Recommender Systems

Introduction

Introduction

Today, we look at a topic that has become enormously important: recommender systems.

In Part I, we will:

- Define recommender systems
- Review the challenges they pose
- Discuss two classic methods:
  - Collaborative Filtering
  - Matrix Factorization

In Part II, we will:

- Delve into more recent deep learning methods.

This section draws heavily on

- These slides by Alex Smola
- *Matrix Factorization Techniques for Recommender Systems*, (Koren et al. 2009)
- *Collaborative Filtering with Temporal Dynamics*, (Koren 2009)

What are Recommender Systems?

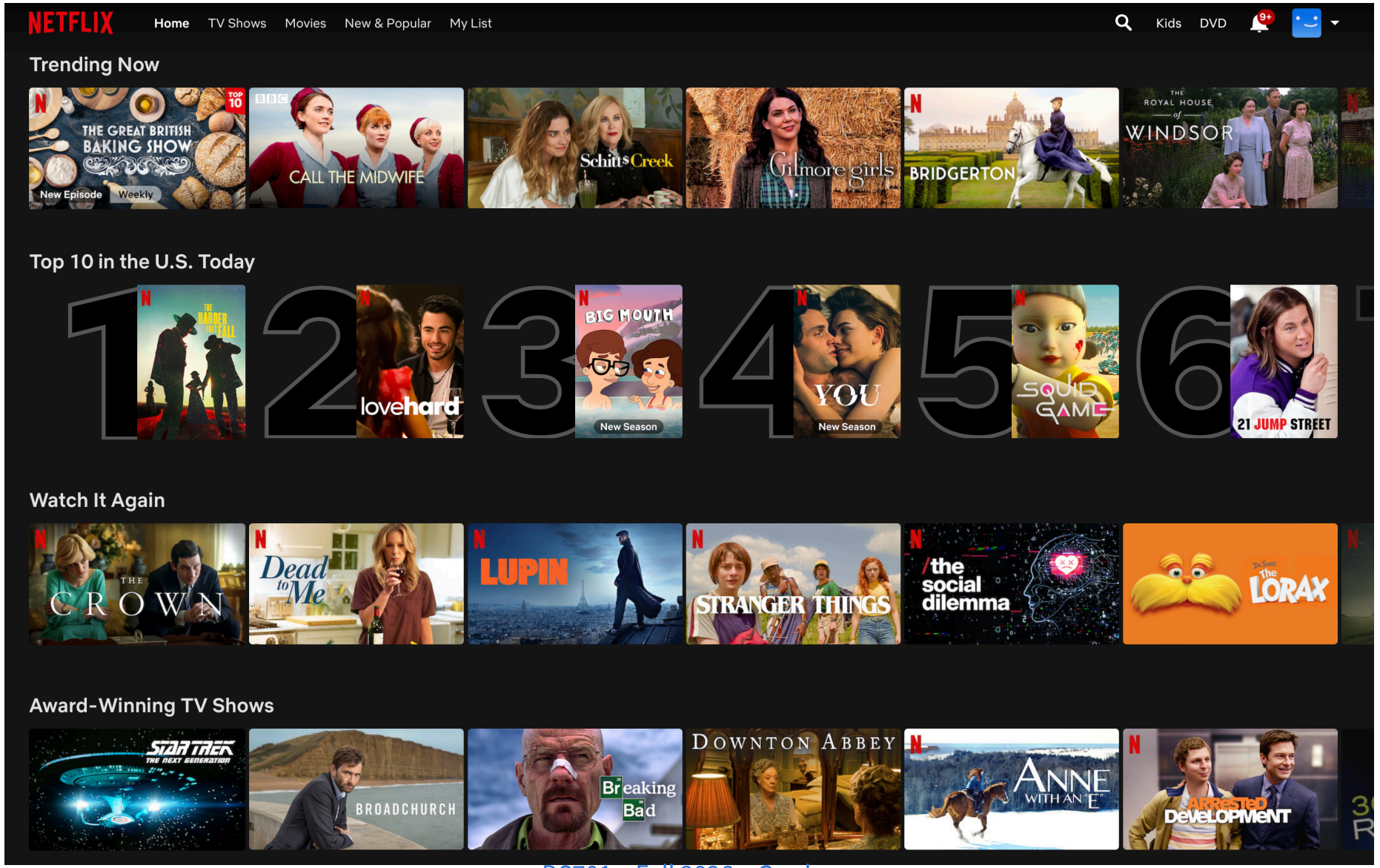
The concept of recommender systems emerged in the late 1990s / early 2000s as social life moved online:

- online purchasing and commerce
- online discussions and ratings
- social information sharing

In these systems the amount of content was exploding and users were having a hard time finding things they were interested in.

> Users wanted recommendations.

Over time, the problem has only gotten worse:



Find the perfect gift Shop Holiday Gifts


Early Black Friday


Holiday Prep Shop


Holiday Toy List


Electronic Gifts


Fashion Gifts


Beauty Gifts


Home Gifts

Holiday gifts, at your finger tips


Holiday Toy List


Electronics Gifts


Home Gifts


Fashion Gifts

Shop all gifts

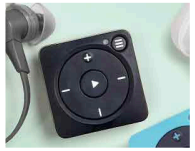
Review your purchase

See more products to review

Start your gifting early


Gift guide


Shop deals


Tech gifts


Shop by price

See more from Amazon Launchpad

Tech under $100


Fitbit Inspire 2 Health…

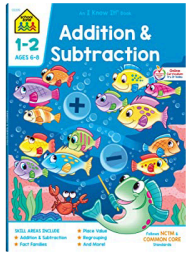
Sony SRS-XB13 Extra B…

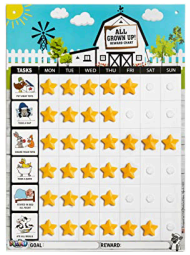
Tile Pro (2020) 1-pack -…


Satechi Aluminum Stan…

Shop gifts under $100

Small Business products related to items you viewed





An enormous need has emerged for systems to help sort through products, services, and content items.

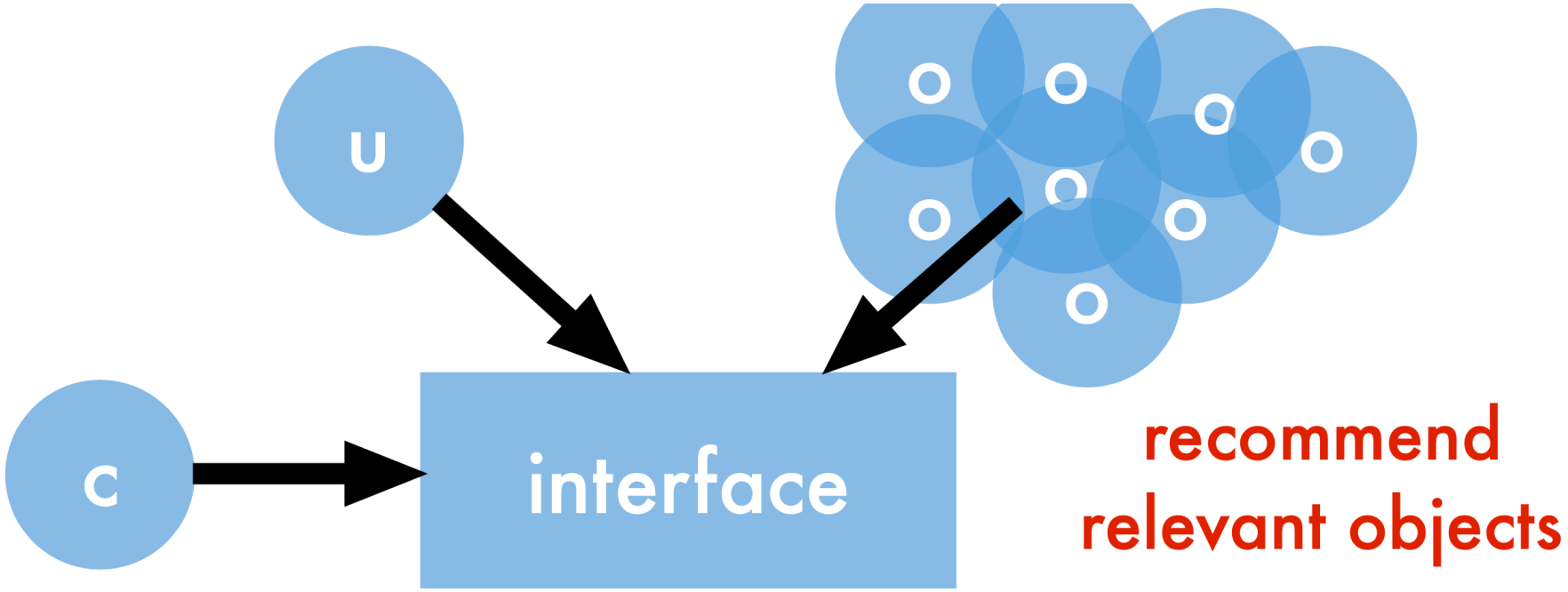
This often goes by the term **personalization.**

Some examples (as of Fall 2024):

- Movie recommendation (Netflix ~6.5K movies and shows, YouTube ~14B videos)
- Related product recommendation (Amazon ~600M products)
- Web page ranking (Google >>100B pages)
- Social content filtering (Facebook, Twitter)
- Services (Airbnb, Uber, TripAdvisor)
- News content recommendation (Apple News)
- Priority inbox & spam filtering (Gmail)
- Online dating (Match.com)

A more formal view:

- User - requests content
- Objects - instances of content
- Context - ratings, purchases, views, device, location, time, history
- Interface - browser, mobile



Inferring Preferences

Unfortunately, users generally have a hard time **explaining** what types of content they prefer.

Some early systems worked by interviewing users to ask what they liked. Those systems did not work very well.

A very interesting article about the earliest personalization systems is User Modeling via Stereotypes by Elaine Rich, dating from 1979.

Instead, modern systems work by capturing user's opinions about **specific** items.

This can be done actively:

- When a user is asked to **rate** a movie, product, or experience,

or it can be done passively:

- By noting which items a user **chooses** to purchase (for example).

Training data

user	movie	date	score
1	21	5/7/02	1
1	213	8/2/04	5
2	345	3/6/01	4
2	123	5/1/05	4
2	768	7/15/02	3
3	76	1/22/01	5
4	45	8/3/00	4
5	568	9/10/05	1
5	342	3/5/03	2
5	234	12/28/00	2
6	76	8/11/02	5
6	56	6/15/03	4

Test data

user	movie	date	score
1	62	1/6/05	?
1	96	9/13/04	?
2	7	8/18/05	?
2	3	11/22/05	?
3	47	6/13/02	?
3	15	8/12/01	?
4	41	9/1/00	?
4	28	8/27/05	?
5	93	4/4/05	?
5	74	7/16/03	?
6	69	2/14/04	?
6	83	10/3/03	?

Challenges

- The biggest issue is **scalability**: typical data for this problem is huge.
  - Millions of objects
  - 100s of millions of users
- Changing user base
- Changing inventory (movies, stories, goods)
- Available features
- Imbalanced dataset
  - User activity / item reviews are power law distributed

This data is a subset of the data presented in: “From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews,” by J. McAuley and J. Leskovec. WWW, 2013

Example

Let’s look at a dataset for testing recommender systems consisting of Amazon movie reviews:

We’ll download a compressed pickle file containing the data if it is not already present.

▶ Code

We’ll load the data into a pandas DataFrame.

▶ Code

```
Elapsed read time: 2.39 seconds
```

Run `df.info()` to see the column names and data types.

▶ Code

```
<class 'pandas.DataFrame'>
RangeIndex: 1697533 entries, 0 to 1697532
Data columns (total 9 columns):
 #   Column                  Dtype
---  ------                  -----
 0   Id                      int64
 1   ProductId               object
 2   UserId                  object
 3   HelpfulnessNumerator    int64
 4   HelpfulnessDenominator  int64
 5   Score                   float64
 6   Time                    int64
 7   Summary                 object
 8   Text                    object
dtypes: float64(1), int64(4), object(4)
memory usage: 116.6+ MB
```

where

- HelpfulnessNumerator: The number of users who found the review helpful (the “yes” votes)
- HelpfulnessDenominator: The total number of users who voted on whether the review was helpful (both “yes” and “no” votes)

Now we can count the number of users and movies:

▶ Code

There are:

- 1,697,533 reviews
- 50,052 movies
- 123,960 users

There are 6,204,445,920 potential reviews, meaning sparsity of 0.0274%

where

$$\text{sparsity} = \frac{\#\text{ of reviews}}{\#\text{ of users} \times \#\text{ of movies}} = \frac{\#\text{ of reviews}}{\#\text{ of potential reviews}}$$

Reviews are Sparse

Only 0.02% of the reviews are available – 99.98% of the reviews are missing.

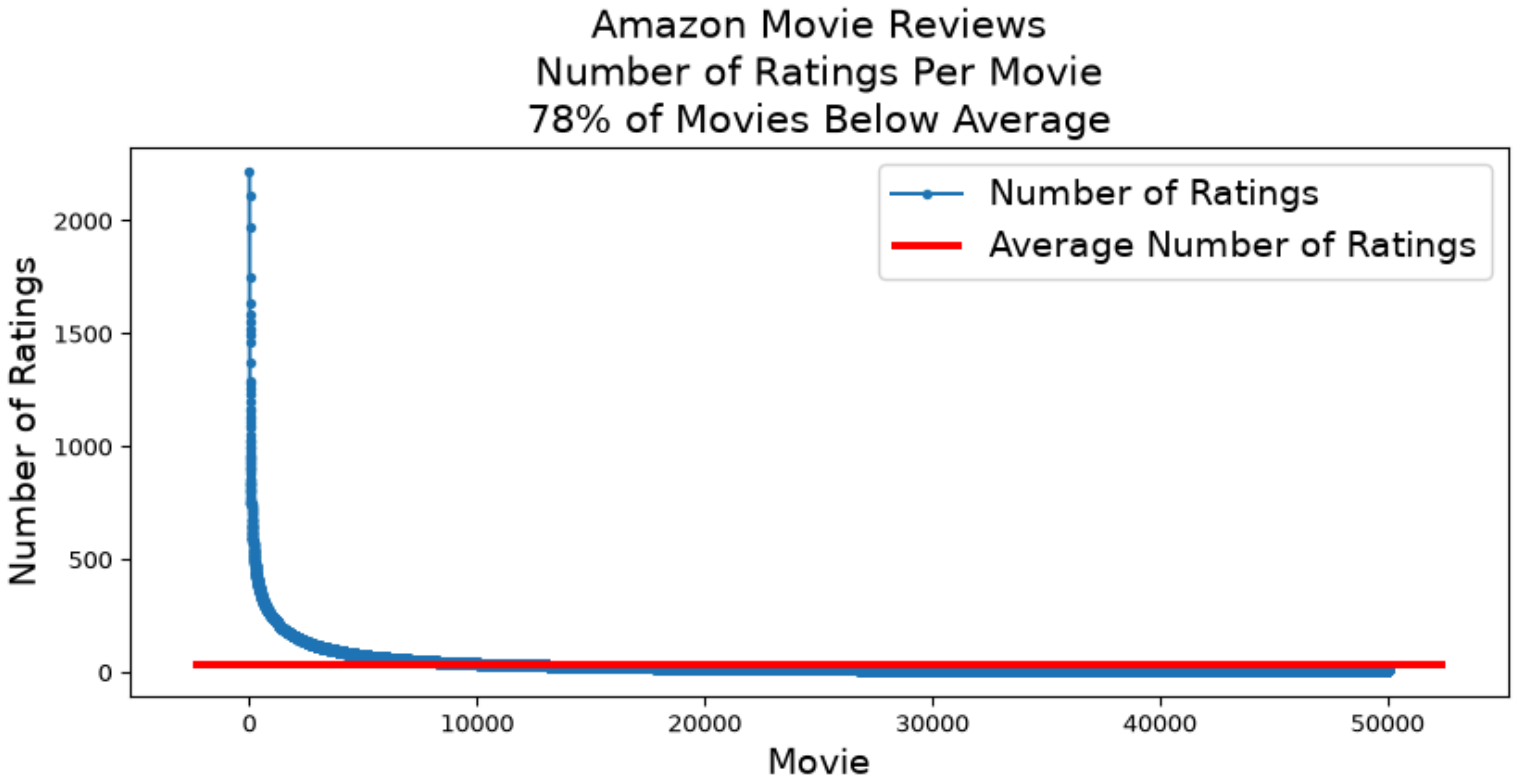
▶ Code

There are on average 33.9 reviews per movie and 13.7 reviews per user

Sparseness is Skewed

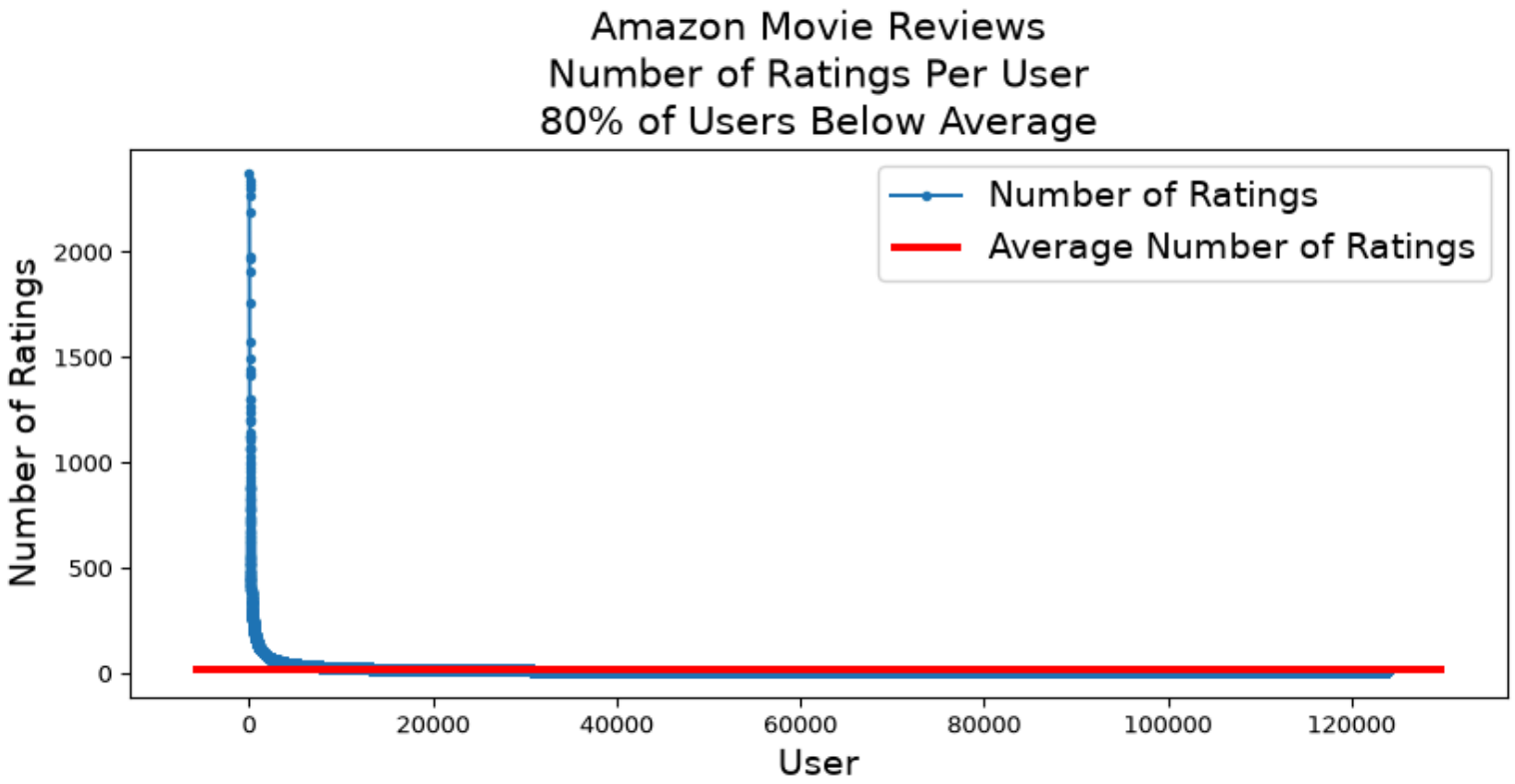
Although on average a movie receives 34 reviews, **almost all movies have even fewer reviews.**

▶ Code



Likewise, although the average user writes 14 reviews, almost all users write even fewer reviews.

▶ Code



Objective Function

Ultimately, our goal is to predict the rating that a user would give to an item.

For that, we need to define a loss or objective function.

A typical objective function is root mean square error (RMSE)

$$\text{RMSE} = \sqrt{\frac{1}{|S|} \sum_{(i,u) \in S} (\hat{r}_{ui} - r_{ui})^2},$$

where

- r_{ui} is the rating that user u gives to item i, and
- S is the subset of items that have ratings.

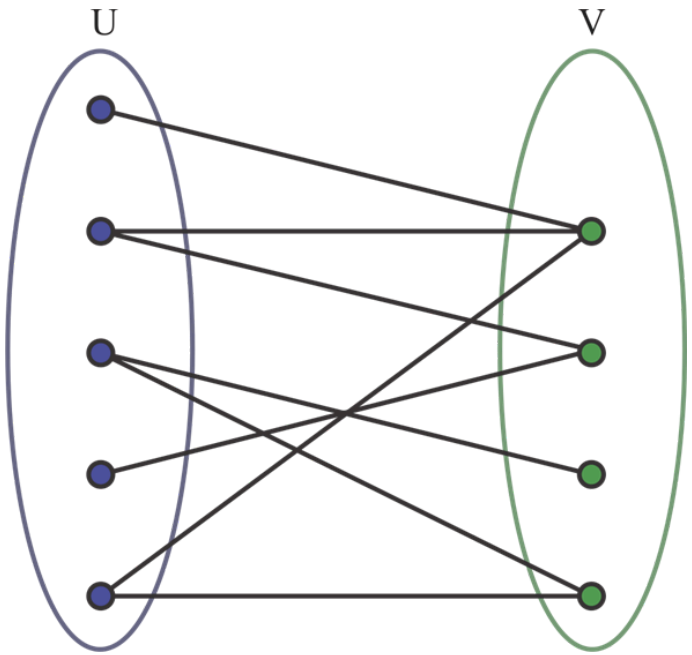
OK, now we know the problem and the data available. How can we address the problem?

The earliest method developed is called **collaborative filtering.**

Colaborative Filtering

Collaborative Filtering

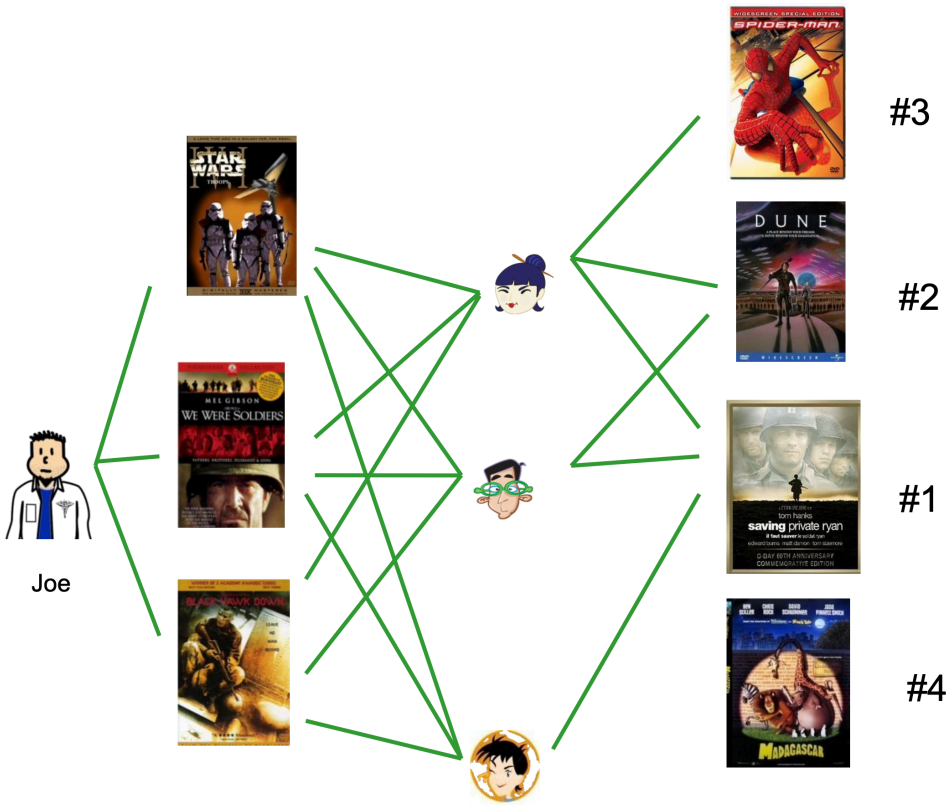
The central idea of collaborative filtering is that the set of known recommendations can be considered to be a **bipartite graph.**



The nodes of the bipartite graph are **users** (U) and **items** (V).

Each edge corresponds to a known rating r_{ui}.

Then recommendations are formed by traversing or processing the bipartite graph.



There are at least two ways this graph can be used.

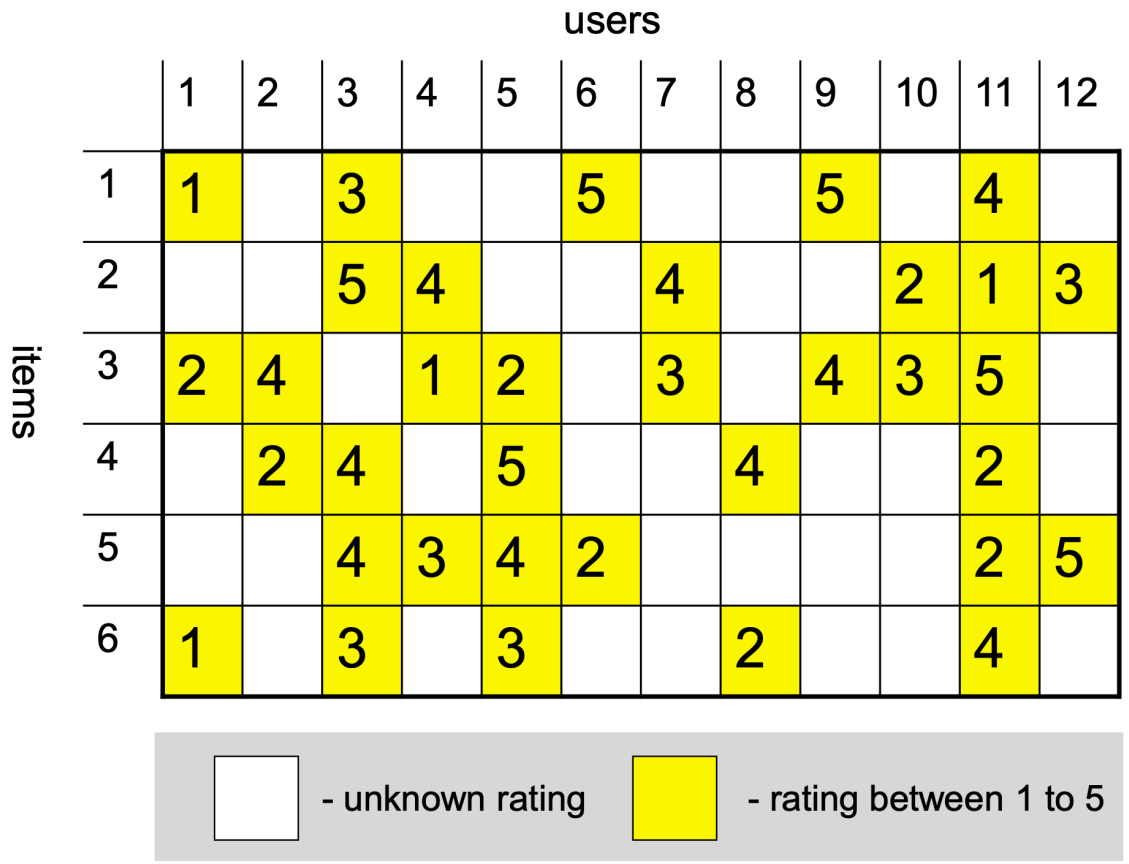
Two ways to form a rating for item (u, i):

Item-Item CF

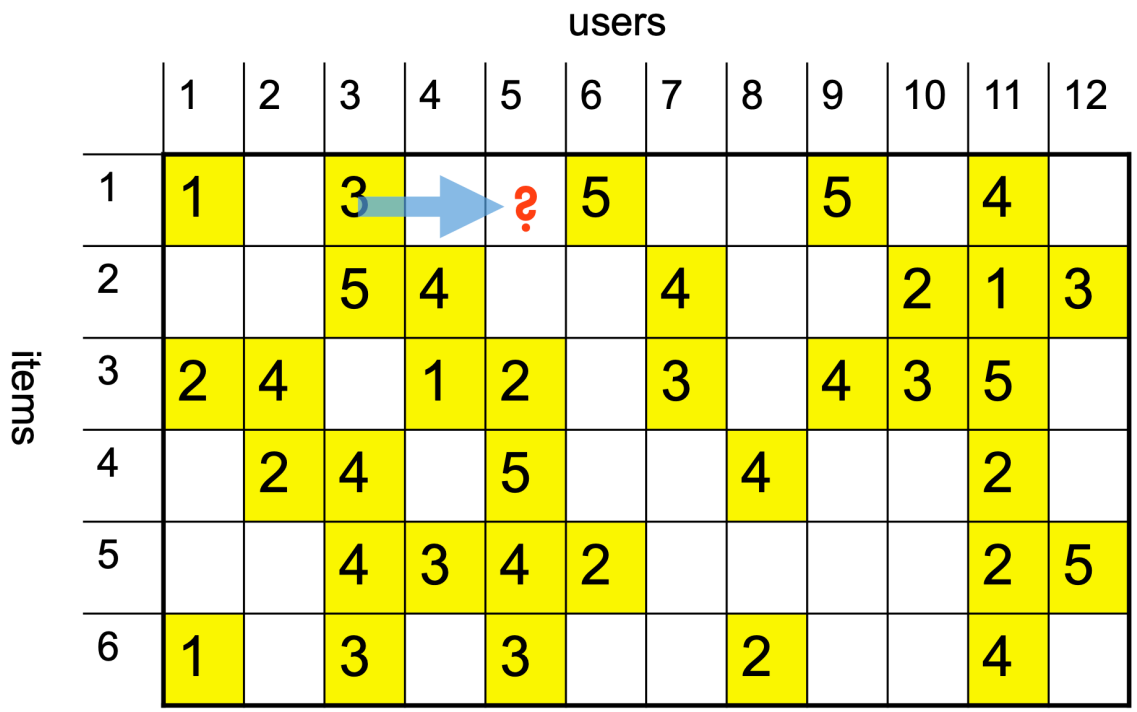
For item-item similarity, we'll look at **item-item Collaborative Filtering (CF).**

The questions are:

Here is another view of the ratings graph, this time as a matrix that includes missing entries:

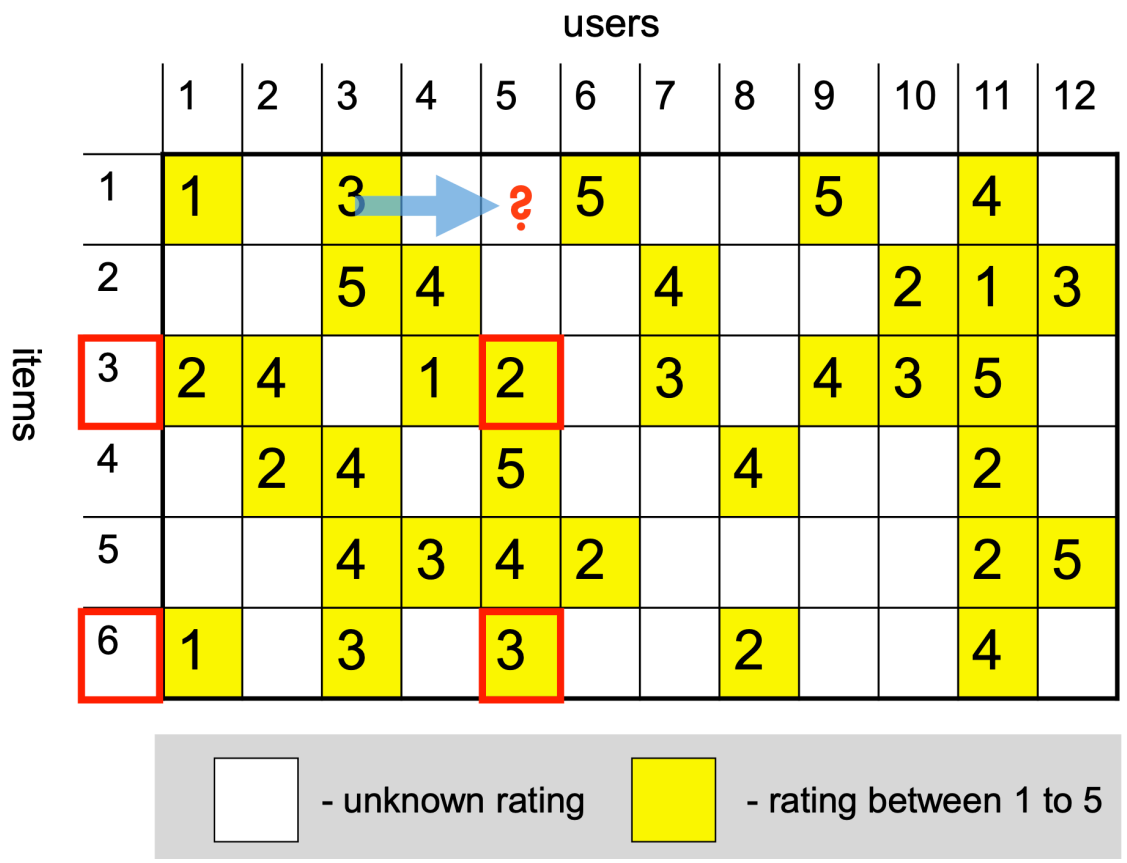


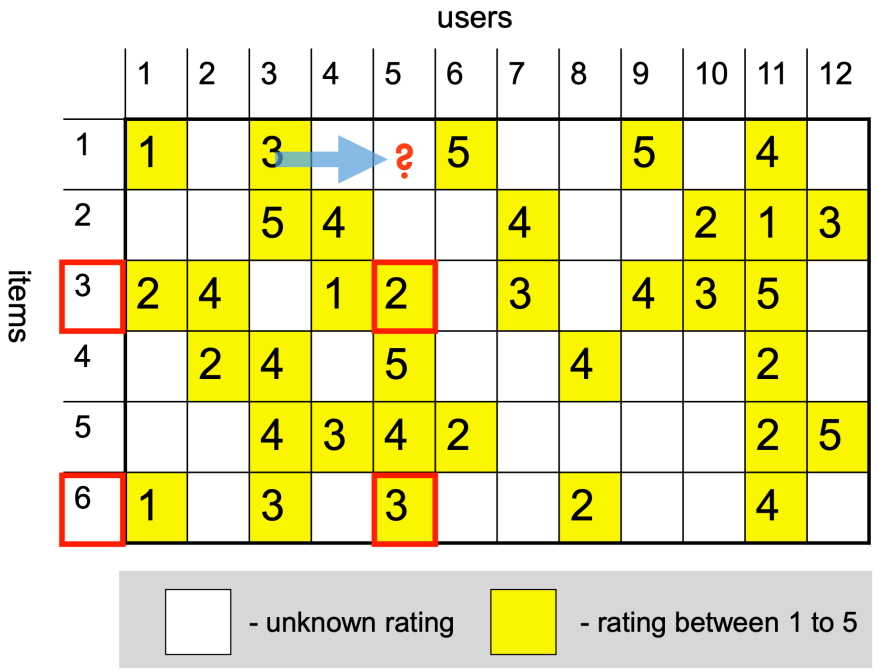
Let’s say we want to predict the value of this unknown rating:



We'll consider two other items, namely items 3 and 6 (for example).

Note that we are only interested in items that this user has rated.





We will discuss strategies for assessing similarity shortly.

How did we choose these two items?

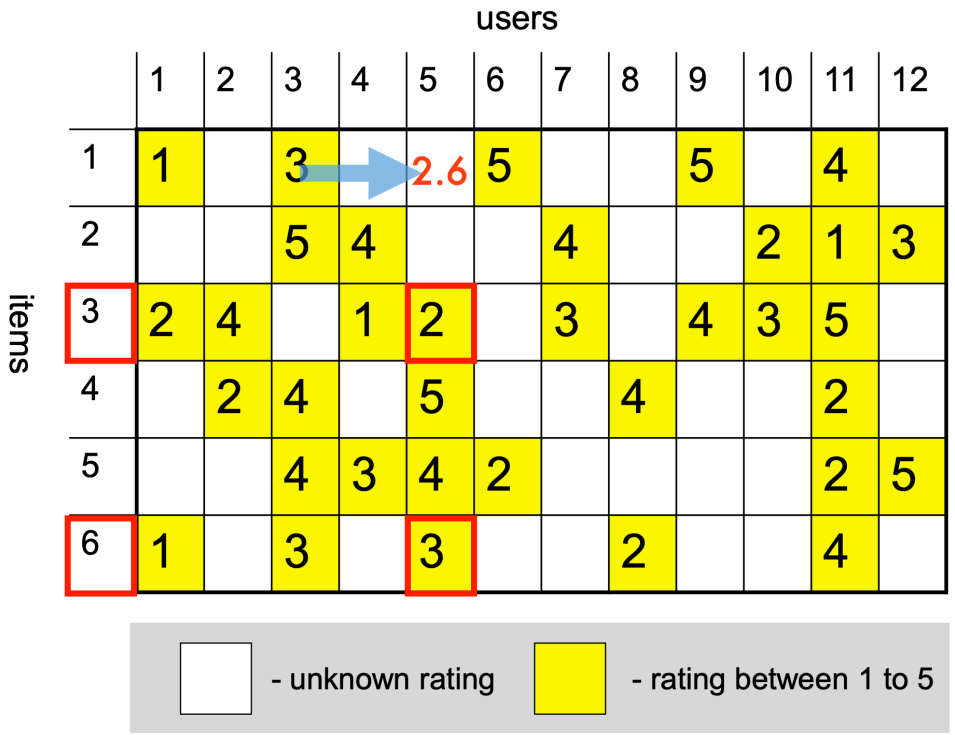
We used k-**nearest neighbors**. Here $k = 2$.

For now, let's just say we determine the similarities as:

$$s_{13} = 0.2$$

$$s_{16} = 0.3$$

These similarity scores tell us how much weight to put on the rating of the other items.



So we can form a prediction of $\hat{r}_{15}$ as:

$$\hat{r}_{15} = \frac{s_{13} \cdot r_{35} + s_{16} \cdot r_{65}}{s_{13} + s_{16}} = \frac{0.2 \cdot 2 + 0.3 \cdot 3}{0.2 + 0.3} = 2.6$$

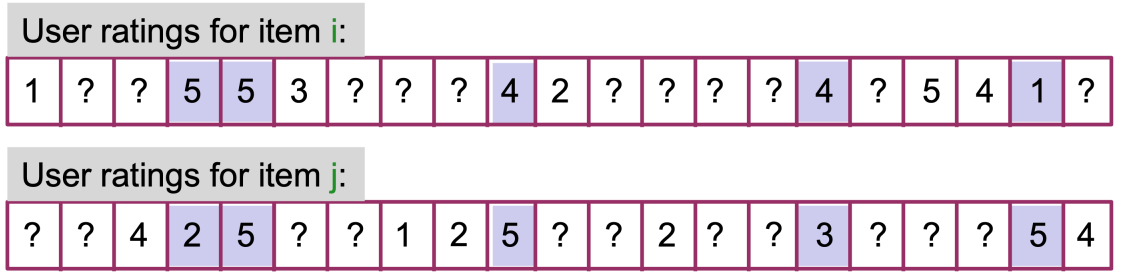
Similarity

How should we assess similarity of items?

A reasonable approach is to consider items similar if their ratings are **correlated:**, i.e.,

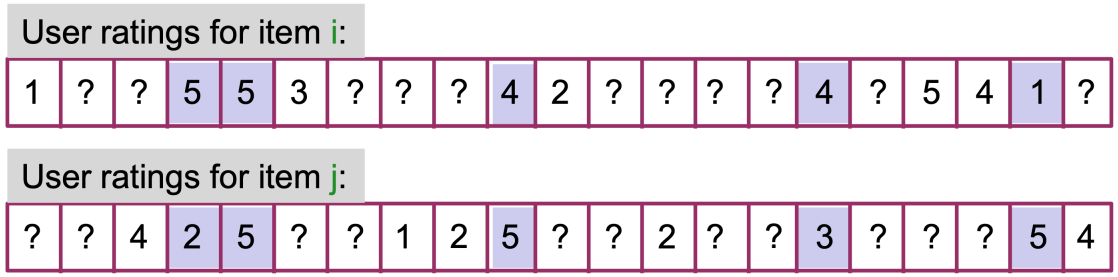
$$\rho(X, Y) = \frac{E\left[(X - \mu_X)(Y - \mu_Y)\right]}{\sigma_X \sigma_Y}.$$

However, note that two items will not have ratings in the same positions.



So we want to compute correlation only over the users who rated both the items.

Example



Let's put the ratings in python lists:

▶ Code

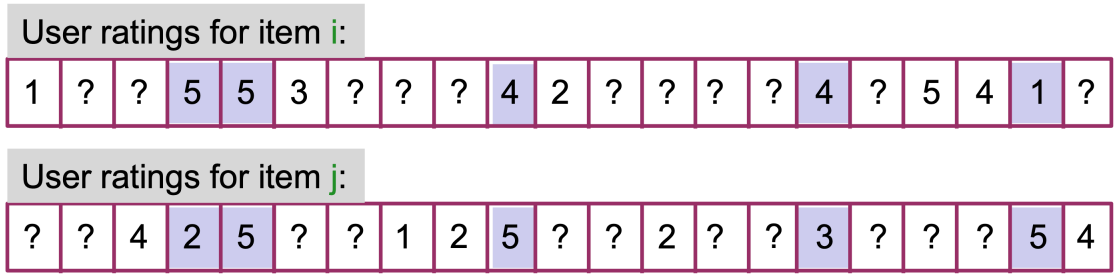
Ratings for item i:

[1, nan, nan, 5, 5, 3, nan, nan, nan, 4, 2, nan, nan, nan, nan, 4, nan, 5, 4, 1, nan]

Ratings for item j:

[nan, nan, 4, 2, 5, nan, nan, 1, 2, 5, nan, nan, 2, nan, nan, 3, nan, nan, nan, 5, 4]

Example, continued



Let's drop the non-common ratings:

▶ Code

Common ratings for item i: [5, 5, 4, 4, 1]

Common ratings for item j: [2, 5, 5, 3, 5]

Example, continued

▶ Code

Common ratings for item i: [5, 5, 4, 4, 1]

Common ratings for item j: [2, 5, 5, 3, 5]

Now we can compute the Pearson correlation coefficient:

▶ Code

Pearson correlation coefficient: -0.43

Which is a *moderate negative correlation*, meaning that as item i gets rated higher, item j gets rated lower.

Similarity for Binary Data

In some cases we will need to work with binary r_{ui}.

For example, purchase histories on an e-commerce site, or clicks on an ad.

In this case, an appropriate replacement for Pearson r is the **Jaccard similarity coefficient** or **Intersection over Union**.

$$J_{Sim}(\mathbf{x}, \mathbf{y}) = \frac{|\mathbf{x} \cap \mathbf{y}|}{|\mathbf{x} \cup \mathbf{y}|}.$$

See the lecture on similarity measures.

Improving CF in the presence of bias

One problem with the story so far arises due to **bias**.

- Some items are significantly higher or lower rated
- Some users rate substantially higher or lower in general

These properties interfere with similarity assessment.

Bias correction is crucial for CF recommender systems.

We need to include

- Per-user offset
- Per-item offset
- Global offset

Representing biases

Hence we need to form a per-item bias of:

$$b_{ui} = \mu + \alpha_u + \beta_i$$

where

- b_{ui} is the bias of user u for item i.
- μ is the global average rating across all items and users.
- α_u is the offset of user u and
- β_i is the offset of item i.

If we gather all these elements together we can form:

- $\boldsymbol{\alpha}$ an $n \times 1$ vector of per-user offsets, and
- $\boldsymbol{\beta}$ an $m \times 1$ vector of per-item offsets.

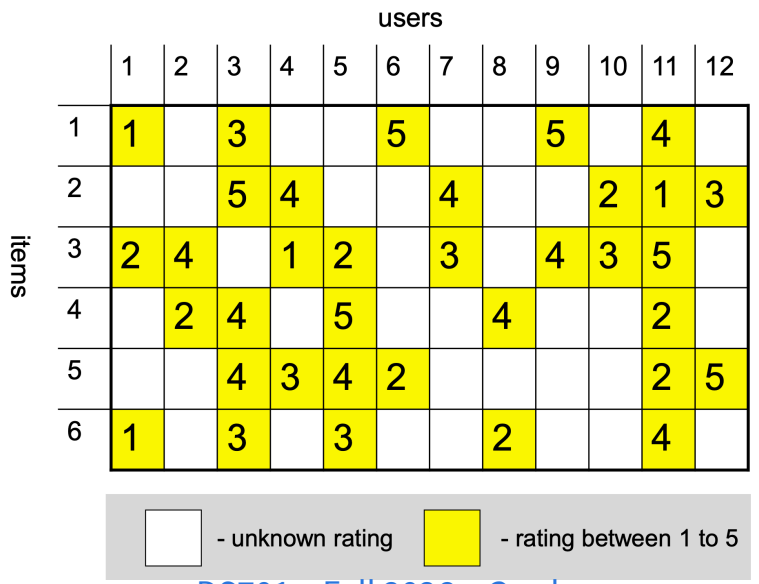
Estimating biases

$$b_{ui} = \mu + \alpha_u + \beta_i$$

How can we estimate the parameters α, β, and μ?

Let's assume for a minute that we had a fully-dense matrix of ratings R.

Recall that each of the m rows of R represents an item and each of the n columns represents a user.



Estimating biases, continued

One way to do this is to minimize the squared error between the ratings and the biases:

$$\min_{\boldsymbol{\alpha},\boldsymbol{\beta},\mu} \|R - \mathbf{1}\boldsymbol{\alpha}^T + \boldsymbol{\beta}\mathbf{1}^T + \mu 1\|^2 + \lambda(\|\boldsymbol{\alpha}\|^2 + \|\boldsymbol{\beta}\|^2).$$

and include a regularization term to minimize the magnitude of the biases.

- Here, bold-faced $\mathbf{1}$ represents appropriately sized vectors of ones, and non-boldfaced 1 is an $m \times n$ matrix of ones.
- So $\mathbf{1}\boldsymbol{\alpha}^T$ is an $m \times n$ matrix where each row is the bias for a user.
- Similarly, $\boldsymbol{\beta}\mathbf{1}^T$ is an $m \times n$ matrix where each column is the bias for an item.
- And $\mu 1$ is an $m \times n$ matrix where each element is μ.

Estimating biases, continued

$$\min_{\alpha,\beta,\mu} \|R - \mathbf{1}\alpha^T + \beta\mathbf{1}^T + \mu 1\|^2 + \lambda(\|\alpha\|^2 + \|\beta\|^2)$$

While this is not a simple ordinary least squares problem, there is a strategy for solving it.

Assume we hold $\beta\mathbf{1}^T$ and $\mu 1$ constant.

Then the remaining problem is

$$\min_{\alpha} \|R - \mathbf{1}\alpha^T\|^2 + \lambda\|\alpha\|^2,$$

which (for each column of R) is a standard regularized least squares problem solved via Ridge regression.

Aside: Ridge Regression

Ridge Regression is a regularized least squares method that adds a penalty term to prevent overfitting.

Standard Least Squares:

$$\min_{\beta} \|X\beta - \mathbf{y}\|_2^2$$

Ridge Regression:

$$\min_{\beta} \|X\beta - \mathbf{y}\|_2^2 + c\|\beta\|_2^2$$

The penalty term $c\|\beta\|_2^2$ shrinks the coefficients β towards zero.

Why Ridge Regression?

- Addresses **multicollinearity** (when features are highly correlated)
- Prevents coefficient magnitudes from becoming too large
- Always has a unique solution: $\hat{\boldsymbol{\beta}} = (X^T X + cI)^{-1} X^T \mathbf{y}$

The hyperparameter c:

- When $c = 0$: ordinary least squares
- When $c \to \infty$: all coefficients shrink to zero
- Choose c via cross-validation to balance fitting vs. regularization

Back to Solving the problem

This sort of problem is called **jointly convex** in that it is convex in each of the variables α, β, and μ.

The strategy for solving is:

1. Hold α and β constant, solve for μ.
2. Hold α and μ constant, solve for β.
3. Hold β and μ constant, solve for α.

Each of the three steps will reduce the overall error. As a result, we iterate over them until convergence.

The last issue is that the matrix R is not dense - in reality we only have a small subset of its entries.

We simply need to adapt the least-squares solution to only consider the entries in R that we know.

As a result, the actual calculation is as follows…

Step 1:

$$\mu = \frac{\sum_{(u,i)\in R}(r_{ui} - \alpha_u - \beta_i)}{|R|}$$

Step 2:

$$\alpha_u = \frac{\sum_{i\in R(u)}(r_{ui} - \mu - \beta_i)}{\lambda + |R(u)|}$$

Step 3:

$$\beta_i = \frac{\sum_{u\in R(i)}(r_{ui} - \mu - \alpha_u)}{\lambda + |R(i)|}$$

Step 4: If not converged, go to Step 1.

Here $i \in R(u)$ means the set of items rated by user u and $u \in R(i)$ means the set of users who have rated item i and $|R(u)|$ is the number of ratings.

Now that we have learned the biases, we can do a better job of estimating correlation:

$$\hat{\rho}_{ij} = \frac{\sum_{u \in U(i,j)} (r_{ui} - b_{ui})(r_{uj} - b_{uj})}{\sqrt{\sum_{u \in U(i,j)} (r_{ui} - b_{ui})^2 \sum_{u \in U(i,j)} (r_{uj} - b_{uj})^2}},$$

where

- $b_{ui} = \mu + \alpha_u + \beta_i$, and
- $U(i, j)$ are the users who have rated both i and j.

And using biases we can also do a better job of estimating ratings:

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{j \in n_k(i,u)} s_{ij}(r_{uj} - b_{uj})}{\sum_{j \in n_k(i,u)} s_{ij}},$$

where

- $b_{ui} = \mu + \alpha_u + \beta_i$,
- $n_k(i, u)$ are the k nearest neighbors to i that were rated by user u and
- s_{ij} is the similarity between items i and j, estimated as above.

Negative Similarity Scores?

When using correlation coefficient as the similarity score, negative values have important implications for the weighted calculation.

Looking at the weighted rating formula:

$$\hat{r}_{ui} = b_{ui} + \frac{\sum_{j \in n_k(i,u)} s_{ij}(r_{uj} - b_{uj})}{\sum_{j \in n_k(i,u)} s_{ij}},$$

How Negative Similarity Scores Work

When s_{ij} is **negative** (negative correlation), here's what happens:

1. **Numerator**: The term $s_{ij}(r_{uj} - b_{uj})$ becomes negative when $s_{ij} < 0$. This means:
   - If user u rated item j **above** its bias ($r_{uj} - b_{uj} > 0$), the negative similarity will push the prediction for item i **down**
   - If user u rated item j **below** its bias ($r_{uj} - b_{uj} < 0$), the negative similarity will push the prediction for item i **up**

   This makes intuitive sense: if items i and j are negatively correlated, then liking one means you're likely to dislike the other.

2. **Denominator**: When s_{ij} is negative, it contributes a negative value to the sum in the denominator. This can potentially lead to issues:
   - If all similarities are negative, the denominator becomes negative
   - If there's a mix of positive and negative similarities, they might partially cancel out, potentially making the denominator close to zero or even zero

Common Solutions

To address these issues, practitioners typically use one of these approaches:

1. **Use only positive similarities**: Filter out items with negative correlation (only use k-NN where $s_{ij} > 0$)
2. **Use absolute values in denominator**: $\hat{r}_{ui} = b_{ui} + \frac{\sum_{j \in n_k(i,u)} s_{ij}(r_{uj} - b_{uj})}{\sum_{j \in n_k(i,u)} |s_{ij}|}$
3. **Shift and scale similarities**: Transform correlation values from $[-1, 1]$ to $[0, 1]$ using $s'_{ij} = \frac{s_{ij}+1}{2}$

The first approach (filtering to positive similarities) is most common because negatively correlated items are typically less informative for prediction than positively correlated items in most domains.

Assessing CF

This completes the high level view of CF.

Working with user-user similarities is analogous.

Strengths:

- Essentially no training.
  - The reliance on k-nearest neighbors helps in this respect.
- Easy to update with new users, items, and ratings.
- Explainable:
  - "We recommend *Minority Report* because you liked *Blade Runner* and *Total Recall.*"

Weaknesses:

- Accuracy can be a problem – resulting in poor recommendations
- Scalability can be a problem – compute grows (think k-NN)

Matrix Factorization Approaches

Matrix Factorization

Note that standard CF forces us to consider similarity among items, **or** among users, but does not take into account **both.**

Can we use both kinds of similarity simultaneously?

We can't use both the rows and columns of the ratings matrix R at the same time – the user and item vectors live in different vector spaces.

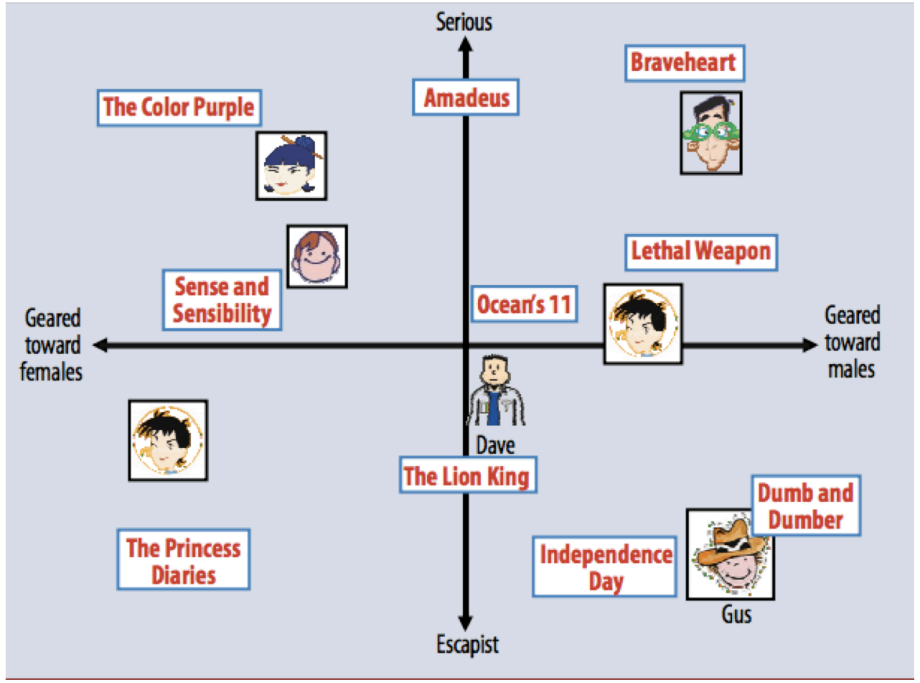


Figure 1

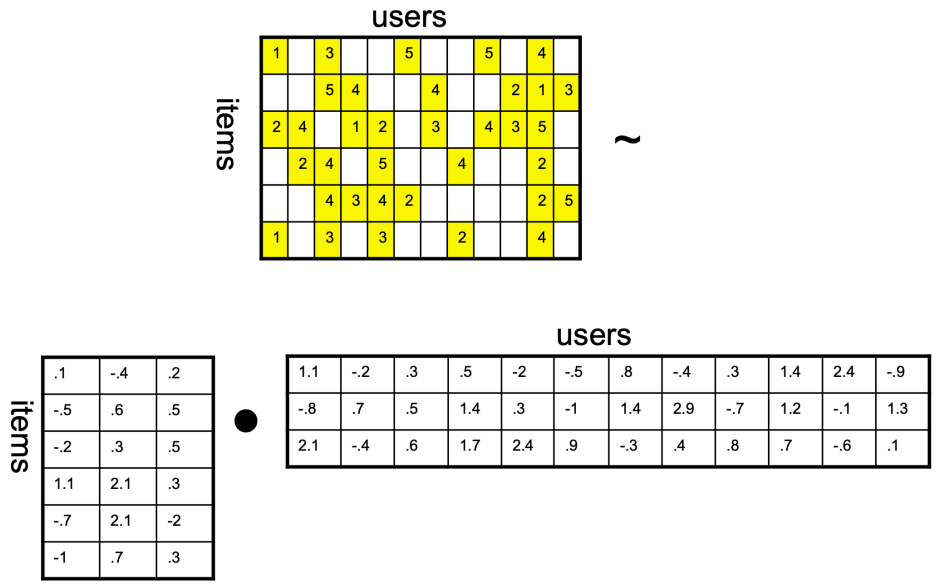
(Koren et al. 2009)

What we could try to do is find a **single** vector space in which we represent **both** users **and** items, along with a similarity function, such that:

- users who have similar item ratings are similar in the vector space
- items who have similar user ratings are similar in the vector space
- when a given user highly rates a given item, that user and item are similar in the vector space.

Now, however, we are working with a matrix which is only **partially observed.** That is, we only know **some** of the entries in the ratings matrix.

Nonetheless, we can imagine a situation like this:

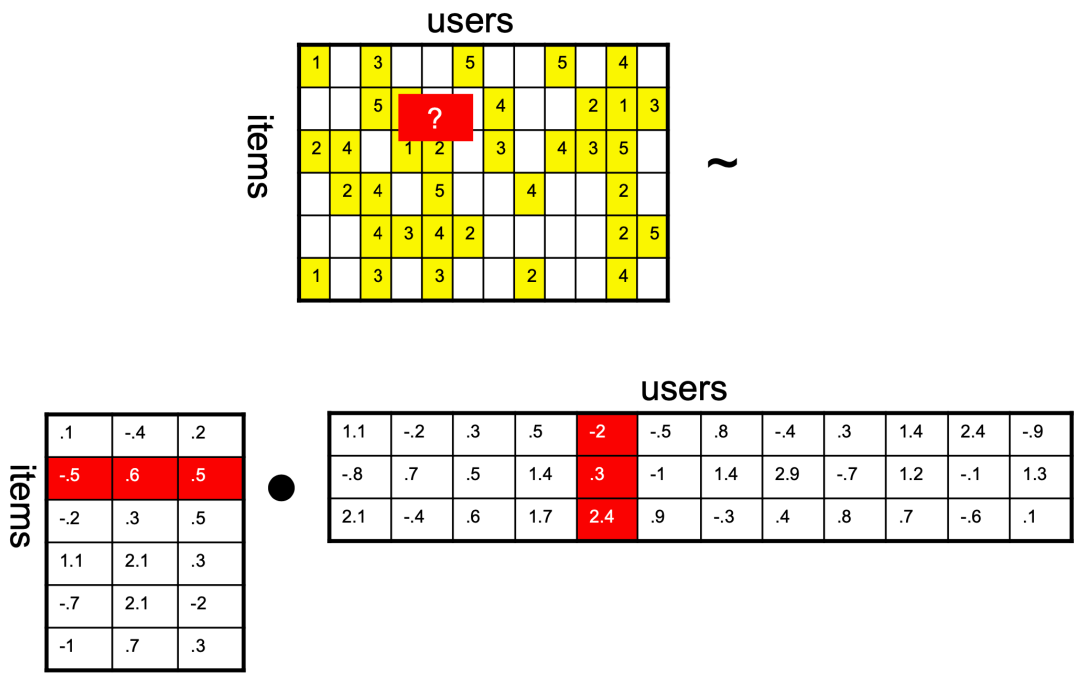


where we decompose the ratings matrix R into two matrices.

We want the product of the two matrices to be as close as possible **to the known values** of the ratings matrix.

What this setup implies is that our similarity function is the **inner product.**

Which means that to predict an unknown rating, we take the **inner product of latent vectors:**

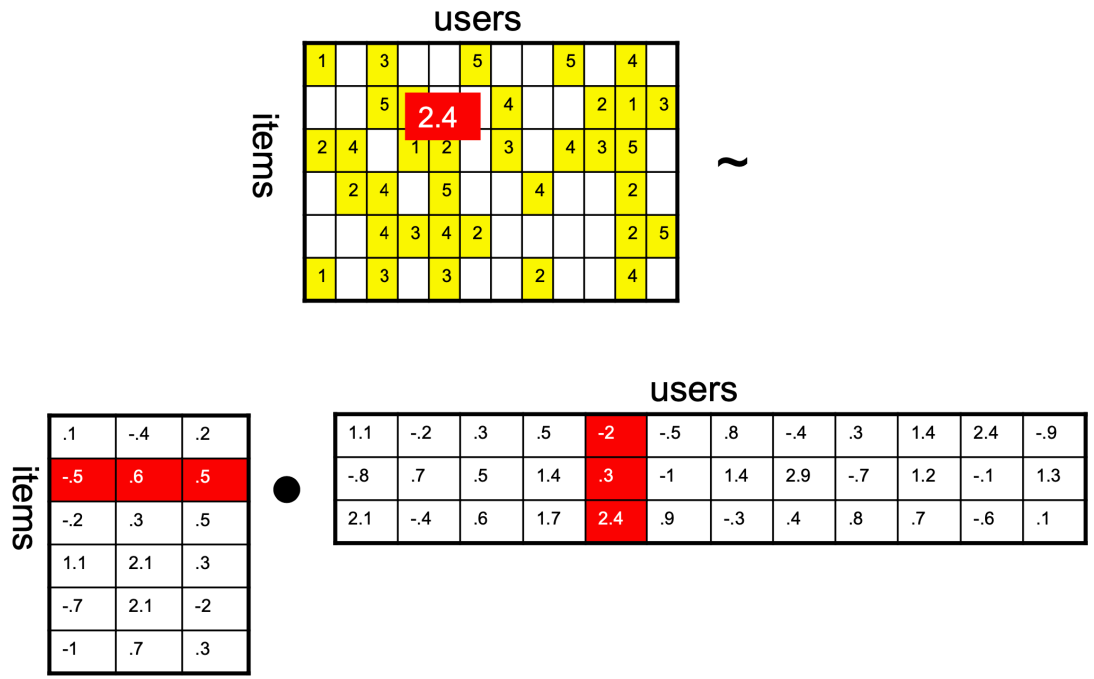


Taking, for example, the 2nd row of “items” and the 5th row of “users”…

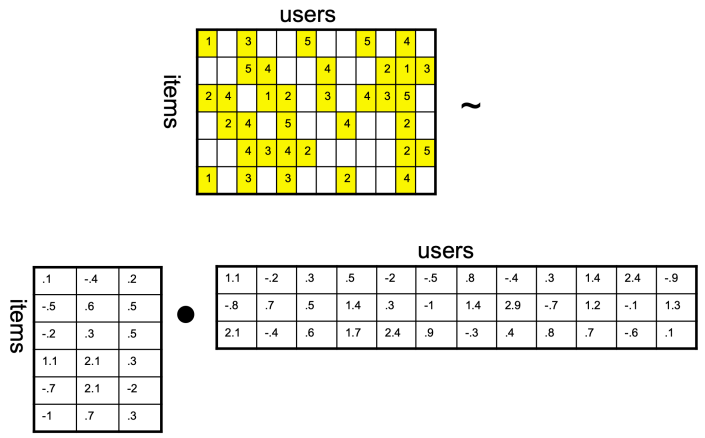
We have

$$(-0.5 \cdot -2) + (0.6 \cdot 0.3) + (0.5 \cdot 2.4) = 2.43,$$

so:



Solving Matrix Factorization



Notice that in this case we've decided that the factorization should be rank 3, i.e., low-rank.

So we want something like an SVD.

(Recall that SVD gives us the most-accurate-possible low-rank factorization of a matrix).

However, we can't use the SVD algorithm directly, because we don't know all the entries in R.

> Indeed, the unseen entries in R are exactly what we want to predict.

Here is what we want to solve:

$$\min_{U,V} \|(R - UV^T)_S\|^2 + \lambda(\|U\|^2 + \|V\|^2)$$

where:

- R is $m \times n$,
- U is the $m \times k$ items matrix,
- V is the $n \times k$ users matrix and
- k is the rank of the factorization and dimensionality of the latent space.

The $(\cdot)_S$ notation means that we are only considering the *subset* of matrix entries that correspond to known reviews (the set S).

Note that as usual, we add ℓ_2 penalization to avoid overfitting (Ridge regression).

$$\min_{U,V} \|(R - UV^T)_S\|^2 + \lambda(\|U\|^2 + \|V\|^2)$$

Once again, this problem is **jointly convex** in that it is convex in each of the variables U and V.

In particular, if we hold either U or V constant, then the result is a simple ridge regression.

$$\min_{U,V} \|(R - UV^T)_S\|^2 + \lambda(\|U\|^2 + \|V\|^2)$$

So one commonly used algorithm for this problem is called **alternating least squares (ALS):**

1. Hold U constant, and solve for V
2. Hold V constant, and solve for U
3. If not converged, go to Step 1.

The only thing we've left out at this point is how to deal with the missing entries of R.

It's not hard, but the details aren't that interesting, so we'll give you code instead!

ALS in Practice

The entire Amazon reviews dataset is too large to work with easily, and it is too sparse.

Hence, we will take the densest rows and columns of the matrix.

▶ Code

Now we create a $\mathrm{UserID} \times \mathrm{ProductID}$ matrix from these reviews.

Missing entries will be filled with NaNs.

▶ Code

```
Review matrix shape: (3677, 7244)
```

And we can look at a small part of the matrix:

▶ Code

ProductId	1417024917	1417030321	1417030976	1417054069
UserId				
A1WLZYEOIL1HLT	NaN	NaN	NaN	NaN
A1WNJVA59HLMO5	NaN	NaN	NaN	NaN
A1WR12AC35R3K6	NaN	NaN	NaN	NaN
A1WSFHRBY2ZD1R	NaN	NaN	NaN	NaN
A1WUMTJOASEL5F	NaN	NaN	NaN	NaN

We'll use code from the Antidote Data Framework to do the matrix factorization and ALS. We have local copies recommender_MF.py, recommender_als.py and recommender_lmafit.py in our repository.

```
# Import local python package MF.py
import recommender_MF as MF

# Instantiate the model
# We are pulling these hyperparameters out of the air -- that's not the right way to do it!
RS = MF.als_MF(rank = 20, lambda_ = 1)
```

```
%time pred, error = RS.fit_model(R)
```

```
CPU times: user 29.8 s, sys: 166 ms, total: 29.9 s
Wall time: 29.7 s
```

```
print(f'RMSE on visible entries (training data): {np.sqrt(error/R.count().sum()):0.3f}')
```

```
RMSE on visible entries (training data): 0.343
```

And we can look at a small part of the predicted ratings matrix and see that it is a dense matrix:

▶ Code

```
Shape of predicted ratings matrix: (3677, 7244)
```

ProductId	1417024917	1417030321	1417030976	1417054069
UserId				
A1WLZYEOIL1HLT	2.828816	4.067808	3.950319	3.949854
A1WNJVA59HLMO5	4.043491	4.930218	4.928916	4.785983
A1WR12AC35R3K6	5.356613	2.955982	4.108460	5.506213
A1WSFHRBY2ZD1R	2.724232	3.618981	3.932077	4.570697
A1WUMTJOASEL5F	4.712673	4.985779	3.714126	4.616174

▶ Code

Just for comparison's sake, let's check the performance of k-NN on this dataset.

Again, this is only on the training data – so overly optimistic for sure.

And note that this is a subset of the full dataset – the subset that is "easiest" to predict due to density.

▶ Code

```
CPU times: user 302 ms, sys: 4.26 ms, total: 307 ms
Wall time: 310 ms
```

▶ Code

```
RMSE on visible entries (test set): 0.494
```

Assessing Matrix Factorization

Matrix Factorization per se is a good idea.
However, many of the improvements we've discussed for CF apply to MF as well.

To illustrate, we'll look at some of the successive improvements used by the team that won the Netflix prize ("BellKor's Pragmatic Chaos").

When the prize was announced, the Netflix supplied solution achieved an RMSE of 0.951.

By the end of the competition (about 3 years), the winning team's solution achieved RMSE of 0.856.

Let's restate our MF objective in a way that will make things clearer:

$$\min_{U,V} \sum_{(u,i)\in S} (r_{ui} - u_u^T v_i)^2 + \lambda(\|U\|^2 + \|V\|^2)$$

where we have written out the vector ℓ_2 norm as the summation.

1. Adding Biases

If we add biases:

$$\min_{U,V} \sum_{(u,i)\in S} (r_{ui} - (\mu + \alpha_u + \beta_i + u_u^T v_i)^2 + \lambda(\|U\|^2 + \|V\|^2 + \|\alpha\|^2 + \|\beta\|^2)$$

we see improvements in accuracy:

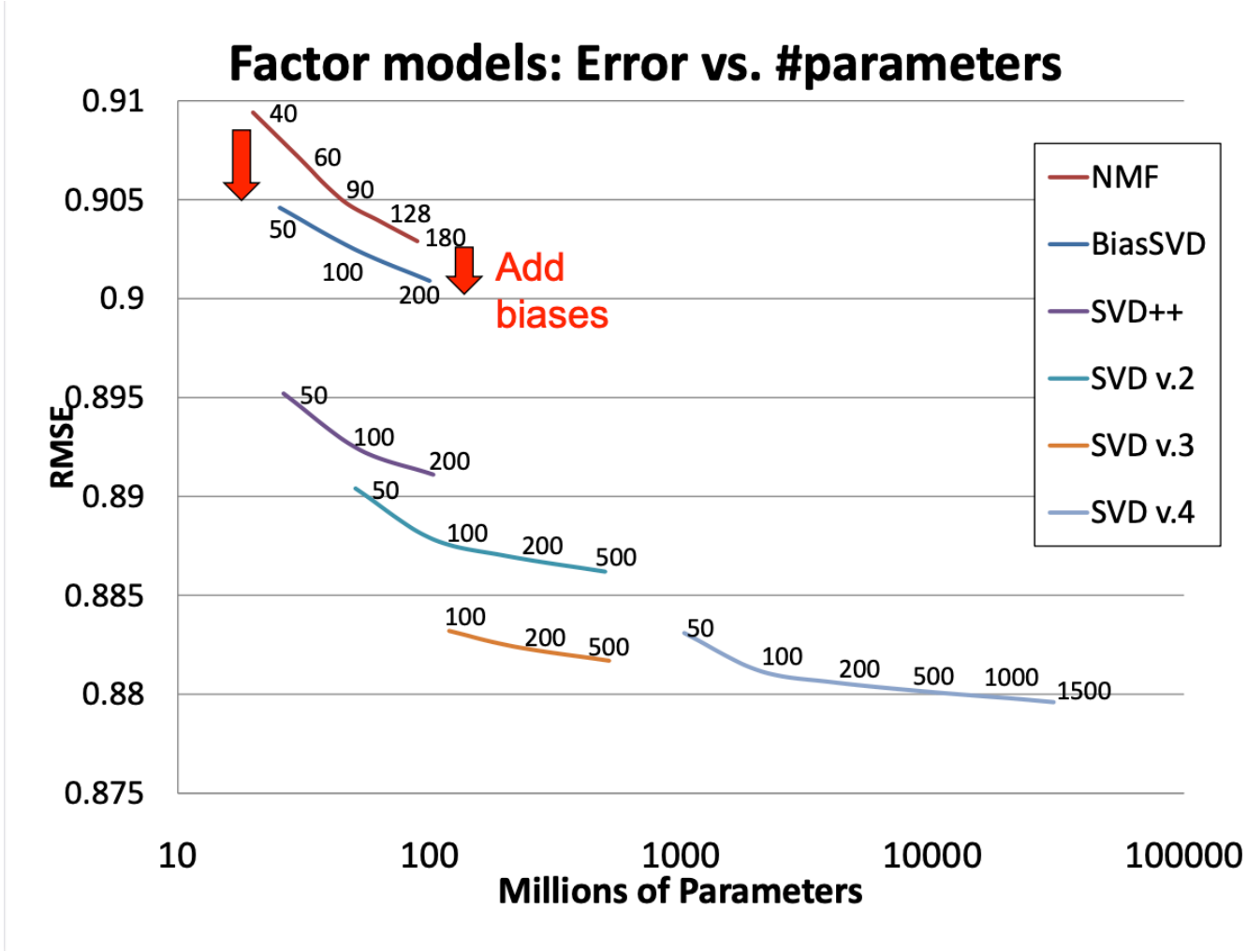
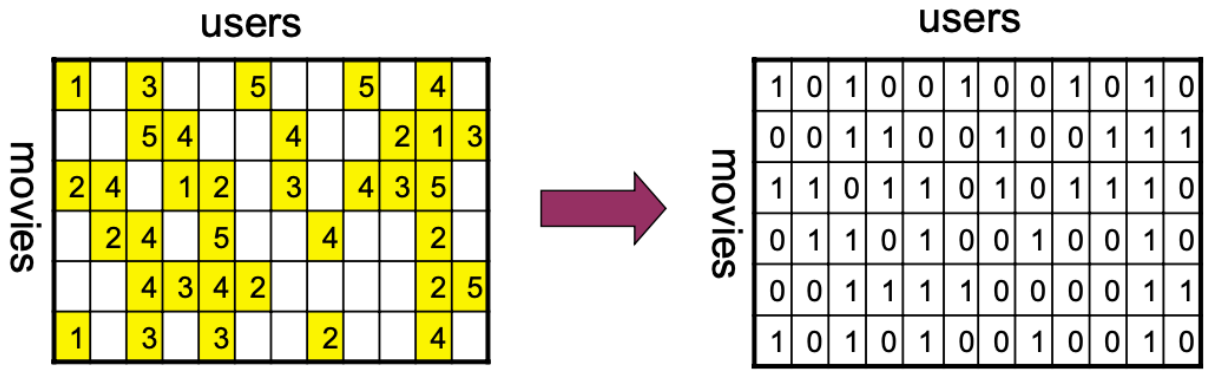


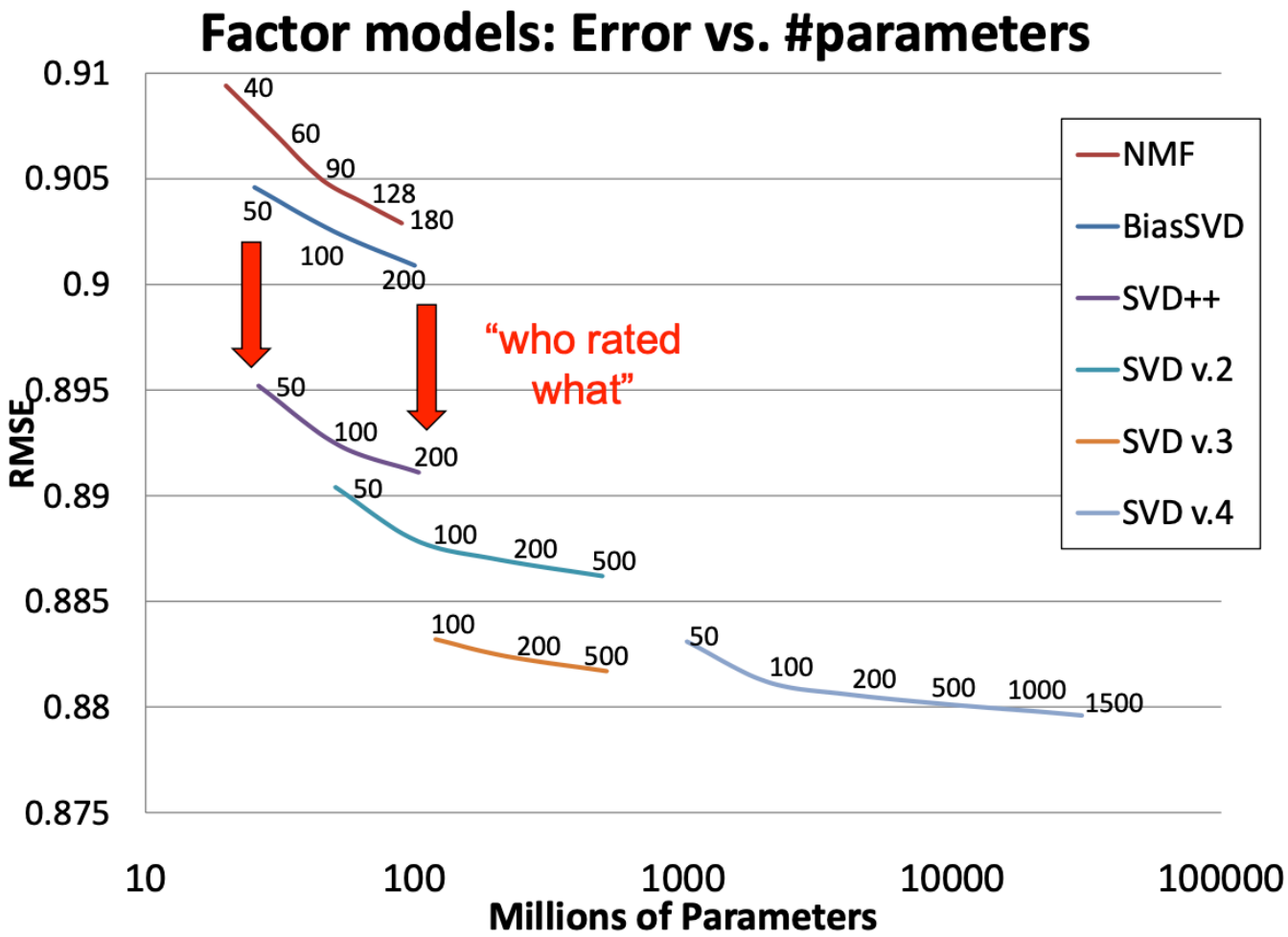
Figure 2: Matrix factorization models' accuracy. The plots show the root-mean-square error of each of four individual factor models (lower is better). Accuracy improves when the factor model's dimensionality (denoted by numbers on the charts)

2. Who Rated What?

In reality, ratings are not provided **at random.**

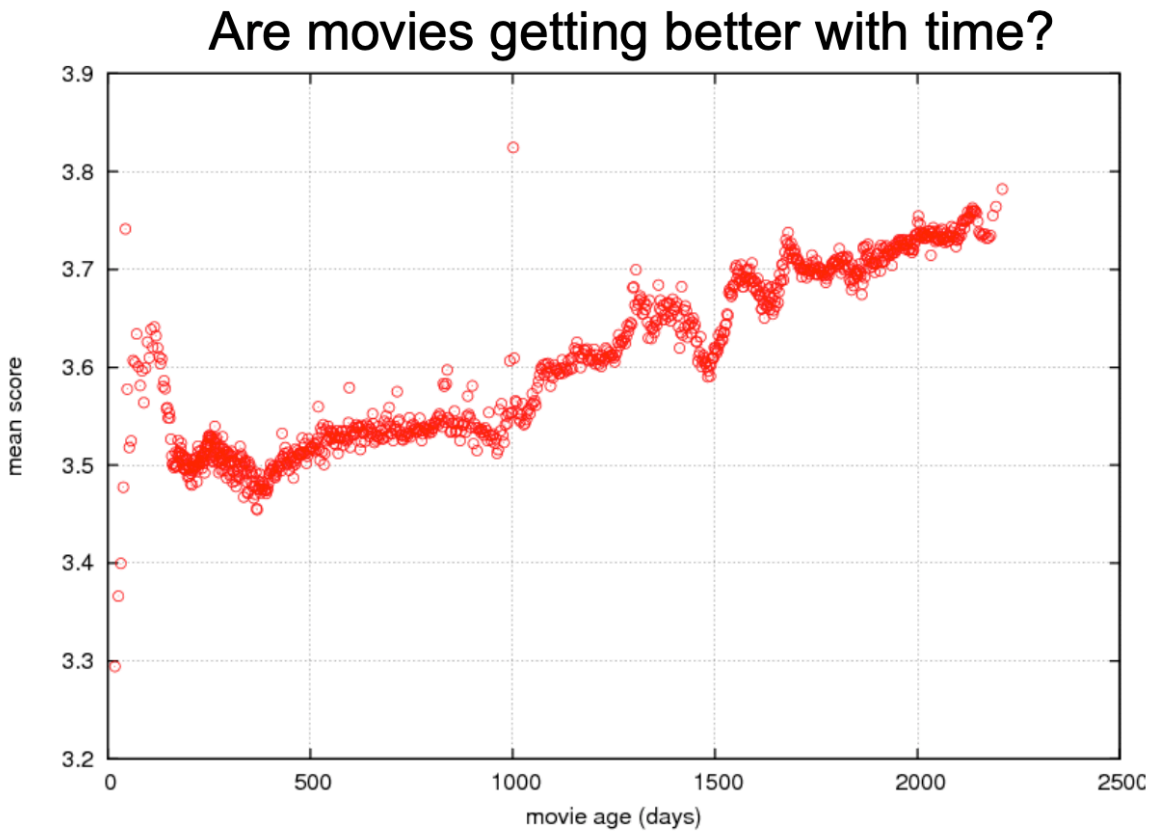
Take note of which users rated the same movies (ala CF) and use this information.




Factor models: Error vs. #parameters
RMSE
0.91
0.905
0.9
0.895
0.89
0.885
0.88
0.875
10
100
1000
10000
100000
Millions of Parameters
NMF
BiasSVD
SVD++
SVD v.2
SVD v.3
SVD v.4
40
60
90
128
180
50
100
200
50
100
200
"who rated what"
50
100
200
500
100
200
500
50
100
200
500
1000
1500

3. Ratings Change Over Time

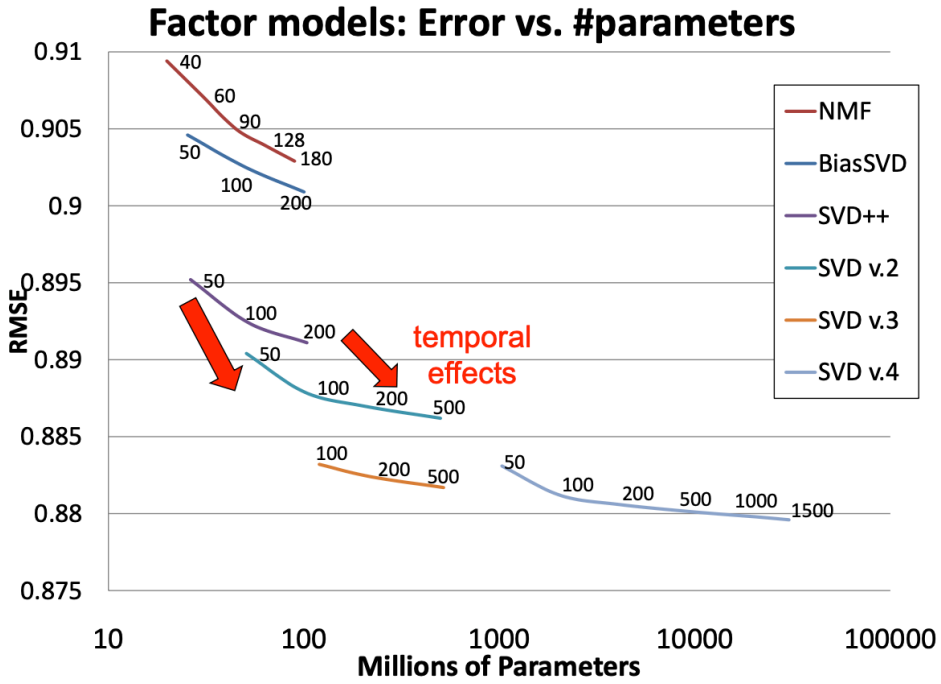
Older movies tend to get higher ratings!



If we add time-varying biases:

$$\min_{U,V} \sum_{(u,i)\in S} (r_{ui} - (\mu + \alpha_u(t) + \beta_i(t) + u_u^T v_i(t))^2 + \lambda(\|U\|^2 + \|V\|^2 + \|\alpha\|^2 + \|\beta\|^2$$

we see further improvements in accuracy:



To estimate these billions of parameters, we cannot use alternating least squares or any linear algebraic method.

We need to use gradient descent (which we covered previously).

Recap

- Introduction to recommender systems and their importance in modern society.
- Explanation of collaborative filtering (CF) and its two main approaches: user-user similarity and item-item similarity.
- Discussion on the challenges of recommender systems, including scalability and data sparsity.
- Introduction to matrix factorization (MF) as an improvement over CF, using latent vectors and alternating least squares (ALS) for optimization.
- Practical implementation of ALS for matrix factorization on a subset of Amazon movie reviews.

References

Koren, Yehuda. 2009. “Collaborative Filtering with Temporal Dynamics.” *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 42: 447–56. https://dl.acm.org/doi/10.1145/1557019.1557072.

Koren, Yehuda, Robert Bell, and Chris Volinsky. 2009. “Matrix Factorization Techniques for Recommender Systems.” *Computer* 42 (8): 30–37. https://ieeexplore.ieee.org/document/5197422.

Naumov, Maxim et al. 2019. “Deep Learning Recommendation Model for Personalization and Recommendation Systems.” *arXiv Preprint arXiv:1906.00091*, May. http://arxiv.org/abs/1906.00091.