벡터의 차원이 늘어나면 우리가 입력해야할 데이터의 크기가 늘어난다. 단, 문제는 우리가 세상의 모든 정보를 알 수 없는 것에 있다.
즉, 세상의 모든 정보를 모르니까 입력해야할 데이터의 크기가 늘어나도 정확하게 표현 할 수 없다.

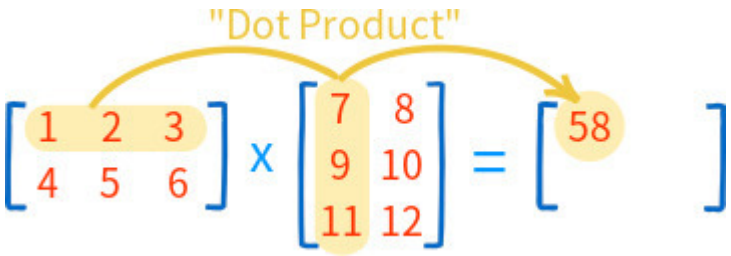
때문에 임베딩을 통해서 정보를 축소 시킨다. 하지만, 무언가를 축소한다는건 반드시 손실이 따른다. 이러한 손실을 최소화 하기 위해 여러 방법이 있겠지만 이번에는 행렬 분해에 대해 배웠다.

행렬 분해(Matrix Factorization)

우선 행렬의 곱셈에 대해 먼저 알아야하는데 꼭 기억해야할 부분은 **두 행렬의 곱셈은 순수가 매우 중요하는 점이다.**

두 수 2와 3은 어느 수를 앞에 두어도 곱셈 결과가 같지만 (2×3=3×2=6), 두 행렬 A, B 간의 곱셈은 어떤 행렬을 앞에 두냐에 따라 결과가 달라진다. (A·B≠B·A).
곧 설명할 두 행렬의 모양에 대한 룰을 지키지 못할 경우, 곱셈 연산 자체를 수행할 수 없기도 함. 따라서 어느 행렬을 앞에 놓을지 반드시 먼저 정해야 함.

이렇게 행렬의 순서를 정하고 나면, 왼쪽의 행렬은 가로(행) 방향의 벡터를 한 덩어리로, 오른쪽에 있는 행렬은 세로(열) 방향의 벡터를 한 덩어리로 생각합니다. 그 후 각 행 덩어리와 열 덩어리를 서로 곱해 결과값 행렬의 셀을 채웁니다. 아래 그림을 예로 들면, 왼쪽 행렬의 제1행과 오른쪽 행렬의 제1열의 내적(dot product)값은 1·7+2·9+3·11=58 이며 이 값은 결과값 행렬의 (1,1) 셀에 위치하게 됨. 왼쪽 행렬의 제2행과 오른쪽 행렬의 제2열이 곱해지면 결과값 행렬의 오른쪽 아래 코너를 채우게 됨.



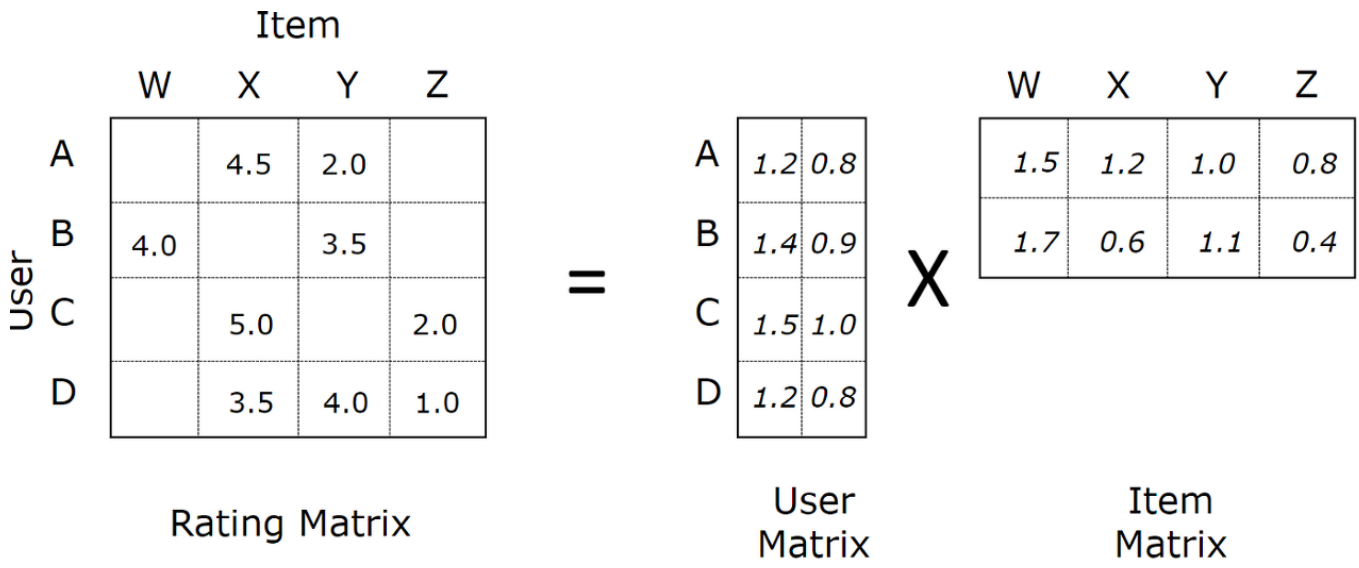
왼쪽 행렬의 행의 갯수는 결과값 행렬의 행의 갯수와 연관되어 있고, 오른쪽 행렬의 열의 갯수는 결과값 행렬의 열의 갯수와 연관되어 있다.

1. 결과값 행렬의 모양은 왼쪽 행렬의 행의 갯수, 오른쪽 행렬의 열의 갯수에 의해 정해진다.
2. 왼쪽 행렬의 열의 갯수와 오른쪽 행렬의 행의 갯수가 일치해야지만 결과값 행렬을 온전히 채울 수 있다.

수학적인 표현으로 정리하자면, m×n 차원(행이 m개, 열이 n개)의 행렬 A와 j×k 차원의 행렬 B의 행렬곱 AB는 m×k 차원의 행렬이며, n=j 이어야지만 행렬곱을 구할 수 있습니다.

행렬 분해와 행렬 곱의 관계

행렬 곱을 왜 말했냐? 행렬 분해가 행렬 곱 거꾸로임.



그림은 행렬 곱을 반대 방향으로 뒤집은 모습이고 이게 행렬 분해라고 함.

행은 유저, 열은 아이템으로 구성된 평점 매트릭스(R). 이 매트릭스에서 i번째 행, j번째 열에 해당하는 값은 유저 i가 아이템 j에 준 평점을 나타낸다. 등호(=)의 오른편에 있는 두 매트릭스는 각각 '유저 매트릭스(U)'와 '아이템 매트릭스(I)'라 부른다.

행렬 곱을 뒤집은게 행렬 분해라고 했으니 행렬 곱의 특성 2가지를 뒤집은게 행렬 분해의 특성이라고 함.

1. R의 행의 갯수는 U의 행의 갯수와 일치하고, R 의 열의 갯수는 I의 열의 갯수와 일치해야한다.
2. U의 열의 갯수와 I의 행의 갯수는 일치한다.

곰셈의 결과가 평점 매트릭스를 닮도록 분해하기

MF알고리즘은 매트릭스 U와 I의 행렬곱 U·I 가 매트릭스 R과 최대한 일치한 값을 갖게끔 두 매트릭스의 셀 값을 조정한다.
가장 먼저 랜덤한 숫자로 매트릭스 U와 I채운다. 그리고 U,I를 행렬곱하고 R과 비교한다.

> 행렬곱 U·I는 예측값(prediction), R은 실제값(ground truth)이라 부른다.

첫 예측 오류는 각 셀에 대해 예측값과 실제값 간의 차이를 구한 후 이를 모두 더한 값이다.

두번째 목적으로 매트릭스 U,I 셀 값을 조정해서 조금 전보다 더 잘하는 것이다. 이전의 예측 오류보다 더 작은 오류를 달성하는게 목표다. 이런 과정을 계속해서 반복하게 된다.
즉, MF알고리즘은 매 loop마다 예측값과 실제값 간의 오차를 점차 줄이려는 목표를 가지고 있고 다음과 같은 목표 함수로 표현된다.

위 함수를 최소화 하는 최적의 방법을 찾기 위해 최적화 알고리즘을 활용해야하고, 결과에 따라 매트릭스 U,I의 셀 값을 저정하게 된다.

Matrix Factorization 목적 함수를 최적화 시키는 알고리즘 소개

Matrix Factorization(행렬 분해)는 아이템 점수 패턴으로 부터 추론된 요소의 vector들로 아이템과 사용자들 모두 특성화해서 학습시키는 기법을 의미함.
특히, 아이템과 사용자 간의 높은 상관 관계를 지닐 경우 추천하는 방법이다.

특징으로는

1. 예측정확도와 우수한 확장성
2. 다양한 실상황을 모델링 하기 위한 뛰어난 유연성 제공
3. 손실을 최소화하며 임베딩하는 방법

추천 시스템의 경우 입력 데이터의 다양한 유형에 의존하는데 **Matrix Factorization의 경우 추가적인 정보 통합을 허용한다.**
이게 무슨 말이냐면 명시적으로 피드백을 사용할 수 없을 경우, 추천 시스템은 과거 구매, 브라우저 쿠키, 검색 패턴 등 **사용자의 행동 패턴을 관찰한 암시적 피드백(Implicit Feedback)을 사용해서 사용자 선호를 유추한다.**

> 암시적 피드백(Implicit Feedback) : 이벤트 존재 유무를 나타내므로 보통 조밀하게 채워진 밀집행렬(Dense Matrix)로 표시된다.

Basic Matrix Factorization Model

Matrix Factorization 모델은 **사용자와 아이템 차원(f)의 공동 잠재 요인 공간에 매핑한다. 이러한 사용자-아이템 상호 작용은 해당 공간에서 내적(inner-products)로 모델링한다.**

위의 내적 결과는 **사용자와 아이템 간의 상호 작용, 즉 아이템에 대한 사용자의 전반적인 관심을 나타냄** 이때 주어진 아이템(i)는 벡터 q_{i} 로, 주어진 사용자(u)는 p_{u} 벡터로 표한한다. 이때 발생하는 주요 문제는 요인 벡터(factor vector)에 대한 각 아잍메과 사용자의 매핑을 계산하는 것이다.

이와 같은 모델은 사용자 - 아이템 점수 행렬의 인수분해를 요구하는 특이값 분해(SVD, Singular Value Decomposition)과 매우 유사함. **사용자-아이템 점수 행렬의 희소성**으로 인해 SVD를 사용하는 것은 어렵다. 또한 비교적 적은 수로 알려진 항목들에대해 임의로 지정 할 경우 과적합(overfitting)문제가 발생할 수 있다.

> 과적합(overfitting) : 학습을 과하게 시켜 학습 데이터에선 최적의 결과를 내지만 새로운 데이터에 대해선 판단력이 부정확해지는 문제

따라서 정규화된 모델을 통해 과적합을 방지하면서 관측된 점수만 직접적으로 모델링하는 방법이 제시되었다. 요인 벡터, q_{i} 와 p_{u} 를 배우기 위해 시스템은 관측된 점수 세트를 통해 정규화된 Squared Error(제곱오차, 추정에 대한 정확성 측정 지표)를 최소화한다. 해당 시스템은 이전에 관찰된 점수들을 fitting하여 모델을 학습함.

그러나 이 모델은 알려지지 않은 점수를 예측하는 것이 목적임. 따라서 시스템은 학습된 매개 변수를 정규화해서 관찰된 데이터의 과적합을 방지해야 하고 주로 교차검증(cross validation)에 의해 결정되는 lambda로 정규화 범위를 제어한다.

Learning Algorithms

위 식을 최소화 하기 위한 방법으로는 두가지 방법을 제시함

Stochastic Gradient Descent(SGD, 확률적 경사 하강법)

요거는 빠르고 쉽게 구현할 수 있는데 각 training set에 대해 알고리즘은 r_{ui} 을 예측하고 아래와 같이 오차를 계산할 수 있음.

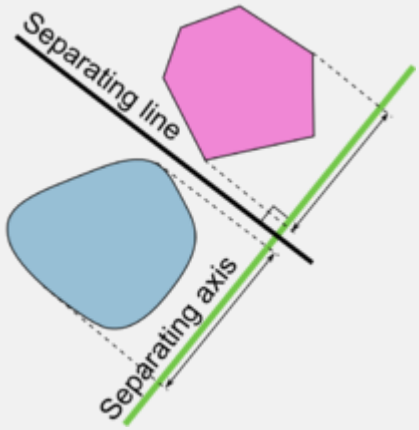
 2

산출한 후에는 gradient 의 반대 방향에서 gammam에 비례하는 크기로 각 q_i와 p_u를 아래처럼 수정해서 반영할 수 있음

 2

Alternating Least Squares(ALS)

위의 식은 q_{i}와 p_{u} 둘 다 알 수 없기 때문에 convex하다고 할 수 없습니다. 그래서 이번에는 각각의 값을 고정하면서 최소값을 찾는 방법으로 아래와 같이 동작함.

> convex하다??
> 아래 그림 처럼 집합 A, B가 초평면을 기준으로 완벽하게 분리된(disjoint) 초평면이 존재한다라는 의미로 받아들였음.
>
> 
>

하지만, 둘 중 하나를 고정하면 이 최적화 문제를 해결 할 수 있다. 즉, ALS은p_{u} 를 고정했다가 p_{u} 를 고정하는 방식으로 동작한다. 만약에 p_{u} 가 고정된 경우 시스템은 최소 제곱법으로 p_{u} 를 다시 계산한다고 함.

일반적으로 SGD가 ALS보다 빠르고 쉽다. 그런데 ALS을 왜 쓰냐면 특정 유리한 경우 2가지 때문이라고 함.

1. 병렬화를 지원하는 경우
2. 암시적 데이터에 중점을 둔 경우