = 10)	SVD++ (dim = 50)	SVD++ (dim = 100)
Epoch	500	500	500
LR	5e-7	5e-7	5e-7
RMSE (Test-set)	1.3083	2.8427	4.1661
Time	2m 29s	1m 58s	2m 20s

6. Implementation – Integrated Model

*Code

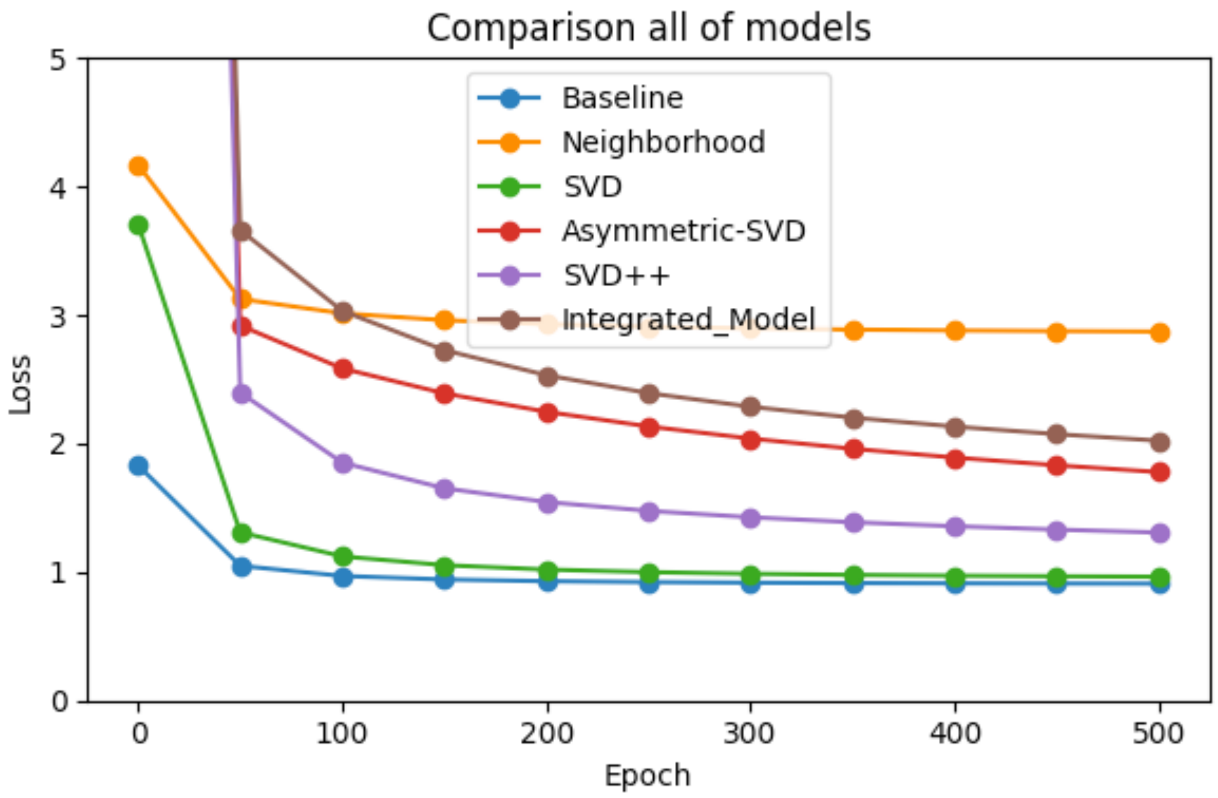
```
class Integrated_Model(nn.Module):
    def __init__(self, overall, matrix, n_users, n_movies, pearson_coefficient,
    top_k=10, lam_u=0.01):
        super().__init__()
        self.matrix = matrix
        self.lam_u = lam_u
        self.revised_ngbr = Improved_Neighborhood(overall, matrix, n_users,
        n_movies, pearson_coefficient)
        self.baseline = Baseline(overall, matrix, n_users, n_movies)
        self.svd_double_plus = SVDdoublePlus(overall, matrix, n_users, n_movies)


    def forward(self):
        baseline_matrix = self.baseline()
        svd_matrix = self.svd_double_plus(except_bl = True)
        ngbr_matrix = self.revised_ngbr(except_bl = True)

        return baseline_matrix + svd_matrix + ngbr_matrix

    def loss(self, prediction):
        diff = (self.matrix - prediction)**2
        prediction_error = torch.sum(diff*self.svd_double_plus.none_zero_mask)

        u_regularization (variable) user_features: Parameter ine.user_bias.norm
        (p=2)) + torch.s ) + torch.sum(self.
        svd_double_plus.user_features.norm(p=2)) + torch.sum(self.
        svd_double_plus.movie_features.norm(p=2)) + torch.sum(self.
        svd_double_plus.implicit_feedback.norm(p=2)) + torch.sum(self.
        revised_ngbr.item_implicit.norm(p=2)) + torch.sum(self.revised_ngbr.
        item_weight.norm(p=2)) )
        return prediction_error + u_regularization
```

6. Implementation – Integrated Model



Hyperparameter	Itg (dim = 10)	Itg (dim = 50)	Itg (dim = 100)
Epoch	500	500	500
LR	5e-7	5e-7	5e-7
RMSE (Test-set)	2.0229	3.6158	4.8226
Time	14m 28s	14m 00s	19m 18s

6. Implementation – Comparison all of models



6. Implementation – Summary

***잘한 점**

1. 오픈소스 안 보고 혼자 모델을 구현하려고 노력 ..

2. 여러가지 실험 진행하여 결과값 정리한 것 ..?

***아쉬운 점**

1. Regularizing Term 때문에 Gradient Exploding이 너무 심하게 발생

2. 작은 값의 Learning-rate와 Gradient Clipping으로 어찌저찌 막기는 했으나 Dimension이 올라갈수록 Underfitting 되는 경향성이 심해짐

3.다른 Optimizer를 사용하거나 Epoch을 증가시켰어야 했는데 시간 관계상 ..

4.물론 데이터셋이 다르지만 결과값에 대한 재현성 낮음

5.Top-K Recommender에 대한 실험 X

Thank You