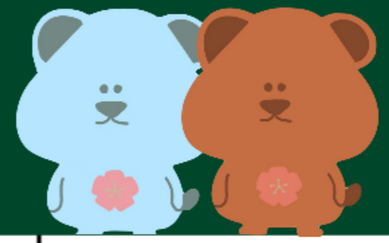


파이썬 머신러닝 완벽가이드 9장

4팀 허성은 최유미

EWHA EURON

목차

#01 추천 시스템의 개요와 발견

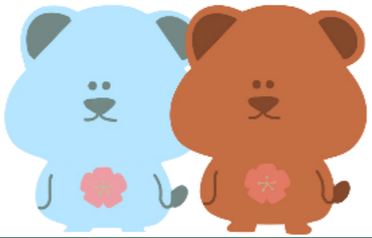
#02 콘텐츠 기반 필터링 추천 시스템

#03 최근접 이웃 협업 필터링

#04 잠재 요입 협업 필터링

EWHA EURON

9.1 추천 시스템의 개요와 배경



EWHA EURON

#추천 시스템의 개요

추천 시스템 | 사용자의 취향을 이해하고 맞춤 상품과 콘텐츠를 제공하는 시스템

전자상거래 업체

- 조금이라도 오래 사이트에 고객을 머무르게 하여 매출 향상
- 사용자의 쇼핑 즐거움이 상승됨

콘텐츠 포털

- 몇 개의 콘텐츠만 클릭해도 이와 유사하거나 연관된 콘텐츠가 추천됨
- 사용자가 관심을 가질만한 콘텐츠 + 새로운 취향의 콘텐츠 -> 대부분 개인적 취향을 정확히 이해하고 추천

✓ 사용자의 관심을 오랫동안 지속하기 위해 추천 시스템의 고도화에 큰 비용과 투자를 기울이고 있음

#온라인 스토어의 필수 요소, 추천 시스템

한정된 시간, 너무 많은 상품과 콘텐츠로 인한 선택의 어려움 -> **추천 시스템**으로 사용자의 선택 부담 해결

온라인 스토어의 추천 시스템

- 사용자가 어떤 상품을 구매했는가?
- 사용자가 어떤 상품을 둘러보거나 장바구니에 넣었는가?
- 사용자가 평가한 영화 평점은? 제품 평가는?
- 사용자가 스스로 작성한 자신의 취향은?
- 사용자가 무엇을 클릭했는가?

✓ 위의 데이터를 기반으로 추천 시스템이 구성됨

〈 데이터 기반의 추천 시스템 〉

#추천 시스템의 유형

- ✓ 콘텐츠 기반 필터링(Content based filtering)

- ✓ 협업 필터링(Collaborative filtering)
  - ✓ 최근접 이웃(Nearest Neighbor) 협업 필터링
  - ✓ 잠재 요인(Latent Factor) 협업 필터링

초창기: 콘텐츠 기반 필터링, 최근접 이웃 기반 협업 필터링 주요 사용

최근: **행렬 분해(Matrix Factorization) 기반 잠재 요인 필터링, 콘텐츠 기반 & 협업 결합 필터링**

9.2 콘텐츠 기반 필터링 추천 시스템

EWHA EURON

#콘텐츠 기반 필터링

✓ 사용자가 선택한 콘텐츠와 비슷한 콘텐츠를 추천하는 방식

*비슷한 콘텐츠: (영화) 장르, 감독, 출연 배우, 키워드

장르: SF, 드라마, 미스테리
감독: 드니 빌뇌브
출연: 에이미 아담스, 제레미 러너
키워드: 외계인 침공, 예술성, 스릴러 요소

장르: SF, 액션, 스릴러
감독: 드니 빌뇌브
출연: 라이언 고슬링, 해리슨 포드
키워드: 리들리 스콧 감독의 전작을 리메이크

장르: SF, 액션, 스릴러
감독: 리들리 스콧
출연: 노미 라마스, 마이클 패스벤더
키워드: 에일리언 프리퀄, 액션과 스리러의 조화

03 최근접 이웃 협업 필터링

#01 협업 필터링

- 사용자 행동 양식만을 기반으로 추천을 수행하는 것
- 사용자-아이템 평점 매트릭스와 같은 축적된 사용자 행동 데이터를 기반으로 사용자가 평가하지 않은 아이템을 예측 평가
- 최근접 이웃 방식과 잠재 요인 방식으로 나뉜다.

#01 협업 필터링

- 사용자 행동 양식만을 기반으로 추천을 수행하는 것
- 사용자-아이템 평점 매트릭스와 같은 축적된 사용자 행동 데이터를 기반으로 사용자가 평가하지 않은 아이템을 예측 평가
- 최근접 이웃 방식과 잠재 요인 방식으로 나뉜다.

#01 사용자-아이템 평점 행렬

로우 레벨 형태의 사용자 - 아이템 평점 데이터

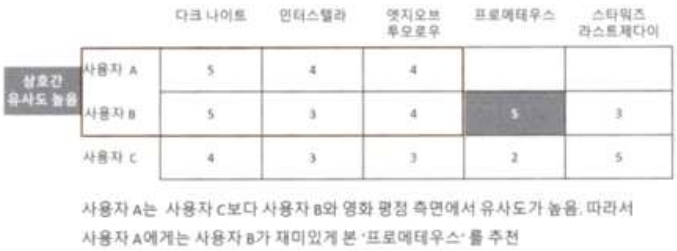
User ID	Item ID	Rating
User 1	Item 1	3
User 1	Item 3	3
User 2	Item 1	4
User 2	Item 2	1
User 3	Item 4	5

사용자 로우, 아이템 칼럼으로 구성된
사용자 - 아이템 평점 데이터

	Item 1	Item 2	Item 3	Item 4
User 1	3		3	
User 2	4	1		
User 3				5

- 행은 개별 사용자, 열은 개별 아이템으로 구성되며, 각 행과 열에 해당하는 값이 평점을 나타내는 형태가 된다.
- 데이터가 레코드 레벨 형태(왼쪽)라면 협업 필터링 알고리즘에 사용하기 위해 사용자-아이템 평점 행렬 형태(오른쪽)로 변경한다.
- 사용자가 아이템에 대한 평점을 많이 매기지 않아 대부분 희소 행렬이다.

02 최근접 이웃 협업 필터링-사용자 기반



- 최근접 이웃 협업 필터링은 사용자 기반/아이템 기반으로 나뉜다.
- 사용자 기반 최근접 이웃 방식은 특정 사용자와 유사한 다른 사용자를 Top-N으로 선정해 이 N명의 사용자가 좋아하는 아이템을 추천한다.
- 즉, 사용자간 유사도를 측정한 뒤 가장 유사도가 높은 사용자 N명을 추출하고, 그들이 선호하는 아이템을 추천한다.

02 최근접 이웃 협업 필터링-아이템 기반

상호간 유사도 놀음	사용자 A	사용자 B	사용자 C	사용자 D	사용자 E
다크 나이트	5	4	5	5	5
프로메테우스	5	4	4		5
스타워즈 라스트제다이	4	3	3		4

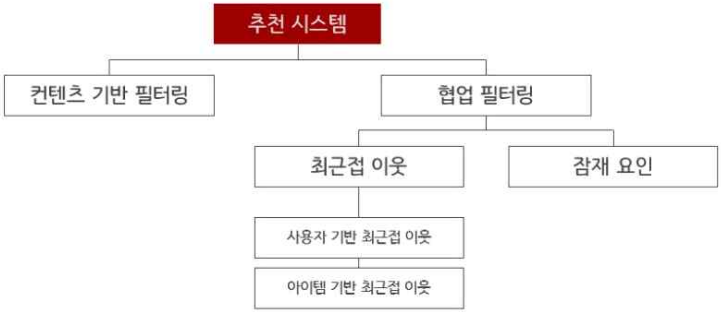
여러 사용자들의 평점을 기준으로 볼 때 '다크 나이트'와 가장 유사한 영화는 '프로메테우스'

- 아이템 기반 최근접 이웃 방식은 아이템이 가지는 속성과는 상관없이 사용자들이 그 아이템을 좋아/싫어하는지의 평가 척도가 추천 기준이 된다.
- 일반적으로 사용자 기반보다는 아이템 기반 협업 필터링이 정확도가 더 높다.
- 유사도 측정 방법은 텍스트 분석과 동일한 코사인 유사도를 사용한다.

04 잠재 요인 협업 필터링

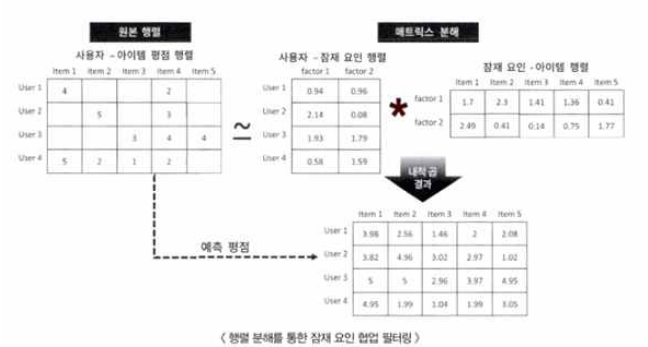
EWHA EURON

01 잠재 요인 협업 필터링



- 사용자-아이템 평점 매트릭스 속에 숨어 있는 잠재 요인을 추출해 추천 예측을 할 수 있게 하는 기법이다.
- 대규모 다차원 행렬을 차원 축소 기법으로 분해하는 과정에서 잠재요인을 추출하며, 이를 행렬 분해라 한다.
- 많은 추천 시스템이 행렬 분해에 기반한 잠재 요인 협업 필터링을 적용하고 있다.

01 잠재 요인 협업 필터링

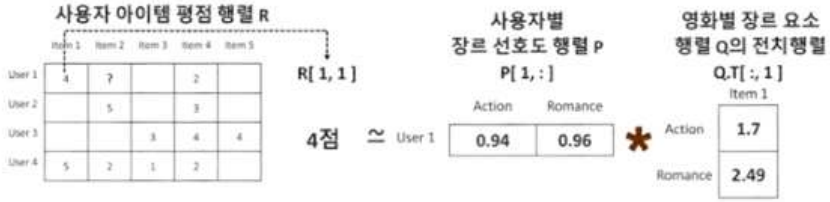


〈 행렬 분해를 통한 잠재 요인 협업 필터링 〉

- 잠재 요인을 기반으로 다차원 희소 행렬의 **사용자-아이템 행렬** 데이터를 저차원 밀집 행렬의 **사용자-잠재 요인 행렬**과 **아이템-잠재 요인 행렬의 전치 행렬**로 분해한다.

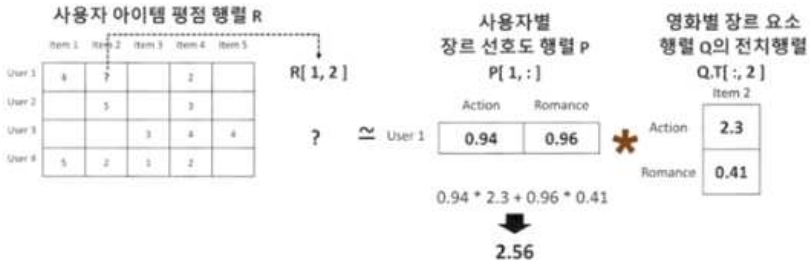
- 분해된 두 행렬의 내적을 통해 새로운 예측 사용자-아이템 평점 행렬 데이터를 만들어 사용자가 아직 평점을 부여하지 않은 아이템에 대한 예측 평점을 생성한다.

01 잠재 요인 협업 필터링-예시



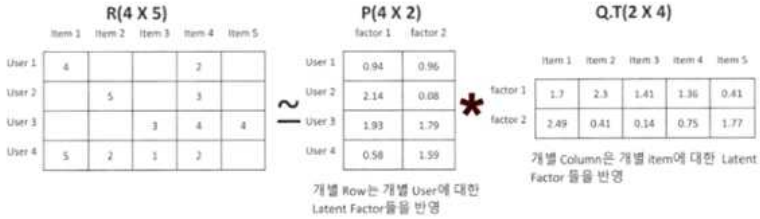
- 사용자-아이템 평점 행렬 R(u,i) : u는 사용자, i는 아이템
- 사용자-잠재 요인 행렬 P(u,k) : u는 사용자, k는 잠재 요인 칼럼(잠재별 선호도)
- 아이템-잠재 요인 행렬 Q(i,k) : i는 아이템 아이디, k는 잠재 요인 칼럼(장르별 요소)
- Q와 P의 내적 계산을 위해 Q는 전치 행렬 Q.T로 변환한다.
- 평점은 사용자의 장르별 선호도 벡터와 영화의 장르별 특성 벡터를 서로 곱해서 만들 수 있다.

01 잠재 요인 협업 필터링-예시



- 같은 방식으로, User10이 평점을 매기지 못한 Item2에 대해 예측 평점을 구할 수 있다.
- R(1,2)는 행렬 분해된 P 행렬의 User1 벡터값과 Q.T 행렬의 Item2 벡터의 내적 결과값이다.
- R(1,2) = 0.94*2.3 + 0.96*0.41 = 2.56

02 행렬 분해

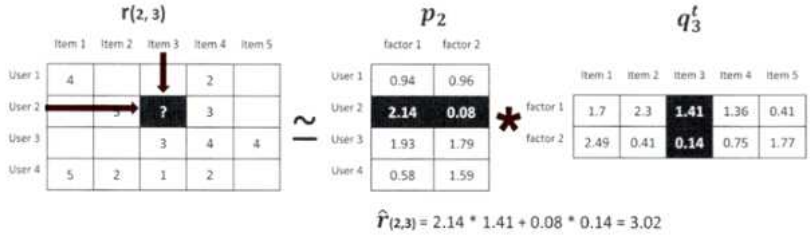


- R = P*Q.T
- 차원으로 보면 MxN = MxK * K*N
- R은 평점 행렬, P는 사용자 잠재 요인 행렬, Q는 아이템-잠재 요인 행렬이다.
- 고차원의 희소 행렬 R는 저차원의 밀집 행렬 P 행렬과 Q행렬로 분해된다.

02 행렬 분해

$$r_{(u,i)} = p_u * q_i^t$$

- R 행렬의 u행 사용자와 i열 아이템 위치에 있는 평점 데이터 r(u,i)는 위와 같이 구할 수 있다.



- 사용자가 평가하지 않은 아이템에 대한 평점도 잠재 요인으로 분해된 P행렬과 Q행렬을 이용해 예측할 수 있다.

- 이 방법을 반복해 NaN값을 포함한 모든 평점 값은 행렬 분해로 얻은 행렬들의 내적을 통해 예측 평점으로 다시 계산할 수 있다.

02 행렬 분해-확률적 경사 하강법

- 행렬 분해는 주로 SVD 방식을 이용하지만, SVD는 널값이 없는 행렬에만 적용할 수 있으므로 확률적 경사 하강법(SGD)을 이용한다.
- 따라서 예측 R 행렬 값이 실제 R행렬 값과 가장 최소의 오류를 가질 수 있도록 반복적인 비용 함수 최적화를 통해 P와 Q를 유추해낸다.

1. P와 Q를 임의의 값을 가진 행렬로 설정한다.
2. P와 Q.T 값을 곱해 예측 R행렬을 계산하고 예측 R 행렬과 실제 R 행렬에 해당하는 오류 값을 계산한다.
3. 이 오류값을 최소화할 수 있도록 P와 Q행렬을 적절한 값으로 각각 업데이트한다.
4. 만족할 만한 오류 값을 가질 때까지 2,3번 작업을 반복하면서 P와 Q값을 업데이트해 근사화한다.

02 행렬 분해-확률적 경사 하강법

$$min\sum(r_{(u,i)} - p_u q_i^t)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

- 실제 값과 예측값의 오류 최소화와 L2 규제를 고려한 비용 함수식이다.

$$p'_u = p_u + \eta(e_{(u,i)} * q_i - \lambda * p_u)$$
$$q'_i = q_i + \eta(e_{(u,i)} * p_u - \lambda * q_i)$$

- 위 비용 함수를 최소화 하기 위해 새롭게 업데이트 되는 P와 Q 행렬의 계산식이다.

02 행렬 분해-예제

```python
import numpy as np

# 원본 행렬 R 생성, 분해 행렬 P와 Q 초기화, 잠재요인 차원 K는 3 설정.
R = np.array([[4, np.NaN, np.NaN, 2, np.NaN ],
              [np.NaN, 5, np.NaN, 3, 1 ],
              [np.NaN, np.NaN, 3, 4, 4 ],
              [5, 2, 1, 2, np.NaN ]])
num_users, num_items = R.shape
K=3

# P와 Q 매트릭스의 크기를 지정하고 정규분포를 가진 random한 값으로 입력합니다.
np.random.seed(1)
P = np.random.normal(scale=1./K, size=(num_users, K))
Q = np.random.normal(scale=1./K, size=(num_items, K))
```

- SGD를 이용해 행렬 분해를 파이썬으로 구현한다.
- 원본 행렬 R을 널(np.NaN)값을 포함해 생성하고 분해 행렬 P와 Q는 정규 분포를 가진 랜덤 값으로 초기화한다.

02 행렬 분해-예제

```python
from sklearn.metrics import mean_squared_error

def get_rmse(R, P, Q, non_zeros):
    error = 0
    # 두개의 분해된 행렬 P와 Q.T의 내적으로 예측 R 행렬 생성
    full_pred_matrix = np.dot(P, Q.T)

    # 실제 R 행렬에서 널이 아닌 값의 위치 인덱스 추출하여 실제 R 행렬과 예측 행렬의 RMSE 추출
    x_non_zero_ind = [non_zero[0] for non_zero in non_zeros]
    y_non_zero_ind = [non_zero[1] for non_zero in non_zeros]
    R_non_zeros = R[x_non_zero_ind, y_non_zero_ind]
    full_pred_matrix_non_zeros = full_pred_matrix[x_non_zero_ind, y_non_zero_ind]

    mse = mean_squared_error(R_non_zeros, full_pred_matrix_non_zeros)
    rmse = np.sqrt(mse)

    return rmse
```

- 실제 R 행렬과 예측 행렬의 오차를 구하는 get_rmse() 함수를 만들었다.
- 실제 R 행렬의 널이 아닌 행렬 값의 위치인덱스를 추출해 이 인덱스에 있는 실제 R 행렬 값과 분해된 P,Q를 이용해 다시 조합된 예측 행렬 값의 RMSE값을 반환한다.

02 행렬 분해-예제

```
# R > 0 인 행 위치, 열 위치, 값을 non_zeros 리스트에 저장.
non_zeros = [ (i, j, R[i,j]) for i in range(num_users) for j in range(num_items) if R[i,j] > 0 ]

steps=1000
learning_rate=0.01
r_lambda=0.01

# SGD 기반으로 P와 Q 매트릭스를 계속 업데이트.
for step in range(steps):
    for i, j, r in non_zeros:
        # 실제 값과 예측 값의 차이인 오류 값 구함
        eij = r - np.dot(P[i, :], Q[j, :].T)
        # Regularization을 반영한 SGD 업데이트 공식 적용
        P[i,:] = P[i,:] + learning_rate*(eij * Q[j, :] - r_lambda*P[i,:])
        Q[j,:] = Q[j,:] + learning_rate*(eij * P[i, :] - r_lambda*Q[j,:])

    rmse = get_rmse(R, P, Q, non_zeros)
    if (step % 50) == 0 :
        print("### iteration step : ", step," rmse : ", rmse)
```

```
### iteration step :  0  rmse :  3.2388050277987723
### iteration step :  50  rmse :  0.4876723101369648
### iteration step :  100  rmse :  0.1564340384819247
### iteration step :  150  rmse :  0.07455141311978046
### iteration step :  200  rmse :  0.04325226798579314
### iteration step :  250  rmse :  0.029248328780878973
### iteration step :  300  rmse :  0.022621116143829466
### iteration step :  350  rmse :  0.019493636196525135
### iteration step :  400  rmse :  0.018022719092132704
### iteration step :  450  rmse :  0.01731968595344266
### iteration step :  500  rmse :  0.016973657887570753
### iteration step :  550  rmse :  0.016796804595895633
### iteration step :  600  rmse :  0.01670132290188466
### iteration step :  650  rmse :  0.01664473691247669
### iteration step :  700  rmse :  0.016605910068210026
### iteration step :  750  rmse :  0.016574200475705
### iteration step :  800  rmse :  0.01654431582921597
### iteration step :  850  rmse :  0.01651375177473524
### iteration step :  900  rmse :  0.01648146573819501
### iteration step :  950  rmse :  0.016447171683479155
```

- SGD를 기반으로 행렬 분해를 수행한다.
- R에서 널 값을 제외한 데이터의 행렬 인덱스를 추출한다.
- steps는 업데이트할 횟수, learning_rate는 SGD의 학습률, r_lambda는 L2 규제 계수이다.
- 1000번 반복하며 새로운 P,Q 행렬로 업데이트를 진행하고, get_rmse() 함수를 이용해 오류값을 출력한다.

02 행렬 분해-예제

```
pred_matrix = np.dot(P, Q.T)
print('예측 행렬:\n', np.round(pred_matrix, 3))
```

```
예측 행렬:
 [[3.991 0.897 1.306 2.002 1.663]
 [6.696 4.978 0.979 2.981 1.003]
 [6.677 0.391 2.987 3.977 3.986]
 [4.968 2.005 1.006 2.017 1.14 ]]
```

```
print('원본 행렬:\n', np.round(R, 3))
```

```
원본 행렬:
 [[ 4. nan nan  2. nan]
 [nan  5. nan  3.  1.]
 [nan nan  3.  4.  4.]
 [ 5.  2.  1.  2. nan]]
```

- 분해된 P와 Q 함수를 P*Q.T로 예측 행렬을 만들어서 출력한다.
- 원본 행렬과 비교했을때 널이 아닌 값은 큰 차이가 없고, 널인 값은 새로운 예측값으로 채워졌다.

THANK YOU