

# Recommender Systems

Rob Hall  
rob.hall@faire.com

Search and Personalization Team  
Faire

- Recommender system setup.
- Collaborative filtering heuristics: item / user based CF.
- Matrix factorization based methods.
- Cool techniques: alternating least squares, locality sensitive hashing.

# Recommender Systems

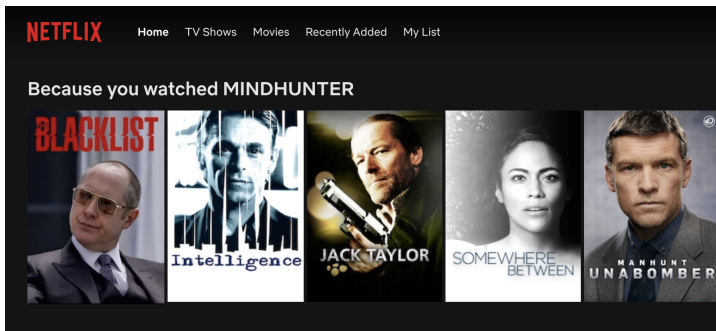
Basic idea: recommend **items** to **users** based on their past behavior.

Related to items you viewed



# Recommender Systems

Basic idea: recommend **items** to **users** based on their past behavior.





# Recommender Systems

Basic idea: recommend **items** to **users** based on their past behavior.

## Recommended for you

Because you viewed Antonio's Pizza



**Pasta E Basta**

●●●●○ 131 reviews



**The Black Sheep**

●●●●○ 126 reviews



**The Lone Wolf**

●●●●○ 178 reviews

# Recommender Systems

Say we have  $n$  users and  $m$  items. We get to see a partially complete matrix  $X \in \mathbb{R}^{n \times m}$ :

						...
Alice	★★★★★	★★★★☆	★★★★☆	?	?	
Bob	★★★★☆	★★★★★	?	★★★★☆	?	
Clair	?	?	?	★★★★☆	★★★★☆	
Dave	?	★★★☆☆	?	?	★★★★★	
⋮						⋮

# Recommender Systems

**Implicit feedback:** we don't get ratings just indicators of whether a user interacted with an item. Different setups, observe ratings vs observe implicit feedback.

						...
Alice	1	1	1	0	0	
Bob	1	1	0	1	0	
Clair	0	0	0	1	1	
Dave	0	1	0	0	1	
⋮						⋮

# Recommender Systems

- Basic idea: predict which unseen items a user would rate highly / interact with.
- Techniques:
  - Similarity based (item / user).
  - Matrix completion.

# Item-based Collaborative Filtering

Consider the cosine similarity between items  $j$  and  $k$ :



$$sim(j, k) = \frac{\sum_{i=1}^n x_{i,j}x_{i,k}}{\sqrt{\sum_{i=1}^n x_{i,j}^2}\sqrt{\sum_{i=1}^n x_{i,k}^2}}$$

i.e., the angle between columns  $j$  and  $k$  of the input matrix.

Tells us how similar the items are based on the ratings / interactions given by users.

**Note:** treating unknown values in the matrix as zero.

# Item-based Collaborative Filtering

			
Alice	1	1	1
Bob	1	1	0
Clair	0	0	0
Dave	0	1	0
⋮			


$$sim = \frac{2}{\sqrt{2}\sqrt{3}} \approx 0.82$$

# Item-based Collaborative Filtering

			
Alice	1	1	1
Bob	1	1	0
Clair	0	0	0
Dave	0	1	0
⋮			

$$sim = \frac{1}{\sqrt{1}\sqrt{2}} \approx 0.71$$





# Item-based Collaborative Filtering

		...	
Alice	1		0
Bob	1		0
Clair	0		1
Dave	0		1
⋮			

$$sim = \frac{0}{\sqrt{2}\sqrt{2}} = 0$$



# Item-based Collaborative Filtering

					...
	1.0	0.8	0.7	0.5	
	0.8	1.0	0.6	0.4	
	0.7	0.6	1.0	0.0	
	0.5	0.4	0.0	1.0	

We can build the  $m \times m$  similarity matrix.

Doable for  $m$  upto 100K or so – more if we can leverage sparsity in  $X$ .

# Item-based Collaborative Filtering

For user  $i$ , recommend the items which have the highest values of:

$$score(i, k) = \sum_{j=1}^m x_{i,j} sim(j, k)$$

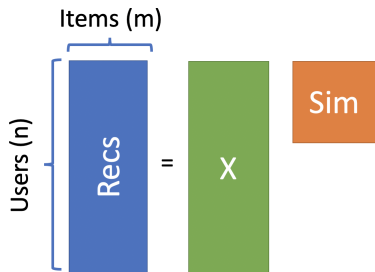
i.e., the items most similar to those that the user interacted with.

# Item-based Collaborative Filtering

For user  $i$ , recommend the items which have the highest values of:

$$score(i, k) = \sum_{j=1}^m x_{i,j} sim(j, k)$$

i.e., the items most similar to those that the user interacted with.



Computing the scores is just a big (sparse) matrix multiply.

# Item-based Collaborative Filtering

Tripadvisor destination recommender “most similar items:”

Miami	Paris	NYC	Turks & Caicos
Fort Lauderdale	Lyon	Washington DC	Providenciales
Key West	Nice	Boston	Seven Mile Beach
Boca Raton	Brussels	Philadelphia	Grand Cayman
West Palm Beach	Amsterdam	Los Angeles	Paradise Island
Key Largo	Venice	San Francisco	New Providence Island
Orlando	Bruges	Miami Beach	Palm - Eagle Beach
Pompano Beach	Milan	Miami	Aruba
Tampa	Florence	Chicago	Negril
Islamorada	Munich	Toronto	Montego Bay
Palm Beach	Barcelona	Montreal	Maui

# Item-based Collaborative Filtering

Tripadvisor recommendation example:

## Views

- (2017-05-20) Minneapolis (Minnesota)
- (2017-05-19) Orlando (Florida)
- (2017-04-14) Des Moines (Iowa) ...

## Recommendations

- (0.22) Saint Paul (Minnesota)
- (0.21) Bloomington (Minnesota)
- (0.19) Kissimmee (Florida)
- (0.12) Duluth (Minnesota)
- (0.12) Edina (Minnesota)
- (0.11) Maple Grove (Minnesota)
- (0.10) Richfield (Minnesota)
- (0.10) Miami (Florida)
- (0.10) Saint Louis Park (Minnesota)
- (0.09) Minnetonka (Minnesota)

Where final score =  
recency of view \*  
similarity to view

# Item-based Collaborative Filtering

- Works well when number of items is small (since you have to compute the similarity matrix).
- Very simple to implement.
- Easy to explain recommendations (“because you liked item XXX and YYY”).

# User-based Collaborative Filtering






- Similar idea (less popular in practise).
- Basic idea:
  - Compute similarity score between users (e.g., cosine similarity)
  - Find nearest neighbors to each user
  - Recommend the most popular items among the nearest neighbors.

# Matrix Factorization

- **Matrix Factorization:** powerful technique for recommender systems (e.g., netflix prize winner).
- Basic idea:
  - Assume a set of vectors  $u_i \in \mathbb{R}^d$  for user, and  $v_j \in \mathbb{R}^d$  for each item.
  - The score for how well user  $i$  likes item  $j$  given by  $u_i^T v_j$ .
  - Fit the vectors to minimize error on the observed ratings.
  - Use them to predict the unseen ratings.



# Matrix Factorization

User Prefs.		Sci fi	Old school	comedy	Item features				
		0.8	0.9	0.6	-0.5	-0.9			
		0.9	0.7	1.0	0.3	-0.6			
		-0.6	-0.9	-0.5	0.7	0.3			
Sci fi							...		
0.5	0.8 -0.3	Alice	★★★★★	★★★★☆	★★★★☆	?	?		
0.9	-0.1 1.5	Bob	★★★★☆	★★★★★	?	★★★★☆	?		
0.0	-0.3 0.8	Clair	?	?	?	★★★★☆	★★★★☆		
-0.3	-0.3 0.8	Dave	?	★★★★☆	?	?	★★★★★		
		⋮						⋮	

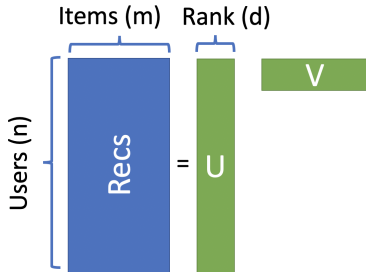
Model is conceptually plausible.

However we won't know what the dimensions "mean."

# Matrix Factorization

For user  $i$ , recommend the items which have the highest values of:

$$\text{score}(i,j) = u_i^T v_j$$



Computing the scores is a big dense matrix product.

Later: approximation methods.

# Matrix Factorization

**Input**  $X \in \mathbb{R}^{n \times m}$  with set of weights  $W$  (e.g., observed entries have weight 1, unobserved entries have weight  $\epsilon$ ).

**Goal:** find matrices  $U, V$  with rank  $d$  which minimize the error:

$$\sum_{i,j} w_{i,j} (x_{i,j} - u_i^T v_j)^2$$

i.e, weight observed values higher than unobserved values (zeros).

note: related to **matrix completion** which tries to fill in the matrix directly rather than via  $U, V$ .

# Matrix Factorization

**Simple Approach:** compute SVD truncated to rank  $d$ :  $X \approx L\Sigma R^T$ .

Let

$$U = L\Sigma^{1/2}, \quad V = R\Sigma^{1/2}$$

i.e., factors just given by first  $d$  singular vectors.

# Matrix Factorization

**Simple Approach:** compute SVD truncated to rank  $d$ :  $X \approx L\Sigma R^T$ .

Let

$$U = L\Sigma^{1/2}, \quad V = R\Sigma^{1/2}$$

i.e., factors just given by first  $d$  singular vectors.

- Pros: trivial operation (can approximate the SVD). Actually works ok in practice.
- Cons: fitting optimizes the error assuming  $w_{i,j} = 1$  – i.e., we treated the zeros as ratings rather than unknown entries.

# Matrix Factorization

**Alternating Least Squares (ALS):** alternate between  $U$  and  $V$  and optimize the objective:

$$\sum_{i,j} w_{i,j} (x_{i,j} - u_i^T v_j)^2$$

- Initialize  $V$  with some random noise (e.g.,  $\mathcal{N}(0, 1)$ ).
- Iterate until convergence:
  - Hold  $V$  fixed and compute  $U$  to minimize the objective.
  - Hold  $U$  fixed and compute  $V$  to minimize the objective.

Each step will decrease the objective function until we get to a local minimum.

# Matrix Factorization

**Update rule** (differentiate objective and set equal to zero):

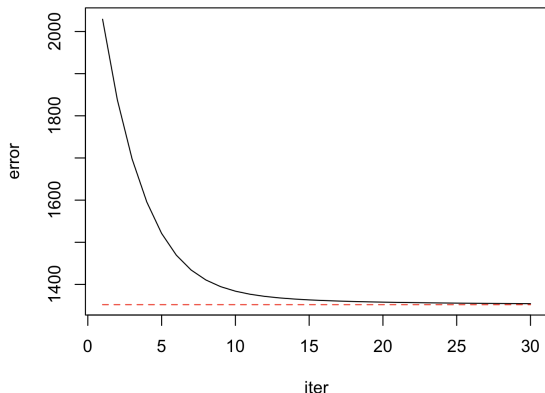
$$u_i = \left( \sum_j w_{i,j} v_j v_j^T \right)^{-1} \sum_j w_{i,j} x_{i,j} v_j$$

$$v_j = \left( \sum_i w_{i,j} u_i u_i^T \right)^{-1} \sum_i w_{i,j} x_{i,j} u_i$$

Very similar to the form of the ordinary least squares estimator.

Computation is easy to parallelize since each  $u_i$  can be found separately from the others.

# Matrix Factorization

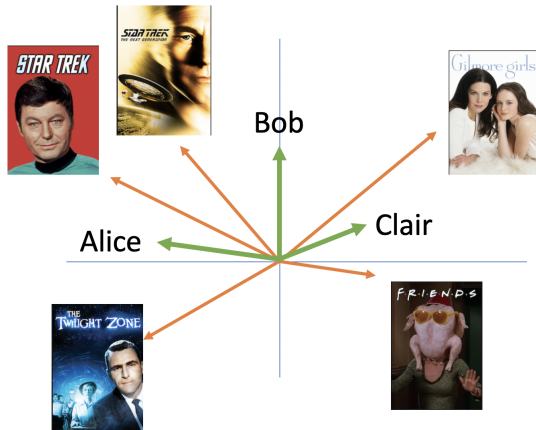


Using ALS to approximate the best low rank approximation to a matrix (where  $w_{i,j} = 1$ ).

Compared to SVD (the best possible).



# Matrix Factorization



End up with an embedding of items and users into  $\mathbb{R}^d$ .

Score = dot product  
= related to the angle  
between the user and  
item.

# Matrix Factorization

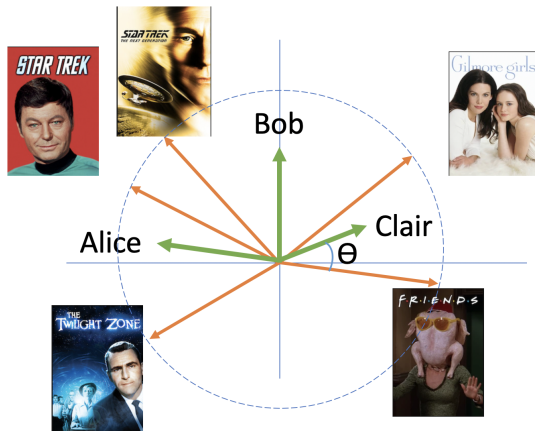
Computing the recommendations =  $n \times m$  dot products.

We cant do this when  $n, m$  are large (millions or more).

Possible solutions:

- Try to do something hierarchical (recommend a category, then recommend items in the category).
- Use an approximation scheme to compute approximate recommendations.

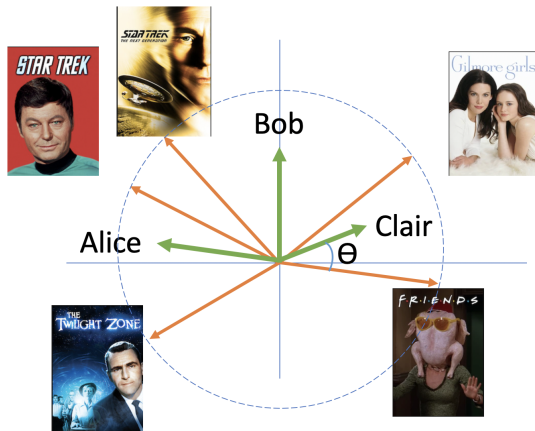
# Matrix Factorization



Can augment the embedding so the items all have unit norm, without affecting the scores (add a dimension)

Then we have an embedding where  
score = dot product  
 $= \|u_i\| \cos(\theta)$ .

# Matrix Factorization



Reduced the recommendation problem to finding nearest neighbors under the cosine similarity.

Can approximate this using Locality Sensitive Hashing (LSH).

# Matrix Factorization

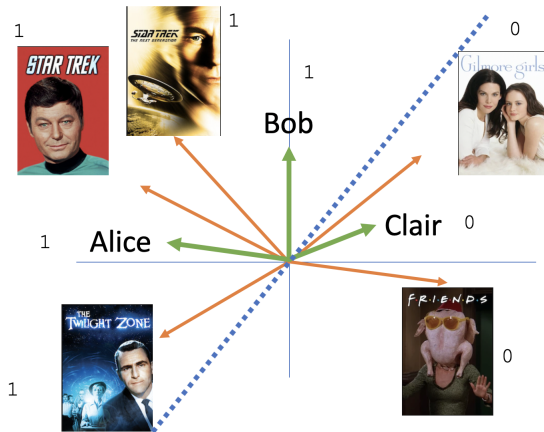
**Locality Sensitive Hashing:** set up a hash function  $H(v)$  so for a pair of vectors  $u, v$ :

$P[H(v) = H(u)]$  is small when  $u, v$  are far apart.

$P[H(v) = H(u)]$  is large when  $u, v$  are close together.

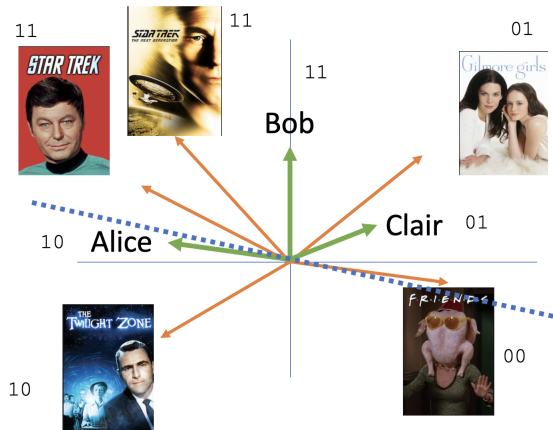
Then we can approximate nearest-neighbors by hashing all the vectors and comparing each vector to the others in the same hash bucket.

# Matrix Factorization



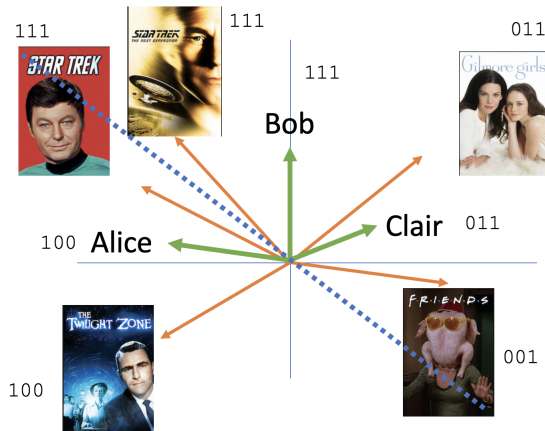
**LSH idea:** generate a hash code  $H(v)$ , where each bit  $H_i(v) = \text{sign}(v^T z_i)$  where  $z_i$  is a random direction (e.g., gaussian).

# Matrix Factorization



**LSH idea:** generate a hash code  $H(v)$ , where each bit  $H_i(v) = \text{sign}(v^T z_i)$  where  $z_i$  is a random direction (e.g., gaussian).

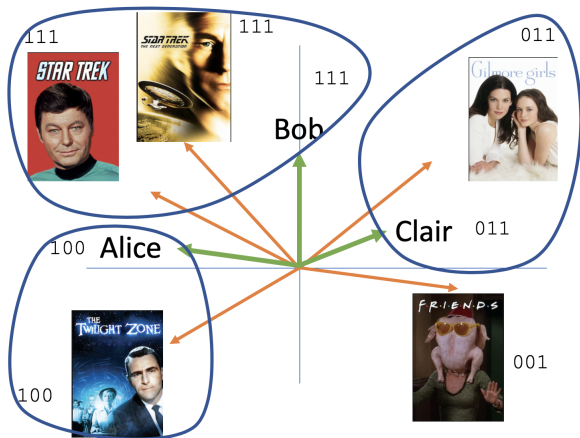
# Matrix Factorization



**LSH idea:** generate a hash code  $H(v)$ , where each bit  $H_i(v) = \text{sign}(v^T z_i)$  where  $z_i$  is a random direction (e.g., gaussian).



# Matrix Factorization



Then only compare users to the items that end up in the same hash bucket as them.

# Matrix Factorization

Does LSH work?

Say  $u^T v = \|u\| \|v\| \cos(\theta)$  then:

$$P(H_i(u) \neq H_i(v)) = \frac{\theta}{180}$$

So for  $b$  bits:

$$P(H(u) = H(v)) = \left(1 - \frac{\theta}{180}\right)^b$$

i.e., probability vanishes quickly when  $\theta$  is larger.

To improve recall, we can just repeat the procedure a few times.

- Recommender system setup / CF heuristics.
- Matrix factorization based methods / Alternating Least Squares.
- Approximating recs / KNN using locality sensitive hashing.