Recommenders

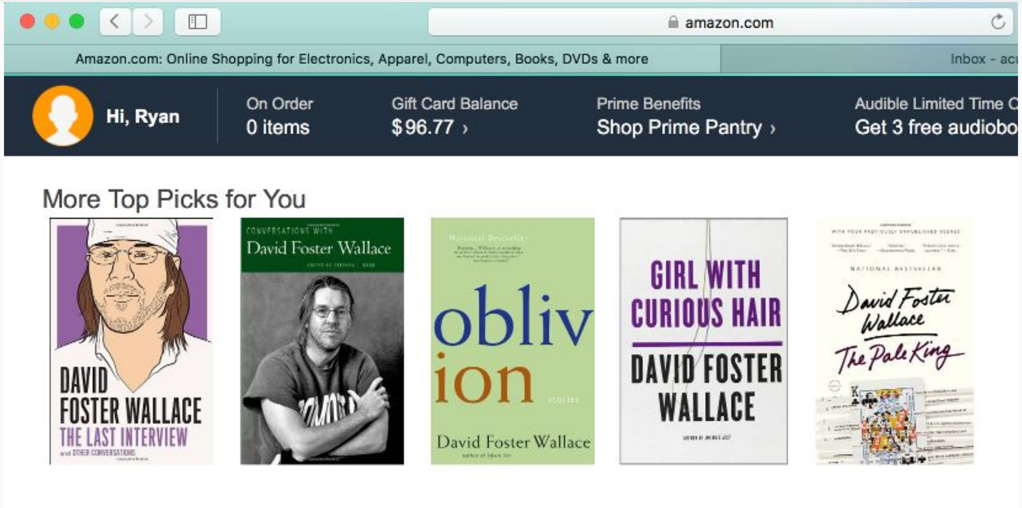
Warning: this will be a very interactive lecture!

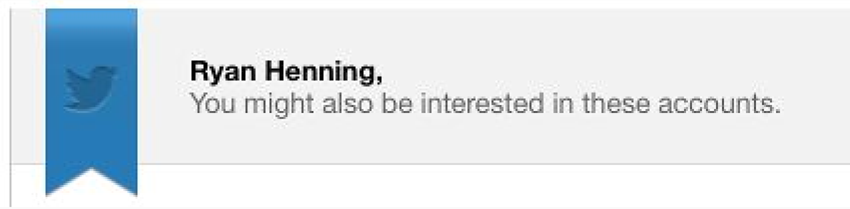
Ryan Henning

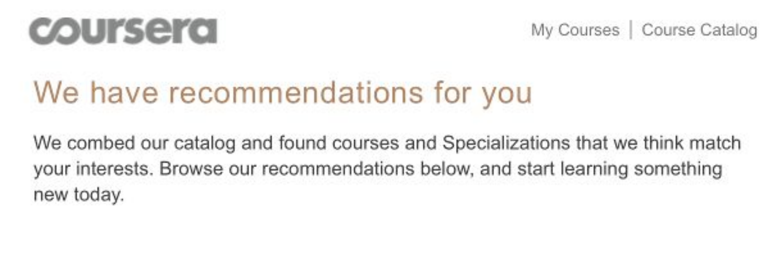
(heavily based on Dan Becker’s slides and on Giovanna's notes)

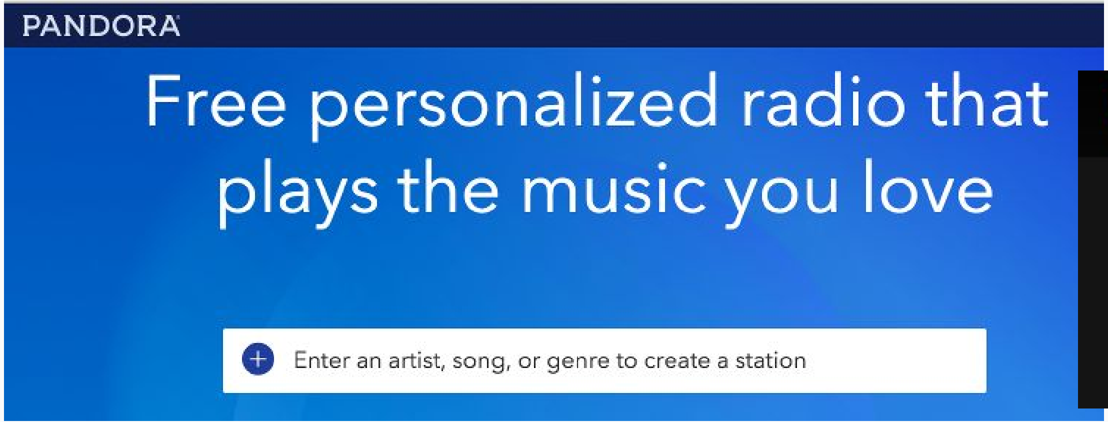
- Where are recommenders used? (Hint: everywhere!)
- What does our dataset look like?
- What are the high-level approaches to building a recommender?
  - Content-based
  - Collaborative filtering
  - Matrix factorization
- How do we evaluate our recommender system?
- How to deal with “cold start”?
- What are the computational performance concerns?

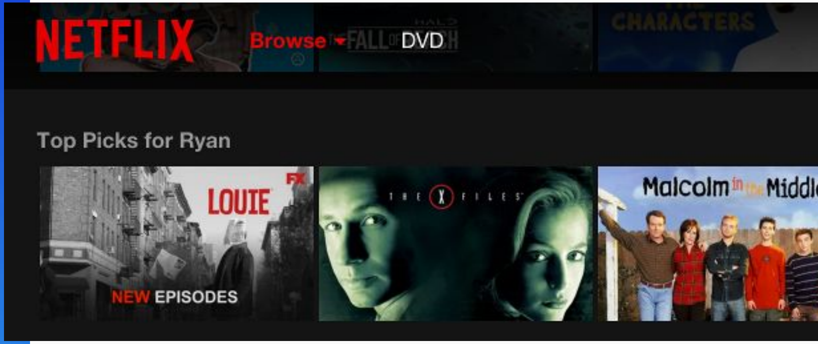
Where are recommenders used?











Business Goals:

What will the user **like**?

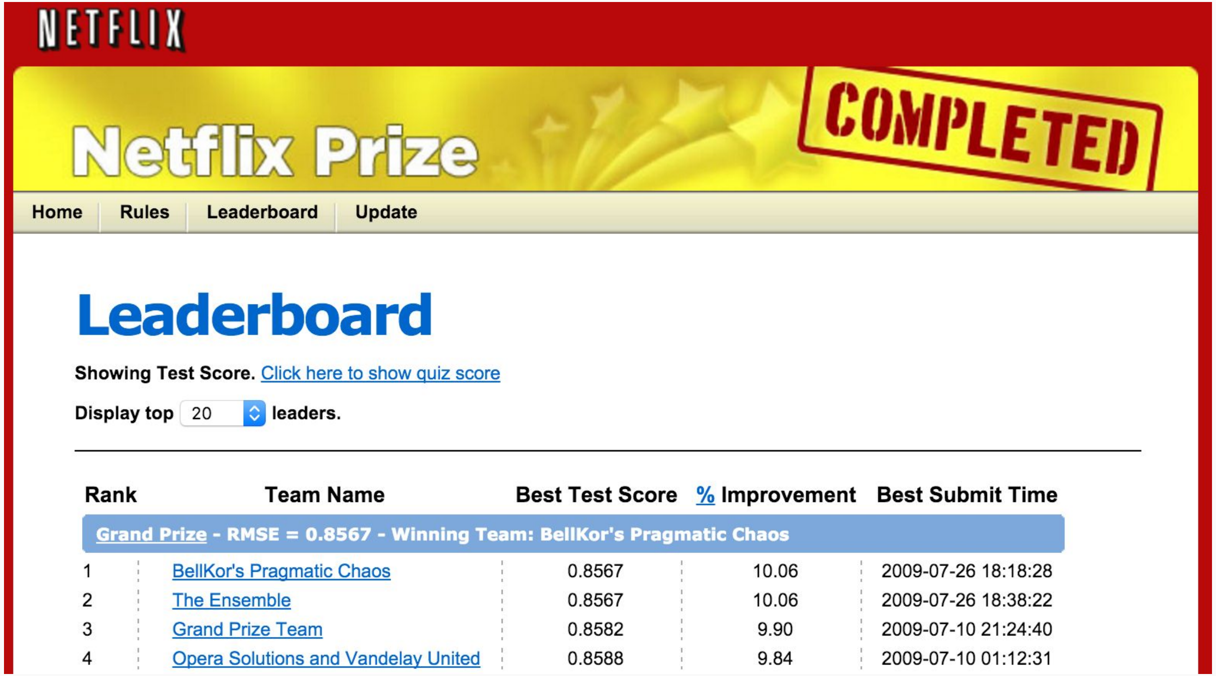
What will the user **buy**?

What will the user **click**?

Name a business that cares are each of these, and tell us why they care.

Data Science Canon:

Netflix’s $1,000,000 Prize (Oct. 2006 - July 2009)


NETFLIX

Netflix Prize

COMPLETED

Home | Rules | Leaderboard | Update

Leaderboard

Showing Test Score. Click here to show quiz score

Display top 20 leaders.

Rank	Team Name	Best Test Score	% Improvement	Best Submit Time
Grand Prize - RMSE = 0.8567 - Winning Team: BellKor's Pragmatic Chaos				
1	BellKor's Pragmatic Chaos	0.8567	10.06	2009-07-26 18:18:28
2	The Ensemble	0.8567	10.06	2009-07-26 18:38:22
3	Grand Prize Team	0.8582	9.90	2009-07-10 21:24:40
4	Opera Solutions and Vandelay United	0.8588	9.84	2009-07-10 01:12:31

Goal: Beat Netflix’s own recommender by 10%.

Took almost 3 years.

The winning team used gradient boosted decision trees over the predictions of **500** other models.

Netflix never deployed the winning algorithm.

Let’s learn recommenders.

Today we’ll learn:

1. How to **build** a recommender,
2. How to **evaluate** your recommender, and
3. How to **deploy** your recommender.

… and tomorrow we’ll do a case study with recommender systems.

What are the high-level approaches to building a recommender?

Popularity:

- Make the **same** recommendation to **every** user, based only on the popularity of an item.
- E.g. Twitter "Moments"

Content-based (aka, Content filtering):

- Predictions are made based on the properties/characteristics of an item.
- User behavior is **not** considered.
- E.g. Pandora Radio

Collaborative filtering:

- Only consider past user behavior. (**not** content properties...)
- User-User similarity: ...
- Item-Item similarity: ...
- E.g.
    - Netflix & Amazon Recommendations,
    - Google Ads,
    - Facebook Ads, Search, Friends Rec.,

      News feed, Trending news, Rank Notifications, Rank Comments

Matrix Factorization Methods:

- Find latent features (aka, factors)

What does our dataset look like? (these looks like explicit ratings to me...)

Item

User

	A	B	C	D	...
Al	1	?	2	?	
Bob	?	2	3	4	
Cat	3	?	1	5	
Dan	?	2	?	?	
Ed	2	?	?	1	
...					

What company might have data like this?

Btw, we call this the *utility matrix*.

What does our dataset look like? (these looks like implicit boolean rating to me...)

User	Item				
	A	B	C	D	...
Al	0	1	0	1	
Bob	0	0	1	0	
Cat	0	1	1	1	
Dan	1	0	0	1	
Ed	0	1	0	0	
...					

What company might have data like this?

Btw, we call this the *utility matrix*.

User-User similarities

Item

User	A	B	C	D	...
Al	1	?	2	?	
Bob	?	2	3	4	
Cat	3	?	1	5	

We look at all pairs of users and calculate their similarity.

How can we calculate the similarity of these row vectors? (We'll get there.)

Item-Item similarities

Item

User

	A	B	C	D
Al	1	?	2	?
Bob	?	2	3	4
Cat	3	?	1	5
Dan	?	2	?	?
Ed	2	?	?	1
…				

We look at all pairs of items and calculate their similarity.

How can we calculate the similarity of these column vectors? (We'll get there.)

User-User or Item-Item?

User-User:

	Item				
User	A	B	C	D	...
Al	1	?	2	?	
Bob	?	2	3	4	
Cat	3	?	1	5	

Item-Item:

	Item			
User	A	B	C	D
Al	1	?	2	?
Bob	?	2	3	4
Cat	3	?	1	5
Dan	?	2	?	?
Ed	2	?	?	1
...				

Let:
m = #users,
n = #items

We want to compute the similarity of all pairs.

What is the algorithmic efficiency of each approach?

User-User: $O(m^2n)$
Item-Item: $O(mn^2)$

Which one is better?

Similarity Metric using Euclidean Distance

$$\mathrm{dist}(a, b) = ||a - b|| = \sqrt{\sum_i (a_i - b_i)^2}$$

What's the range?

But we're interested in a **similarity**, so let's do this instead:

When use this?

$$\mathrm{similarity}(a, b) = \frac{1}{1 + \mathrm{dist}(a, b)}$$

What's the range?

Similarity Metric using Pearson Correlation

$$\text{pearson}(a,b) = \frac{\text{cov}(a,b)}{\text{std}(a) * \text{std}(b)} = \frac{\sum_i (a_i - \bar{a})(b_i - \bar{b})}{\sqrt{\sum_i (a_i - \bar{a})^2}\sqrt{\sum_i (b_i - \bar{b})^2}}$$

What's the range?

When use this?

But we're interested in a **similarity**, so let's do this instead:

$$\text{similarity}(a,b) = 0.5 + 0.5 * \text{pearson}(a,b)$$

What's the range?

Similarity Metric using Cosine Similarity

$$\cos(\theta_{a,b}) = \frac{a \cdot b}{||a||\,||b||} = \frac{\sum_i a_i b_i}{\sqrt{\sum_i a_i^2}\sqrt{\sum_i b_i^2}}$$

What's the range?

When use this?

But we're interested in a **standardized similarity**, so let's do this instead:

$$\text{similarity}(a, b) = 0.5 + 0.5 * \cos(\theta_{a,b})$$

What's the range?

Similarity Metric using Jaccard Index

What's the range?

$$\text{similarity}(a, b) = \frac{|U_a \cap U_b|}{|U_a \cup U_b|}$$

$$U_k \text{ denotes the set of users who rated item } k$$

When use this?

The Similarity Matrix

Pick a similarity metric, create the similarity matrix:

	item 1	item 2	item 3	...
item 1	1	0.3	0.2	...
item 2	0.3	1	0.7	...
item 3	0.2	0.7	1	...
...	...	...	...	...

How to make predictions

Say user *u* hasn't rated item *i*. We want to predict the rating that this user *would* give this item.

$$\text{rating}(u, i) = \frac{\sum_{j \in I_u} \text{similarity}(i, j) * r_{u,j}}{\sum_{j \in I_u} \text{similarity}(i, j)}$$

$$\begin{aligned} I_u &= \text{ set of items rated by user } u \\ r_{u,j} &= \text{ user } u\text{'s rating of item } j \end{aligned}$$

We order by descending predicted rating for a single user, and recommend the top *k* items to the user.

How to make predictions (using neighborhoods)

This calculation of predicted ratings can be very costly. To mitigate this issue, we will only consider the *n* most similar items to an item when calculating the prediction.

$$\text{rating}(u, i) = \frac{\sum_{j \in I_u \cap N_i} \text{similarity}(i, j) * r_{u,j}}{\sum_{j \in I_u \cap N_i} \text{similarity}(i, j)}$$

$$I_u = \text{ set of items rated by user } u$$

$$r_{u,j} = \text{ user } u\text{'s rating of item } j$$

$$N_i \text{ is the } n \text{ items which are most similar to item } i$$

Deploying the recommender

In the middle of the night:

- Compute similarities between all pairs of items.
- Compute the neighborhood of each item.

At request time:

- Predict scores for candidate items, and make a recommendation.

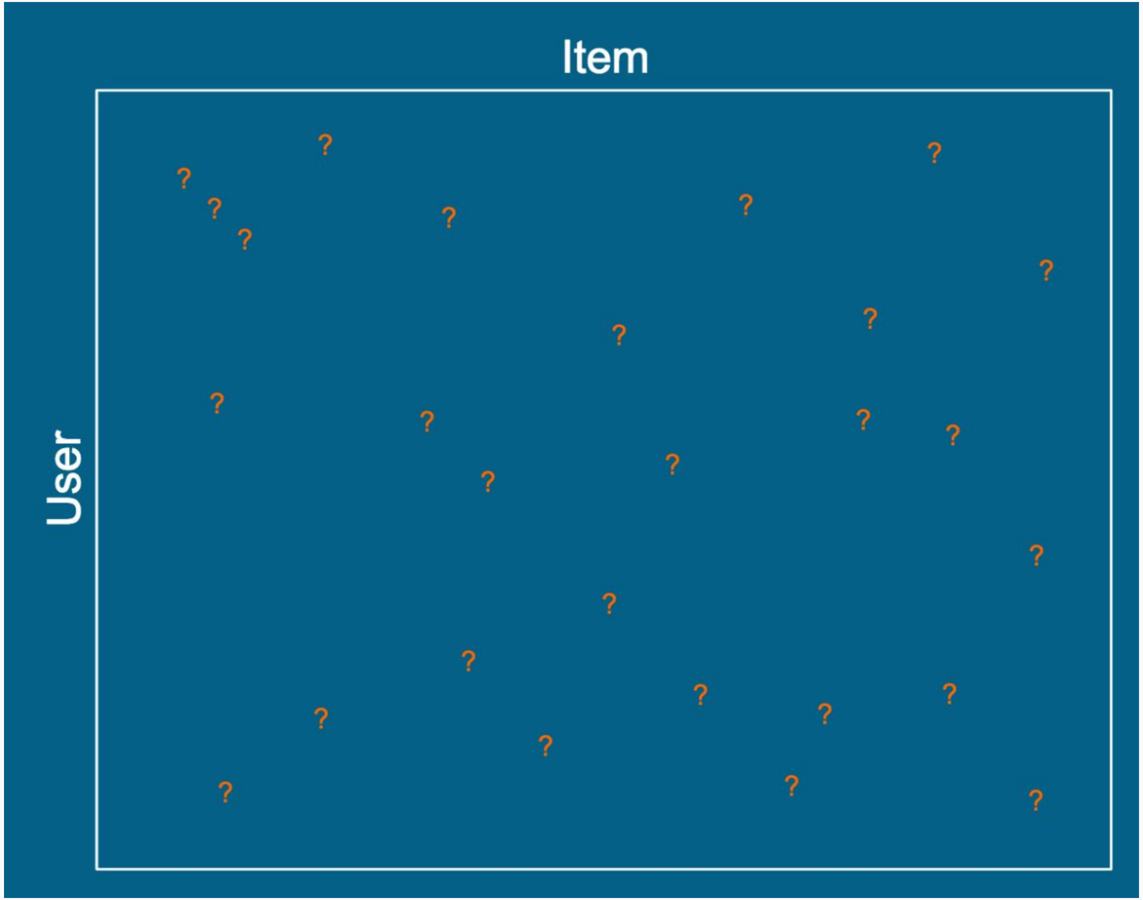
How do we evaluate our recommender system?

Is it possible to do cross-validation like normal?

Before we continue, let's review: Why do we perform cross-validation?

Quick warning: Recommenders are inherently hard to validate. There is a lot of discussion in academia (research papers) and industry (here, Kaggle, Netflix, etc) about this. There is no ONE answer for all dataset.

The closest thing to cross-validation



For this slide, the question marks denote the holdout set (**not** missing values).

We can calculate MSE between the targets and our predictions over the holdout set.

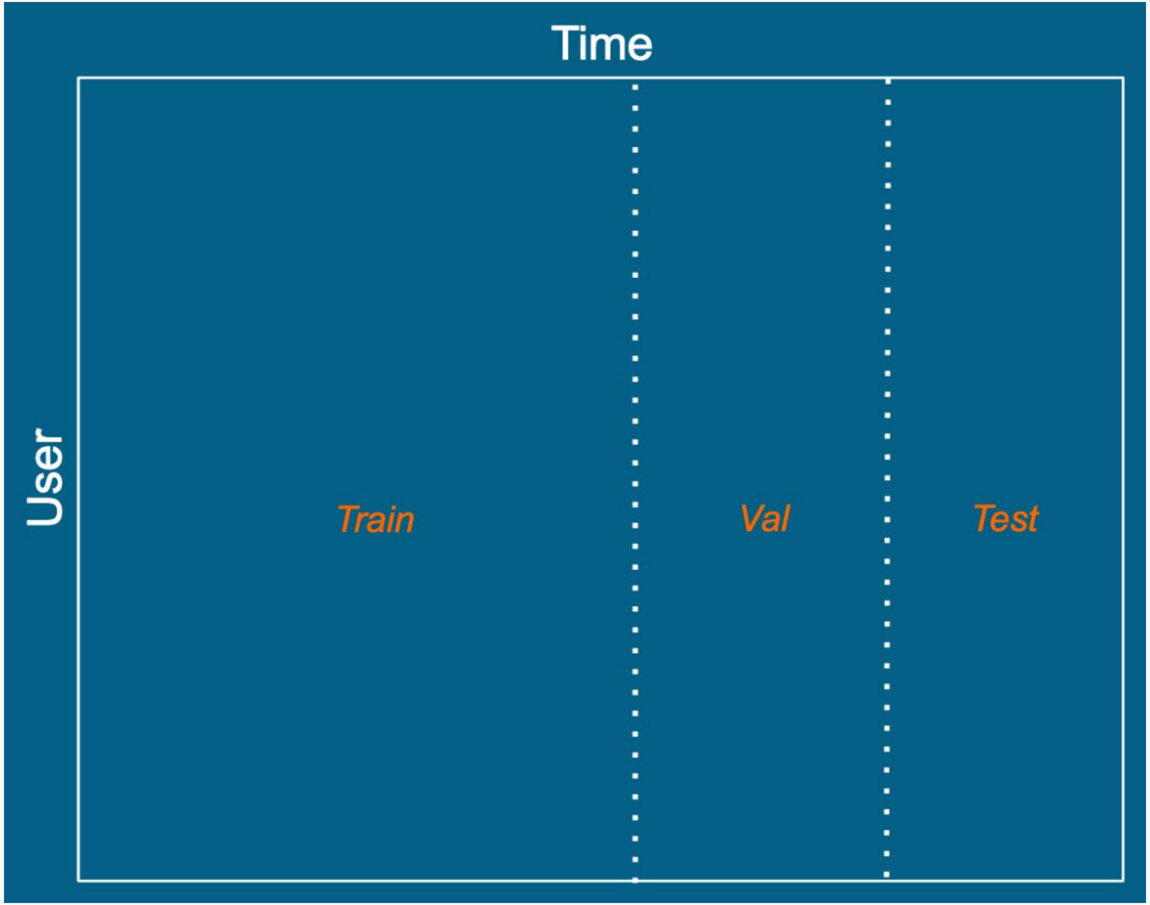
(K-fold cross-validation is optional.)

Recall: Why do we perform cross-validation?

Why isn't the method above a true estimate of a recommender's performance in the field?

Why would A/B testing be better?

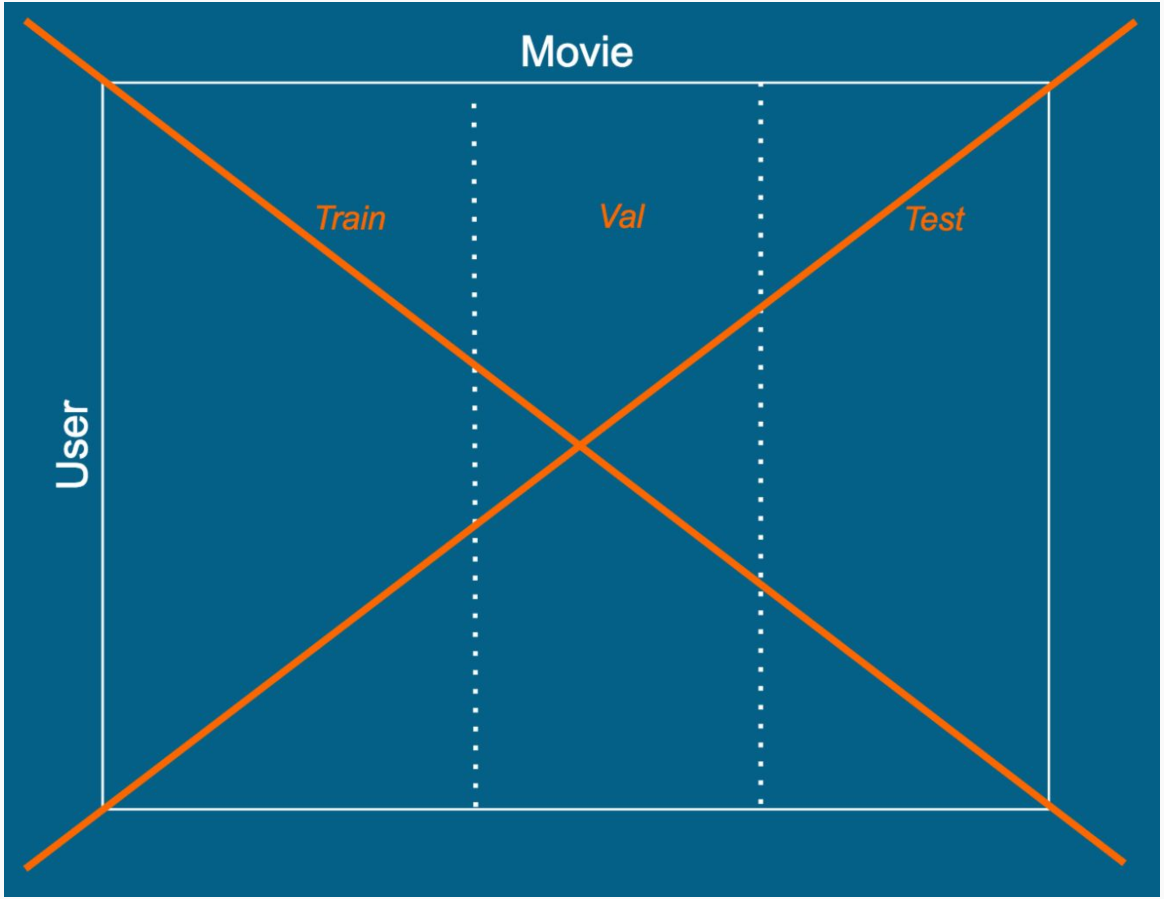
Alternate way to validate:



What's the deal with this?

I.e. Why might we prefer doing this instead of the more "normal" cross-validation from the previous slide?

DON'T DO THIS! Why?



How to deal with “cold start”?

Scenario: A new user signs up.
What will our recommender do (assume we’re using item-item similarities)?

One strategy: Force users to rate 5 items as part of the signup process. AND/OR Recommend popular items at first.

Scenario: A new item is introduced.
What will our recommender do (assume we’re using item-item similarities)?

One strategy: Put it in the “new releases” section until enough users rate it AND/OR use item metatdata if any exists.

How to deal with “cold start”?

Scenario: A new user signs up.
What will our recommender do (assume we’re Youtube and we’re using item popularity to make recommendations)?

This really isn’t a problem...

Scenario: A new item is introduced.
What will our recommender do (assume we’re Youtube and we’re using item popularity to make recommendations)?

One strategy: Don’t use total number of views as the popularity metric (we’d have a *rich-get-richer* situation). Use something else...

Matrix Factorization for Recommendation

Warning: There are a lot of acronyms in this lecture!

Ryan Henning

- UV Decomposition (UVD)
- SVD vs UVD
- UVD vs NMF
- UVD via Stochastic Gradient Descent (SGD)
- Matrix Factorization for Recommendation:
  - Basic system:
    - UVD + SGD... FTW
  - Intermediate topics:
    - regularization
    - accounting for biases

UV Decomposition (UVD)

$$R_{m \times n} \approx U_{m \times k} V_{k \times n}$$

$$r_{ij} \approx u_{i:} \cdot v_{:j}$$

- You choose *k*.
- *UV* approximates *R* by necessity if *k* is less than the rank of *R*.
- Usually choose: *k << min(n,m)*
- Compute *U* and *V* such that:

Least Squares!

$$\arg\min_{U,V} \sum_{i,j} (r_{ij} - u_{i:} \cdot v_{:j})^2$$

SVD vs UVD

SVD:

- $R = USV^T$
- U is an orthogonal matrix
- S is a diagonal matrix of decreasing positive “singular” values
- V is an orthogonal matrix
- Has a unique, exact solution

UVD:

- $R \sim= UV$
- U and V will not (likely) be orthogonal
- Has many approximate, non-unique solutions:
  - non-convex optimization; has many local minima
- Has a tunable parameter k

UVD vs NMF

UVD:

- By convention: $R \sim= UV$
- … (see previous slides)

NMF is a specialization of **UVD**!

Both are approximate factorizations, and both optimize to reduce the RSS.

NMF:

- By convention: $V \sim= WH$
- Same as UVD, but with one extra constraint:
  all values of *V*, *W*, and *H* must be non-negative!

UVD vs NMF *(continued)*

UVD and NMF are both solved using either:

- Alternating Least Squares (ALS)
- Stochastic Gradient Descent (SGD)

You did **ALS** yesterday,
so let's do **SGD** today!

(and we'll see why SGD has some
advantages for recommender systems)

UVD via Stochastic Gradient Descent (SGD)

Boardwork… (take notes!)

ALS vs SGD

ALS:

- Parallelizes very well
- No learning rate required
- Available in Spark/MLlib
- Only appropriate for matrices that don't have missing values
  (we'll call this a **dense** matrix in this lecture)

SGD:

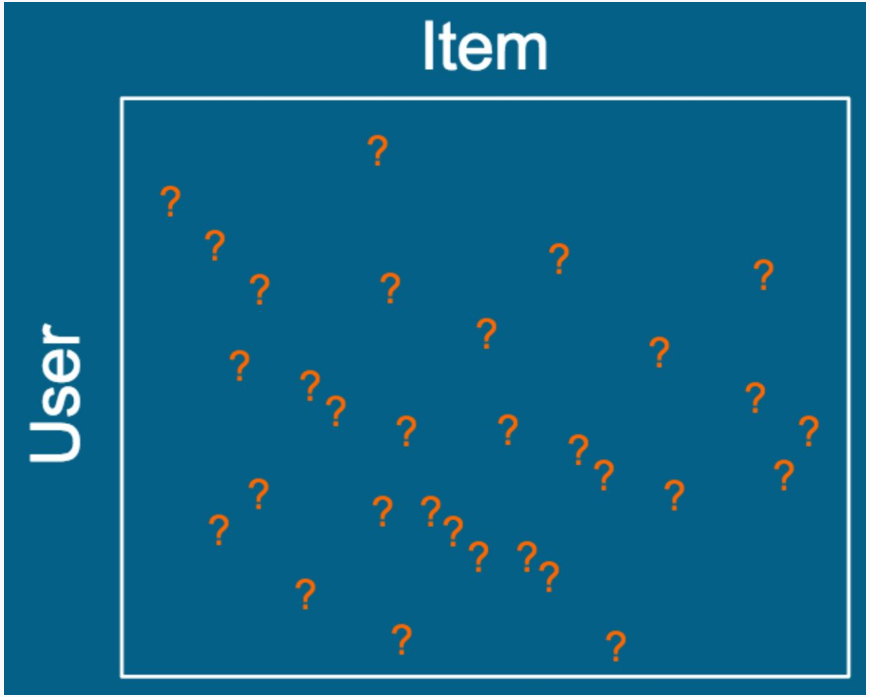
- Faster (if on single machine)
- Requires tuning learning rate
- Anecdotal evidence of better results...
- Works with missing values
  (we'll call this a **sparse** matrix in this lecture)
  (we'll see how missing values are handled soon!)

Matrix Factorization for Recommendation

Recall: An explicit-rating utility matrix is usually VERY sparse…

We've previously used SVD to find latent features (aka, factors)... Would SVD be good for this sparse utility matrix?
(Hint: No!)

What's the problem with using SVD on this sparse utility matrix?



Would UVD (or NMF) work better than SVD to find latent factors when the utility matrix is sparse?

(Hint: Consider ways to change the SGD algorithm to handle missing values in the sparse utility matrix.)

SVD vs UVD *(revisited)*

SVD:

- *$R = USV^T$*
- ...
- **Bad if *R* has missing values!**
  - You are forced to fill in missing values.
  - Solution fits these fill-values (which is silly).
  - Makes for a much larger memory footprint.
  - Slow to compute for large matrices.

UVD:

- *R ~= UV*
- ...
- **Handles missing values when computed via SGD.**

$$\arg\min_{U,V} \sum_{i,j \in \mathcal{K}} (r_{ij} - u_{i:} \cdot v_{:j})^2$$

Set of indices of known rating

UVD (or NMF) + SGD... FTW!

UVD + SGD makes a lot of sense for recommender systems.

In fact, **NMF + SGD** is '**best in class**' option for *many* recommender domains:

- No need to impute missing values.
- Use regularization to avoid overfitting.
- Optionally include biases terms to communicate prior knowledge.
- Can handle time-dynamics (e.g. change in user preference over time).
- Used by the winning entry in the Netflix challenge.

Warning: Don’t forget to regularize!

Since now we’re fitting a large parameter set to sparse data, you’ll most certainly need to regularize!

$$\arg\min_{U,V} \sum_{i,j \in \mathcal{K}} (r_{ij} - u_{i:} \cdot v_{:j})^2 + \lambda(||u_{i:}||^2 + ||v_{:j}||^2)$$

Tune lambda: the amount of regularization

Accounting for Biases (let's capture our domain knowledge!)

In practice, much of the observed variation in rating values is due to item bias and user bias:

- Some items (e.g. movies) have a tendency to be rated high, some low.
- Some users have a tendency to rate high, some low.

We can capture this prior domain knowledge using a few bias terms:

$$b_{ij} = \mu + b_i^* + b_j'$$

The overall bias of the rating by user *i* for item *j*

The overall average rating
(i.e. the overall bias)

User i's average deviation from the overall average

Item j's average deviation from the overall average

We added bais terms... now:

The 4 parts of a prediction

$$r_{ij} \approx \mu + b^*_i + b'_j + u_{i:} \cdot v_{:j}$$

The prediction of user i rating item j

The average rating

User i's tendency to deviate from the average

Item j's tendency to deviate from the average

The prediction of how user i will interact with item j

Accounting for Biases (the new cost function)

Ratings are now estimated as:

$$r_{ij} \approx \mu + b_i^* + b_j' + u_{i:} \cdot v_{:j}$$

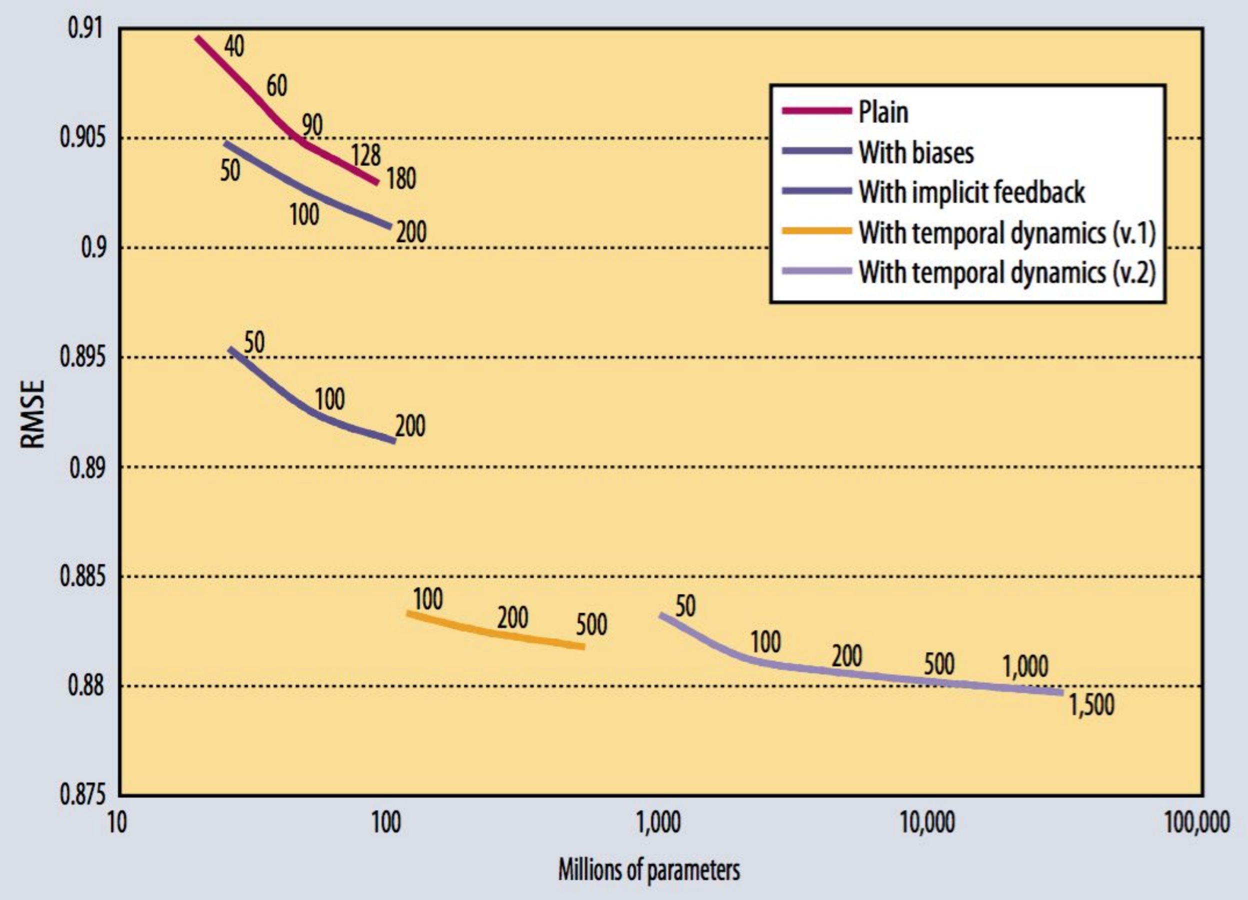
The new cost function, with the biases included:

$$\arg\min_{U,V,b^*,b'} \sum_{i,j \in \mathcal{K}} (r_{ij} - \mu - b_i^* - b_j' - u_{i:} \cdot v_{:j})^2 + \lambda_1(||u_{i:}||^2 + ||v_{:j}||^2) + \lambda_2((b_i^*)^2 + (b_j')^2)$$

New part!

New part!

From the paper: “Matrix Factorization Techniques for Recommender Systems”



Root mean square error over the Netflix dataset using various matrix factorization models.

Numbers on the chart denote each model’s dimensionality (k).

The more refined models perform better (have lower error).

Netflix’s inhouse model performs at RMSE=0.9514 on this dataset, so even the simple matrix factorization models are beating it!

Read the paper for details; it’s a good read!