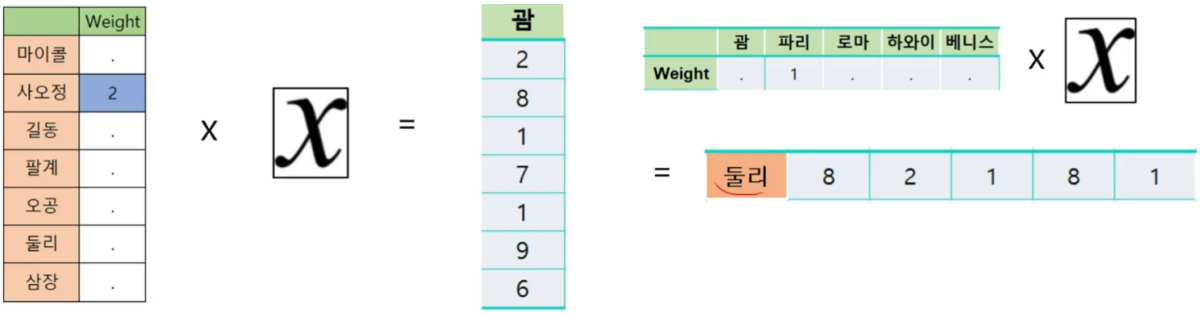
Matrix Factorization

Date @2021/09/15

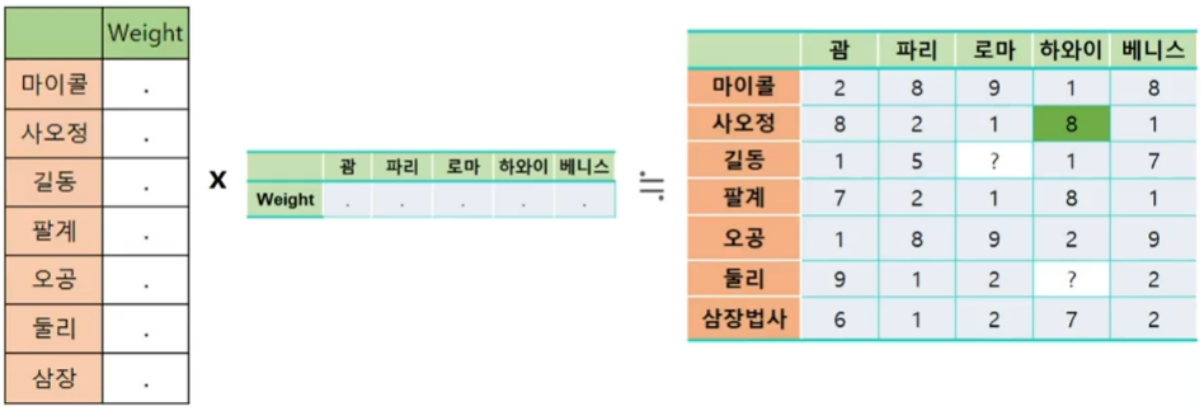
	괌	파리	로마	하와이	베니스
마이콜	2	8	9	1	8
사오정	8	2	1	8	1
길동	1	5	?	1	7
팔계	7	2	1	8	1
오공	1	8	9	2	9
둘리	9	1	2	?	2
삼장법사	6	1	2	7	2

- 특성이 비슷한 장소에 대해 특정 유저가 어떤 선호도를 보일 것인지 예측하는 것
- 유저가 이 정도 규모가 아닌 무수히 많아진다면...?

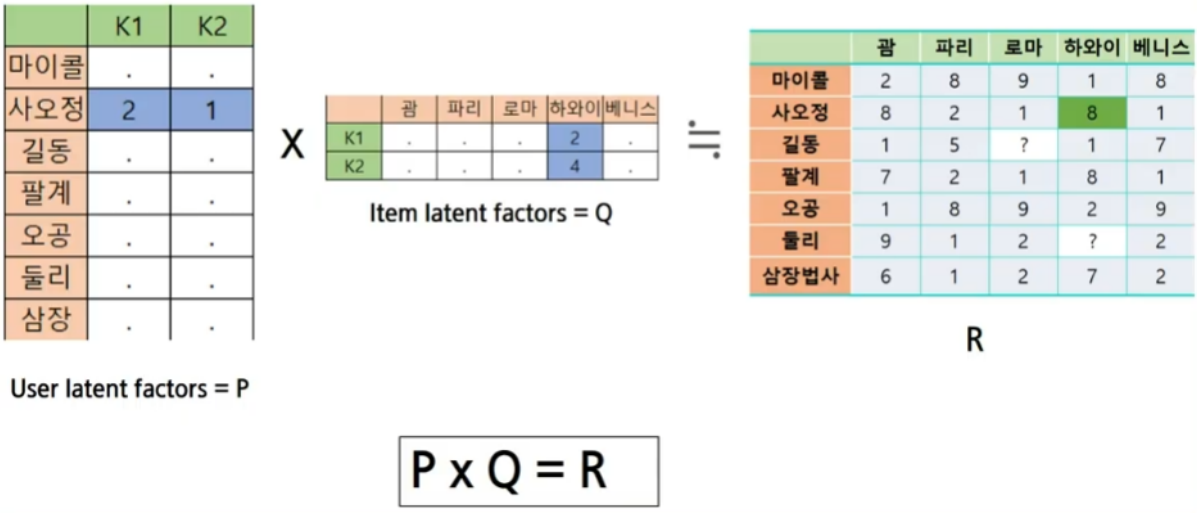
개념



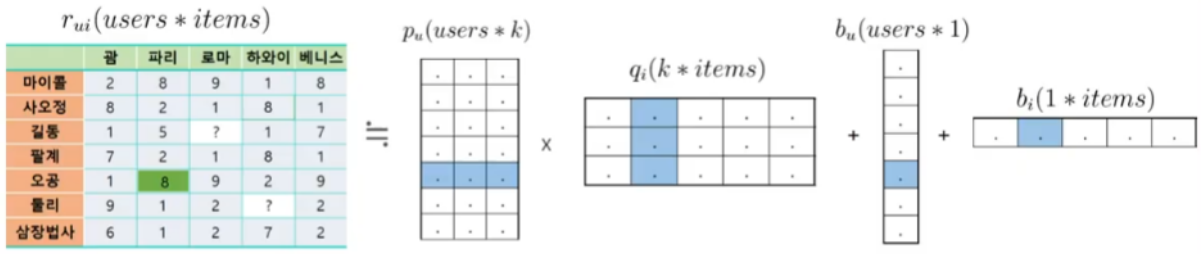
⇒ 임의에 x에 대해서 연산이 된 후 결과가 나온 다는 것을 알 수 있다.



따라서 정리하면 이 모습처럼 결과를 예측할 수 있다. 하지만 단순한 차원으로 인해 결과 역시 단조로울 수 있다.

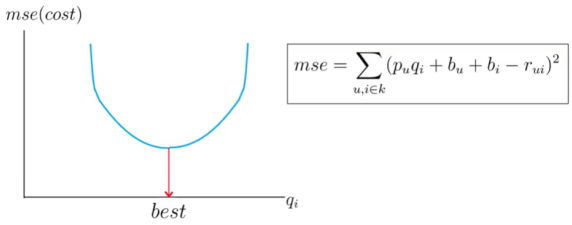


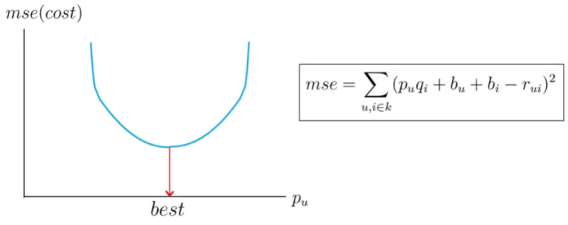
차원을 늘릴 수로 예측이 상세하게 나오지만, 그만큼의 시간 소모가 들어가게 될 것이다.

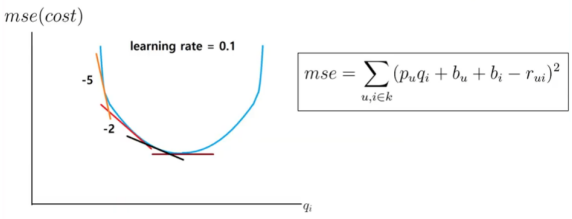


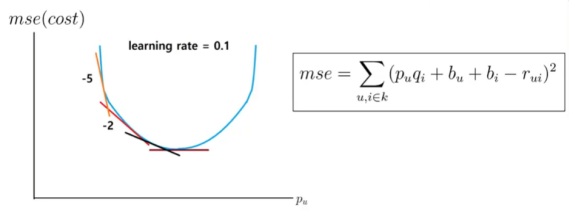
$$mse = \sum_{u,i \in k} (p_u q_i + b_u + b_i - r_{ui})^2$$

경사 하강법









편미분을 하면 그래프의 기울기를 구할 수 있기에, 이를 반복하면 포물선 그래프에서의 최소값을 확인할 수 있다.

$$p_{u(new)} = p_u - \gamma \sum (p_u q_i + b_u + b_i - r_{ui}) q_i$$

$$b_{u(new)} = b_u - \gamma \sum (p_u q_i + b_u + b_i - r_{ui})$$

$$q_{i(new)} = q_i - \gamma \sum (p_u q_i + b_u + b_i - r_{ui}) p_u$$

$$b_{i(new)} = b_i - \gamma \sum (p_u q_i + b_u + b_i - r_{ui})$$

실제 코드

```python
iteration = 5000
k = 3
R = np.array([
    [2, 8, 9, 1, 8],
    [8, 2, 1, 8, 1],
    [1, 5, -1, 1, 7],
    [7, 2, 1, 8, 1],
    [1, 8, 9, 2, 9],
    [9, 1, 2, -1, 2],
    [6, 1, 2, 7, 2]])

predicted_R = matrix_factorization(R, k, iteration)

print(predicted_R)
```

```
[[2.0  8.0  8.9  0.9  8.0]
 [7.9  1.9  1.0  8.0  0.9]
 [0.9  4.9  7.4  1.0  7.0]
 [7.0  2.0  0.9  7.9  1.0]
 [0.9  8.0  9.0  2.0  8.9]
 [8.9  1.0  2.0  9.2  1.9]
 [6.0  0.9  1.9  6.9  2.0]]
```

```python
def matrix_factorization(R, k, iteration):

    user_count, item_count = R.shape

    P = np.random.normal(size=(user_count, k))
    Q = np.random.normal(size=(item_count, k))

    bu = np.zeros(user_count)
    bi = np.zeros(item_count)

    for iter in range(iteration):
        for u in range(user_count):
            for i in range(item_count):
                r = R[u, i]
                if r >= 0:
                    error = prediction(P[u, :], Q[i, :], bu[u], bi[i]) - r

                    delta_Q, delta_bi = gradient(error, P[u, :])
                    delta_P, delta_bu = gradient(error, Q[i, :])

                    P[u, :] -= delta_P
                    bu[u] -= delta_bu

                    Q[i, :] -= delta_Q
                    bi[i] -= delta_bi

    return P.dot(Q.T) + bu[:, np.newaxis] + bi[np.newaxis:, ]
```

$$p_{u(new)} = p_u - \gamma \sum (p_u q_i + b_u + b_i - r_{ui}) q_i$$

$$b_{u(new)} = b_u - \gamma \sum (p_u q_i + b_u + b_i - r_{ui})$$

$$q_{i(new)} = q_i - \gamma \sum (p_u q_i + b_u + b_i - r_{ui}) p_u$$

$$b_{i(new)} = b_i - \gamma \sum (p_u q_i + b_u + b_i - r_{ui})$$

```python
def prediction(P, Q, bu, bi):

    return P.dot(Q.T) + bu + bi
```

$$p_u q_i + b_u + b_i$$

```python
def gradient(error, weight):

    learning_rate = 0.005

    weight_delta = learning_rate * np.dot(weight.T, error)

    bias_delta = learning_rate * np.sum(error)

    return weight_delta, bias_delta
```

$$p_{u(new)} = p_u - \gamma \sum (p_u q_i + b_u + b_i - r_{ui}) q_i$$

$$b_{u(new)} = b_u - \gamma \sum (p_u q_i + b_u + b_i - r_{ui})$$

$$q_{i(new)} = q_i - \gamma \sum (p_u q_i + b_u + b_i - r_{ui}) p_u$$

$$b_{i(new)} = b_i - \gamma \sum (p_u q_i + b_u + b_i - r_{ui})$$