Collaborative Filtering for Implicit Feedback Datasets

Y. Hu, Y. Koren and C. Volinsky

Paper review

2022.07.05

이상민

목차

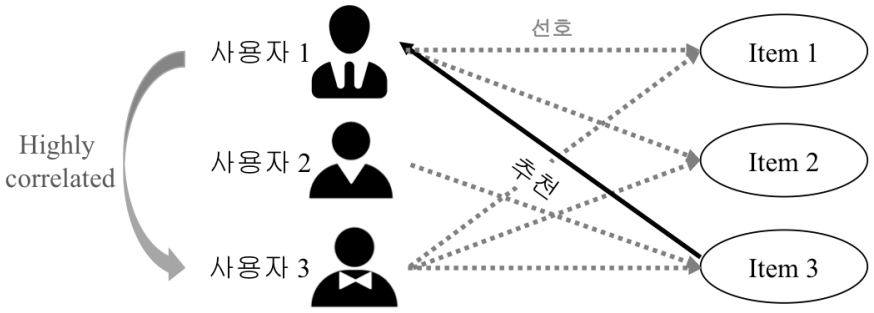
Recommender system

1. Content Based Filtering

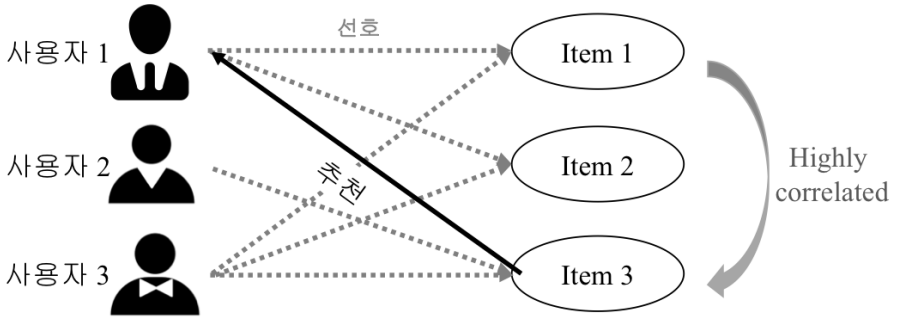
- User와 product의 특성을 나타내는 profile 생성
- 생성된 profile을 기반으로 match 시켜 줌
- Profile을 생성하기 위한 추가적인 정보들을 얻기 어렵거나 불가능할 수 있음

2. Collaborative Filtering

- User들의 past behavior만을 필요로 함
- User-item association을 식별하기 위해 user 및 item 간의 관계를 분석
- 구매 이력이 부족한 user 또는 item에 추천을 제공하기 어려움 (cold start problem)



User-oriented



Item-oriented

User feedback의 종류

1. Explicit Feedback

- 별점, 좋아요와 같이 user의 선호 여부를 정확히 파악할 수 있음
- High quality, but hard to gather

2. Implicit Feedback

- User의 행동을 관찰함으로써 preference를 추론
- 클릭, 구매 기록, 검색 기록, …

Implicit Feedback의 특성

1) **No negative feedback :**
   ⇒ negative feedback이 포함될 가능성이 있는 missing value도 고려
2) **Inherently noisy :** 해당 item을 소비했다고 해서 선호한다고 확신할 수 없음
3) **Numerical value of implicit feedback indicates confidence**
   ⇒ 자주 구매했다고 preference가 증가하는 것이 아니라, confidence가 증가함
4) **Requires appropriate measures to evaluate**

Neighborhood models

일반적으로 user-oriented보다 item-oriented model의 scalability와 accuracy가 더 높음

Item-oriented model에서는 item간의 similarity를 측정한 후 가장 유사한 k개의 item들의 observed value로부터 unobserved value 추정

Item-ID / User-ID	1	2	3	4	5
1	?	4	3	1	5
2	0	3	0	3	?
3	2	2	7	8	2
4	3	4	4	?	7
5	4	2	2	2	9
pearson(1,i)	1	-0.1529	0.3105	-0.1555	0.9707

Explicit feedback의 경우 user와 item마다 rating의 평균이 다른 것을 제거해줌

Implicit feedback의 경우 scale이 천차만별이기 때문에 bias를 제거해주는 것의 효과가 크지 않음

User preference와 confidence를 독립적으로 고려할 수 없음

Neighborhood models

일반적으로 user-oriented보다 item-oriented model의 scalability와 accuracy가 더 높음

Item-oriented model에서는 item간의 similarity를 측정한 후 가장 유사한 k개의 item들의 observed value로부터 unobserved value 추정

Item-ID / User-ID	1	2	3	4	5
1	?	4	3	1	5
2	0	3	0	3	?
3	2	2	7	8	2
4	3	4	4	?	7
5	4	2	2	2	9
pearson(1,i)	1	-0.1529	0.3105	-0.1555	0.9707

$$\hat{r}_{1,1} = \frac{\sum_{j \in S^2(1;1)} s_{1j} r_{1j}}{\sum_{j \in S^2(1;1)} s_{1j}}$$

$$= \frac{0.3105 \cdot 3 + 0.9707 \cdot 5}{0.3105 + 0.9707}$$

$$= 4.51$$

Explicit feedback의 경우 user와 item마다 rating의 평균이 다른 것을 제거해줌

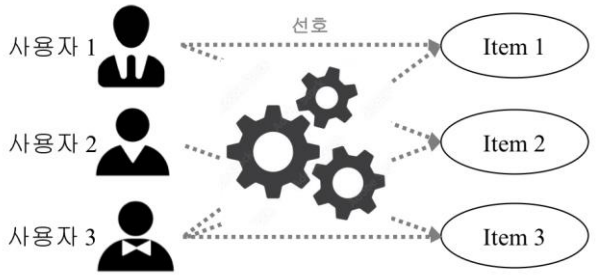
Implicit feedback의 경우 scale이 천차만별이기 때문에 bias를 제거해주는 것의 효과가 크지 않음

User preference와 confidence를 독립적으로 고려할 수 없음

Latent factor models

Observed ratings를 설명해주는 latent feature들을 찾아내는 것을 목표로 함

Singular Value Decomposition



- Accuracy와 scalability가 높음
- User-factors vector $x_u \in \Re^f$, item-factors vector $y_i \in \Re^f$
- 두 벡터의 내적으로 추정 $\hat{r}_{ui} = x_u^T y_i$
- Explicit feedback dataset에 적용할 경우 overfit을 방지하기 위해 adequate regularized model 사용

$$\min_{x_\star, y_\star} \sum_{r_{u,i} is\ known} (r_{ui} - x_u^T y_i)^2 + \lambda(\|x_u\|^2 + \|y_i\|^2)$$

- Netflix dataset에서 neighborhood methods보다 성능이 좋음

Preliminaries & Terms

User: u, v Item: i, j input data associate users and items: r_{ui}

Implicit dataset에서, input data는 observations for user actions를 나타냄

Ex) 구매한 횟수, 웹페이지에 머무른 시간, 시청 횟수, …

Observation이 없다면, missing value가 아닌 0

Observation r_{ui}를 preference p_{ui}와 confidence c_{ui}로 transform

- 한번이라도 소비했으면 preference가 있고, 아닌 경우 no-preference

$$p_{ui} = \begin{cases} 1, & r_{ui} > 0 \\ 0, & r_{ui} = 0 \end{cases}$$

- 일반적으로 r_{ui}가 커질수록 preference에 대한 indication이 강해짐

$$c_{ui} = 1 + \alpha r_{ui}, \qquad \alpha = 40$$

Modeling Overview

Goal

User preference를 결정하는 벡터 $x_u \in \Re^f, y_i \in \Re^f$를 결정

Preference를 두 벡터의 inner product로 표현 $p_{ui} = x_u^T y_i$

Method

SVD와 유사하지만,

1) Confidence level에 대한 정보를 고려해야 하고

2) Observed data가 아닌 모든 user-item pairs에 대한 optimization 필요함

$$\min_{x_\star, y_\star} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

Computational Complexity Issue

Issue

Observed data가 아닌 모든 user-item pairs에 대한 optimization 필요함

Sparse dataset에서, user-item pairs는 많게는 수십억 개에 달함

$$\min_{x_\star, y_\star} \sum_{r_{u,i} is\ known} (r_{ui} - x_u^T y_i)^2 + \lambda(\|x_u\|^2 + \|y_i\|^2)$$

$$\min_{x_\star, y_\star} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

Computational Complexity Issue

Issue

Observed data가 아닌 모든 user-item pairs에 대한 optimization 필요함

Sparse dataset에서, user-item pairs는 많게는 수십억 개에 달함

$$\min_{x_\star, y_\star} \sum_{r_{u,i} is\ known} (r_{ui} - x_u^T y_i)^2 + \lambda(\|x_u\|^2 + \|y_i\|^2)$$

Explicit dataset에서 주로 사용되는 SGD와 같은
direct optimization technique들을 활용하기 어려움

$$\min_{x_\star, y_\star} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

Alternating Least Squares(ALS)

$$\min_{x_\star, y_\star} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

User-factors Matrix $X \in \Re^{m \times f}$, Item-factors Matrix $Y \in \Re^{m \times f}$를 번갈아가면서 re-computing

X와 Y중 하나를 고정하면, 목적함수가 quadratic form이 됨

⇒ 볼록함수의 minimize 문제가 되어서, 반복 단계마다 cost function이 더 낮은 값을 갖도록 보장

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u)$$ confidence: $C_{ii}^u = c_{ui}$, preference: $p(u) \in \Re^n$

$$y_i = \left(X^T C^i X + \lambda I\right)^{-1} X^T C^i p(i)$$ confidence: $C_{uu}^i = c_{ui}$, preference: $p(i) \in \Re^m$

Avoiding Computational Bottleneck

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u), \quad u = 1, \cdots, m$$

Point 1 $Y^T C^u Y = Y^T Y + Y^T (C^u - I) Y^T$

Point 2 $Y^T C^u p(u) = \{Y^T (C^u - I) + Y^T\} p(u)$

$c_{ui} = 1 + \alpha r_{ui}$ ⇒ C^u에는 nonzero element가 n개 있지만 $C^u - I$에는 n_u개만 존재

$Y^T C^u Y$의 require time $O(f^2 n)$ ⇒ $Y^T (C^u - I) Y$의 require time $O(f^2 n_u)$, typically $n_u \ll n$

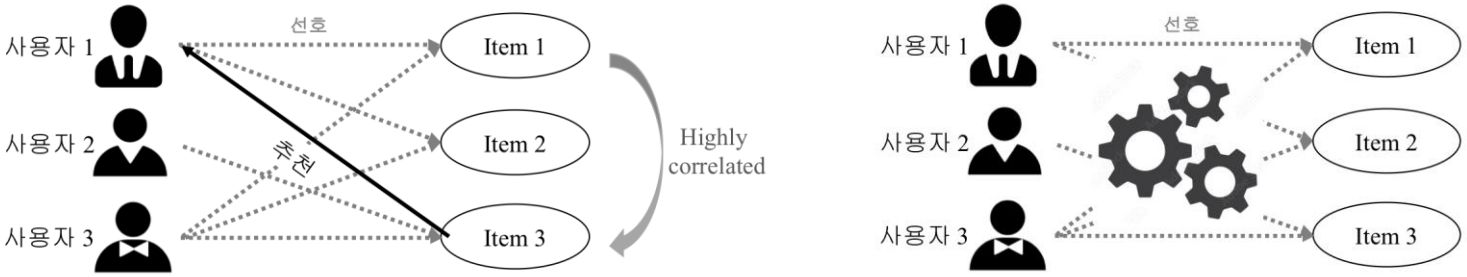
마찬가지로, $Y^T (C^u - I) p(u)$의 require time 역시 n_u에 linear : $O(f n_u)$

$(Y^T C^u Y + \lambda I)^{-1}$의 computing time은 $O(f^3)$이므로, x_u의 computing time은 $O(f^2 n_u + f^3)$

따라서 matrix X를 update하기 위해서는 $O(f^2 N + f^3 m)$의 computing time이 필요함

같은 방법으로, matrix Y는 $O(f^2 N + f^3 n)$ 이때, N은 non-zero observation의 수

Explaining recommendations



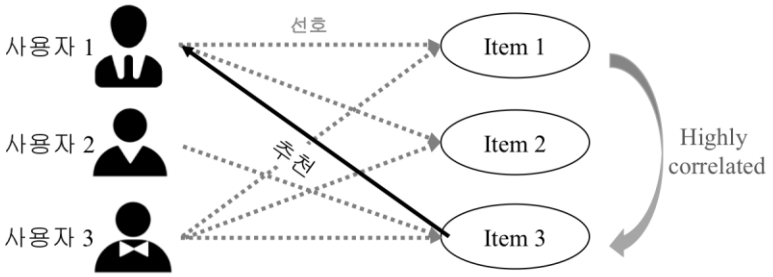
좋은 recommendation은 왜 추천했는지에 대한 이유가 수반되어야 함

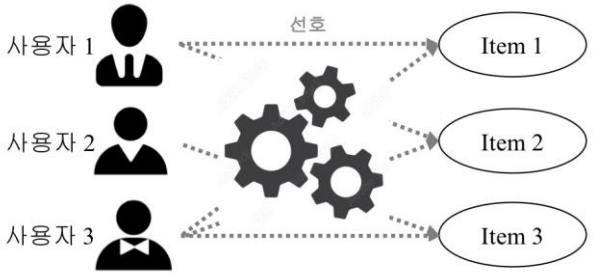
User의 past behavior로부터 추론되어 직관적으로 이해할 수 있는 neighborhood model과 달리,

Latent factor model은 latent factor들로 추상화되어 direct relation을 파악하기 어려움

$$\hat{p}_{ui} = y_i^T x_u = y_i (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u),$$

Explaining recommendations





좋은 recommendation은 왜 추천했는지에 대한 이유가 수반되어야 함

User의 past behavior로부터 추론되어 직관적으로 이해할 수 있는 neighborhood model과 달리,

Latent factor model은 latent factor들로 추상화되어 direct relation을 파악하기 어려움

$$\hat{p}_{ui} = y_i^T x_u = y_i \underbrace{(Y^T C^u Y + \lambda I)^{-1}}_{W^u \in f \times f} Y^T C^u p(u),$$

$$s_{ij}^u = y_i W^u y_j^T, \qquad \hat{p}_{ui} = \sum_{j:r_{uj}>0} s_{ij}^u c_{uj}$$

각 past action들의 contribution을 더해줌

1. **Computed predictions를 설명할 수 있음**
   ⇒ **Preprocessor로 활용 가능**
2. **User마다 다른 item의 similarity 설명 가능**

Data description

In Paper Digital television의 channel tune event data

i: TV program 17,000개

4주간의 training period 후 1주간의 data로 test

test set에서는 $r_{ui} < 0.5$일 경우 0으로 toggle

Training period 동안 시청한 'easy prediction'은 제외

u: Set top box 300,000개

r_{ui}: 프로그램 시청 횟수 32million nonzero values

같은 프로그램을 반복해서 보는 경향을 고려해 log scale 도입

⇒ $c_{ui} = 1 + \alpha \log\left(1 + \frac{r_{ui}}{\epsilon}\right)$으로 계산. $\epsilon = 10^{-8}$

같은 채널을 계속 틀어 놓는 momentum effect를 제거하기 위해 down-weight

⇒ 같은 channel 의 t 번째 show에 $\frac{e^{-(at-b)}}{1+e^{-(at-b)}}$만큼 weight. $a = 2, b = 6$

Evaluation Methodology

각 user에 대한 item들의 predicted preference 순위를 나열한 시나리오를 평가

Confidential nonnegative feedback 이 없고, 사용자 반응을 추적할 수 없음

⇒ Precision보다 Recall 기반의 measure를 적용

$rank_{ui}$: percentile-ranking of program i for user u.

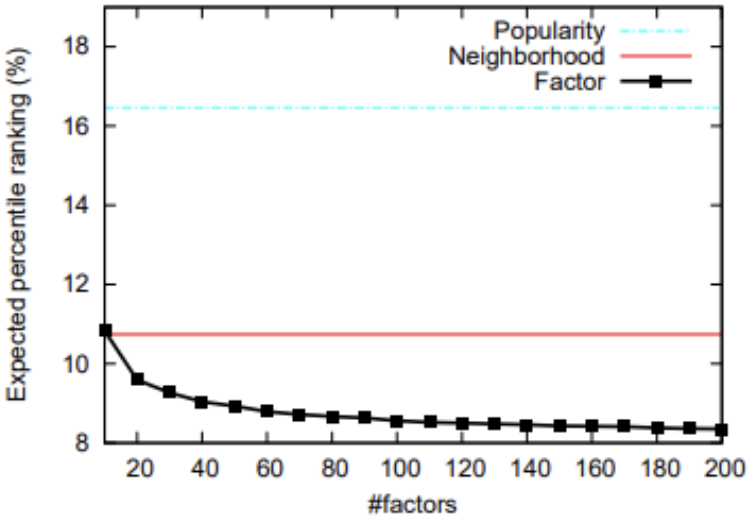
Test period 동안 시청 데이터의 Expected percentile ranking으로 evaluate

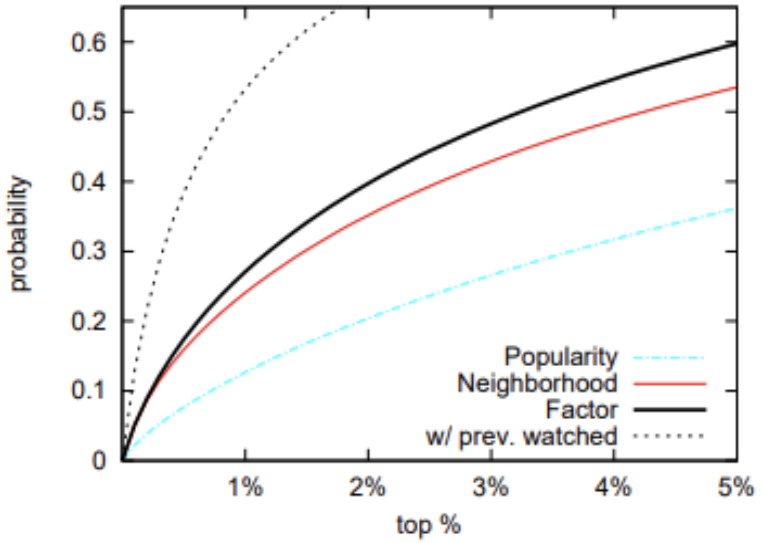
$$\overline{rank} = \frac{\sum_{u,i} r_{ui}^{t} rank_{ui}}{\sum_{u,i} r_{ui}^{t}}$$

$\overline{rank}$가 낮을수록 실제 시청한 프로그램의 순위가 높은 것

무작위 예측의 경우 $\overline{rank}$의 기댓값은 50%

Evaluation results





Evaluation results 1

다른 두 competing model보다 좋은 성능을 보임

Number of factors가 증가함에 따라 예측 성능도 높아짐

Feasible한 범위 내에서 dimension을 최대한 크게 설정

Evaluation results 2

추천한 프로그램을 볼 확률이 가장 높음

Ex) 상위 1% 추천 프로그램을 시청할 확률 27%

'easy prediction'을 포함하면 성능이 더 높아질 것

Easy prediction은 remind의 효과가 있음

Importance of considering confidence levels

Original Model

$$\min_{x_\star, y_\star} \sum_{u,i} c_{ui}(p_{ui} - x_u^T y_i)^2 + \lambda \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

Variation 1 Raw observation을 그대로 사용

$$\min_{x_\star, y_\star} \sum_{u,i} (r_{ui} - x_u^T y_i)^2 + \lambda_1 \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

Dense SVD algorithm의 regularized version

Regularize가 없으면 popularity model보다 성능이 떨어짐

$\lambda_1 = 500$일 때 성능이 개선되지만 여전히 neighborhood model보다는 떨어짐

Variation 2 Binary preference만 적용

$$\min_{x_\star, y_\star} \sum_{u,i} (p_{ui} - x_u^T y_i)^2 + \lambda_2 \left(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2 \right)$$

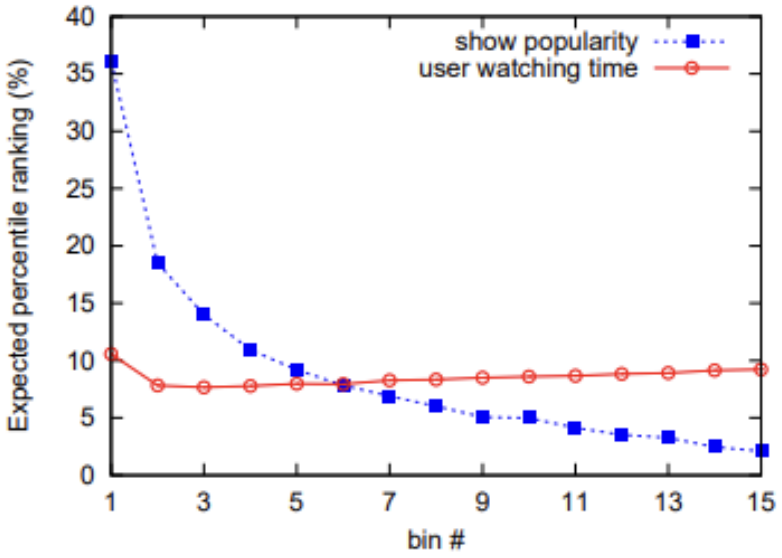
$\lambda_2 = 150$으로 regularize

Variation 1보다 성능이 좋음

Factor가 50개일 때 neighborhood model보다 나아짐

하지만 여전히 Original model보다는 성능이 떨어짐

Analyze Performance by segregating



Items

인기가 적은 순으로 test set을 15개의 equal bins로 분류

인기가 많은 tv show일수록 performance가 증가

따라서 인기가 많은 item을 추천해주는 것이 안전함

Users

거의 시청하지 않는 1번 bin을 제외하고는 효과가 일정

여러 사람이 같은 TV를 시청하기 때문

Data description

In Paper Digital television의 channel tune event data

i: TV program 17,000개

4주간의 training period 후 1주간의 data로 test

test set에서는 $r_{ui} < 0.5$일 경우 0으로 toggle

Training period 동안 시청한 'easy prediction'은 제외

u: Set top box 300,000개

r_{ui}: 프로그램 시청 횟수 32million nonzero values

같은 프로그램을 반복해서 보는 경향을 고려해 log scale 도입

⇒ $c_{ui} = 1 + \alpha \log\left(1 + \frac{r_{ui}}{\epsilon}\right)$으로 계산. $\epsilon = 10^{-8}$

같은 채널을 계속 틀어 놓는 momentum effect를 제거하기 위해 down-weight

⇒ 같은 channel 의 t 번째 show에 $\frac{e^{-(at-b)}}{1+e^{-(at-b)}}$만큼 weight. $a = 2, b = 6$

Data description

My implement Animation episode 시청 데이터

23일간의 데이터를 랜덤하게 train set과 test set으로 분리

따로 처리하지 않아도 'easy prediction'은 존재하지 않음

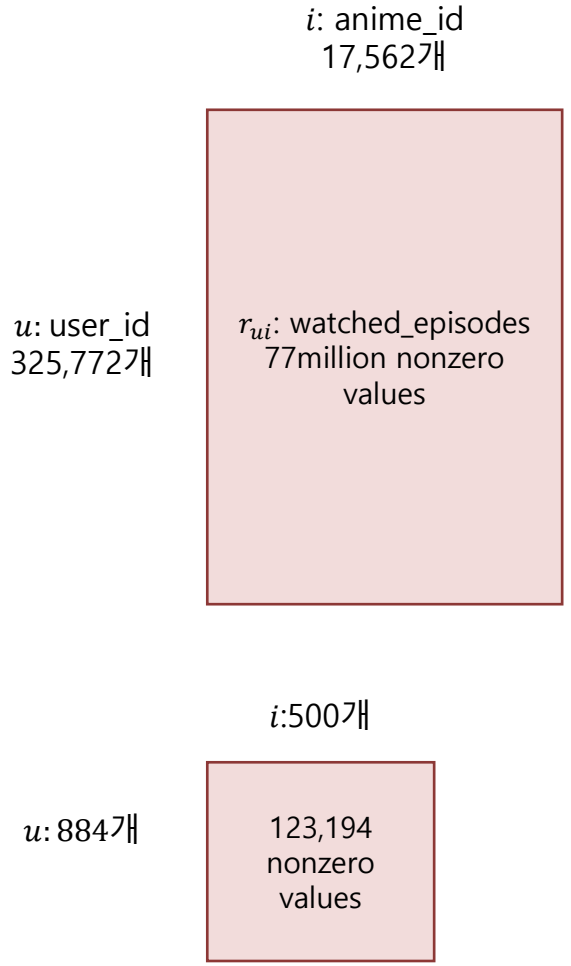
에피소드 시청 횟수 데이터도 범위가 넓음

$\Rightarrow c_{ui} = 1 + \alpha \log\left(1 + \frac{r_{ui}}{\epsilon}\right)$으로 계산. $\epsilon = 10^{-8}$

Momentum effect 는 고려하지 않음

계산의 편의성을 위해 데이터를 축소

- 시청한 user 수가 많은 상위 500개의 애니메이션
- Random한 1000명의 user 중 시청한 애니메이션이 5개 이상인 884명의 user



Modeling with SciPy sparse.csr_matrix

```
observations = np.array(train['watched_episodes'])
row = np.array(train['user_id'])
col = np.array(train['anime_id'])
train_matrix = sparse.csr_matrix((observations, (row,col)), dtype=int)
```

```
class ALS():
    def __init__(self, train, test, K, alpha, epsilon, epochs, regularize):
        self.K = K
        self.alpha = alpha  40
        self.epsilon = epsilon  10^-8
        self.epochs = epochs  10
        self.regularize = regularize

        self.R = train
        self.P = sparse.csr_matrix(train>0, dtype=np.float16)
        self.user_num, self.item_num = train.shape
        self.test = test

        self.C_I = (self.R/self.epsilon +self.P).log1p()   α log(1 + r_ui/ϵ)

        self.X = np.random.normal(scale=1./self.K, size=(self.user_num, self.K))
        self.Y = np.random.normal(scale=1./self.K, size=(self.item_num, self.K))
```

Updating Factor Matrix

```python
def fit(self):
    self.score = []

    for t in range(self.epochs):
        for u in range(self.user_num):
            YTY = np.dot(self.Y.T, self.Y)
            self.X[u,:] = self.update_X(u, YTY) # YTY를 반복 단계 전에 계산

        for i in range(self.item_num):
            XTX = np.dot(self.X.T, self.X)
            self.Y[i,:] = self.update_Y(i, XTX)
        self.score.append([(t+1), self.evaluate()])
        print(f'Step {self.score[t][0]}, score {self.score[t][1]}')
```

$$Y^T C^u Y = Y^T Y + Y^T (C^u - I) Y$$

$$Y^T C^u = Y^T (C^u - I) + Y^T$$

```python
def update_X(self, u, yty):
    sparse_Y = sparse.csr_matrix(self.Y)
    Cu_I = sparse.diags(self.C_I[u].toarray()[0], format='csr')
    YTCuY = sparse_Y.T.dot(Cu_I).dot(sparse_Y) + yty
    YTCu = sparse_Y.T+sparse_Y.T.dot(Cu_I)
    Xu = np.linalg.inv(YTCuY+self.regularize*np.identity(self.K)).dot(YTCu.dot(self.P[u,:].T).toarray())
    return Xu.T[0]
```

Updating Factor Matrix in Full Dataset

```
observations = np.array(dataset['watched_episodes'])
row = np.array(dataset['user_id'])
col = np.array(dataset['anime_id'])
train_matrix = sparse.csr_matrix((observations, (row,col)), dtype=int)
train_matrix.shape
```

✓ 36.7s

```
(353405, 48493)
```

```
MemoryError: Unable to allocate 63.8 GiB for an array with shape (353405, 48493) and data type int32
```

```
u = 1

sparse_Y = sparse.csr_matrix(Y)
Cu_I = sparse.diags(C_I[u].toarray()[0], format='csr')
YTCuY = sparse_Y.T.dot(Cu_I).dot(sparse_Y) + YTY
YTCu = sparse_Y.T+sparse_Y.T.dot(Cu_I)
Xu = np.linalg.inv(YTCuY+regularize*np.identity(K)).dot(YTCu.dot(P[u,:].T).toarray())
Xu.T
```

✓ 0.5s

Evaluating Function

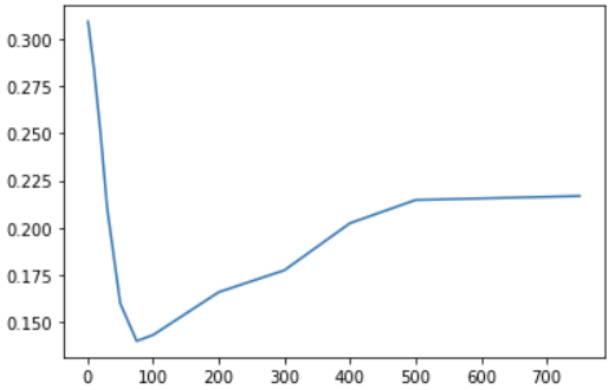
```
def evaluate(self):
    M = 10**5
    for_rank = -self.predicted_preference()
    observed_items = [0]*self.user_num
    rank = np.zeros((self.user_num, self.item_num))
    nonzero = np.array(list(self.R.nonzero()))
    for u, i in nonzero.T:
        for_rank[u,i] = M
        observed_items[u] += 1
    for u in range(self.user_num):
        rank[u,:] = for_rank[u,:].argsort().argsort()/(self.item_num-1-observed_items[u])
    return np.sum(self.test.toarray()*rank)/np.sum(self.test.toarray())
```

큰 수 M을 도입해서 nonzero observation이 있는 경우 rank에서 제외(test set에 존재하지 않기 때문)

각 user마다 maximum rank가 달라 percentile rank를 따로 계산

Cross Validation for Setting Regularization Parameter

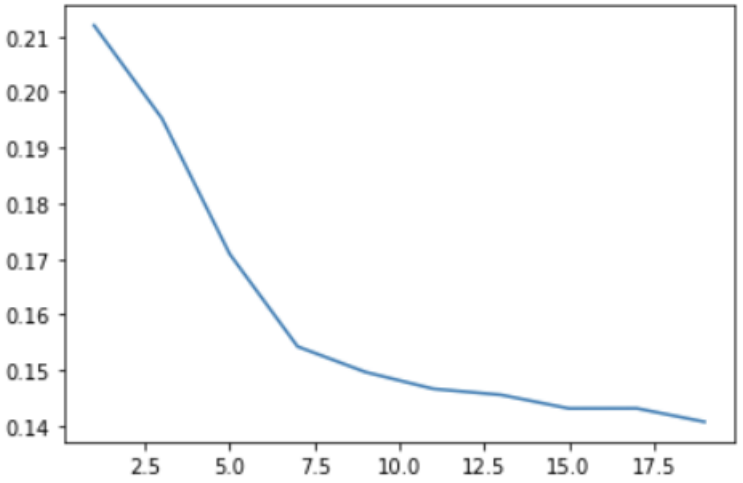
```
datasize = dataset.shape[0]
validsize = int(datasize/5)

for_best_parameter = []

for parameter in [1,10,20,30,50,75,100,200,300,400,500]:
    sum_score = 0
    for fold in range(5):
        train, test = train_test_split(dataset, test_size=0.2, shuffle=True, random_state=10)

        observations = np.array(train['watched_episodes'])
        row = np.array(train['user_id'])
        col = np.array(train['anime_id'])
        train_matrix = sparse.csr_matrix((observations, (row,col)), dtype=int)

        test_observations = np.array(test['watched_episodes'])
        test_row = np.array(test['user_id'])
        test_col = np.array(test['anime_id'])
        test_matrix = sparse.csr_matrix((test_observations, (test_row,test_col)), dtype=int)

        Model = ALS(train_matrix, test_matrix,dimension, alpha, epsilon, epochs, parameter)
        Model.fit()
        sum_score += Model.evaluate()


        index = list(range(datasize-validsize, datasize)) + list(range(0, datasize-validsize))
        dataset = dataset.reindex(index = index)
        print(f'parameter: {parameter}, Fold: {fold+1}, average_score: {sum_score/(fold+1)}')
    for_best_parameter.append([parameter, sum_score/5])
```



$\lambda = 75$일때 expected rank 가 약 0.1401로 최소

Model Performance in Various Dimensions

```
regularize = 75
results = []

for dim in list(range(1,20,2)):
    Model = ALS(train_matrix, test_matrix, dim, alpha, epsilon, epochs, regularize)
    Model.fit()
    results.append([dim, Model.evaluate()])
    print(f'dimension: {dim}, score: {Model.evaluate()}')
```



차원이 증가함에 따라 모델의 성능이 개선됨

약 14%에 수렴

논문에 나온 수치 8.6%보다 성능이 떨어짐