
Collaborative Filtering



Agenda

- **Collaborative Filtering (CF)**
 - Pure CF approaches
 - User-based nearest-neighbor
 - The Pearson Correlation similarity measure
 - Memory-based and model-based approaches
 - Item-based nearest-neighbor
 - The cosine similarity measure
 - Data sparsity problems
 - Discussion and summary
 - Code reading

Collaborative Filtering (CF)

- **The most prominent approach to generate recommendations**
 - Used by large, commercial e-commerce sites
 - Well-understood, various algorithms and variations exist
 - Applicable in many domains (book, movies, DVDs, ..)
- **Approach**
 - Use the "wisdom of the crowd" to recommend items
- **Basic assumption and idea**
 - Users give ratings to catalog items (implicitly or explicitly)
 - Customers who had similar tastes in the past, will have similar tastes in the future



Pure CF Approaches

- **Input**
 - Only a matrix of given user–item ratings
- **Output types**
 - A (numerical) prediction indicating to what degree the current user will like or dislike a certain item
 - A top-N list of recommended items

User-based nearest-neighbor collaborative filtering

- **The basic technique**

- Given an "active user" (Alice) and an item i not yet seen by Alice
 - find a set of users (peers/nearest neighbors) who liked the same items as Alice in the past **and** who have rated item i
 - use, e.g. the average of their ratings to predict, if Alice will like item i
 - do this for all items Alice has not seen and recommend the best-rated

- **Basic assumption and idea**

- If users had similar tastes in the past they will have similar tastes in the future
- User preferences remain stable and consistent over time

Common Characteristics

- **Collection of Ratings**
- **Measure of Inter-User Agreement**
 - Correlation, Vector Cosine
- **Personalized Recommendations/Predictions**
 - Weighted Combinations of Others' Ratings
- **Tweaks to make things work right ...**
 - Neighborhood limitations
 - Normalization
 - Dealing with limited co-ratings

Implementation process (predicting ratings)

- **Given a set of items I , and a set of users U , and a sparse matrix of ratings R , We compute the prediction $s(u, i)$ as follows:**
 - For all users $v \neq u$, compute w_{uv}
 - similarity metric (e.g., Pearson correlation)
 - Select a neighborhood of users $V \subset U$ with highest w_{uv}
 - may limit neighborhood to top-k neighbors
 - may limit neighborhood to $\text{sim} > \text{sim_threshold}$
 - may limit neighborhood to people who rated i (necessary)
 - Compute prediction

$$s(u, i) = \bar{r}_u + \frac{\sum_{v \in V} (r_{vi} - \bar{r}_v) * w_{uv}}{\sum_{v \in V} w_{uv}}$$

Implementation process (predicting labels)

- What if we want to predict class label r , but not score $s(u, i)$?
- u_{ir} : the weight of rating r for user u to item i ,

$$u_{ir} = \sum_{v \in V} \sigma(r_{vi} = r) w_{uv}$$

- Return the class with the maximal weight,

$$r = \arg \max_r u_{ir}$$

Implementation Issues

- **Given $m=|U|$ users and $n=|I|$ items:**
 - Computation can be a bottleneck
 - Correlation between two users is $O(n)$
 - All correlations for a user is $O(mn)$
 - All pairwise correlations is $O(m^2n)$
 - Recommendations at least $O(mn)$
 - Lots of ways to make more practical
 - Many pairs have no overlaps (i.e., intersected items)
 - Calculation can be avoided

Example of UserCF

- **Example**

- A database of ratings of the current user, Alice, and some other users is given:

	Item1	Item2	Item3	Item4	Item5
Alice	5	3	4	4	?
User1	3	1	2	3	3
User2	4	3	4	3	5
User3	3	3	1	5	4
User4	1	5	5	2	1

- Determine whether Alice will like or dislike *Item5*, which Alice has not yet rated or seen

User-based nearest-neighbor collaborative filtering

■ Some first questions

- How do we measure similarity?
- How many neighbors should we consider?
- How do we generate a prediction from the neighbors' ratings?



	Item1	Item2	Item3	Item4	Item5
Alice	5	3	4	4	?
User1	3	1	2	3	3
User2	4	3	4	3	5
User3	3	3	1	5	4
User4	1	5	5	2	1

Measuring user similarity (1)

- **A popular similarity measure in user-based CF: Pearson correlation**

a, b : users

$r_{a,p}$: rating of user a for item p

P : set of items, rated both by a and b

– Possible similarity values between -1 and 1

$$\text{sim}(a, b) = \frac{\sum_{p \in P} (r_{a,p} - \bar{r}_a)(r_{b,p} - \bar{r}_b)}{\sqrt{\sum_{p \in P} (r_{a,p} - \bar{r}_a)^2} \sqrt{\sum_{p \in P} (r_{b,p} - \bar{r}_b)^2}}$$

Measuring user similarity (1)

- A popular similarity measure in user-based CF: Pearson correlation

a, b : users

$r_{a,p}$: rating of user a for item p

P : set of items, rated both by a and b

- Possible similarity values between -1 and 1

	Item1	Item2	Item3	Item4	Item5
Alice	5	3	4	4	?
User1	3	1	2	3	3
User2	4	3	4	3	5
User3	3	3	1	5	4
User4	1	5	5	2	1



sim = 0,85

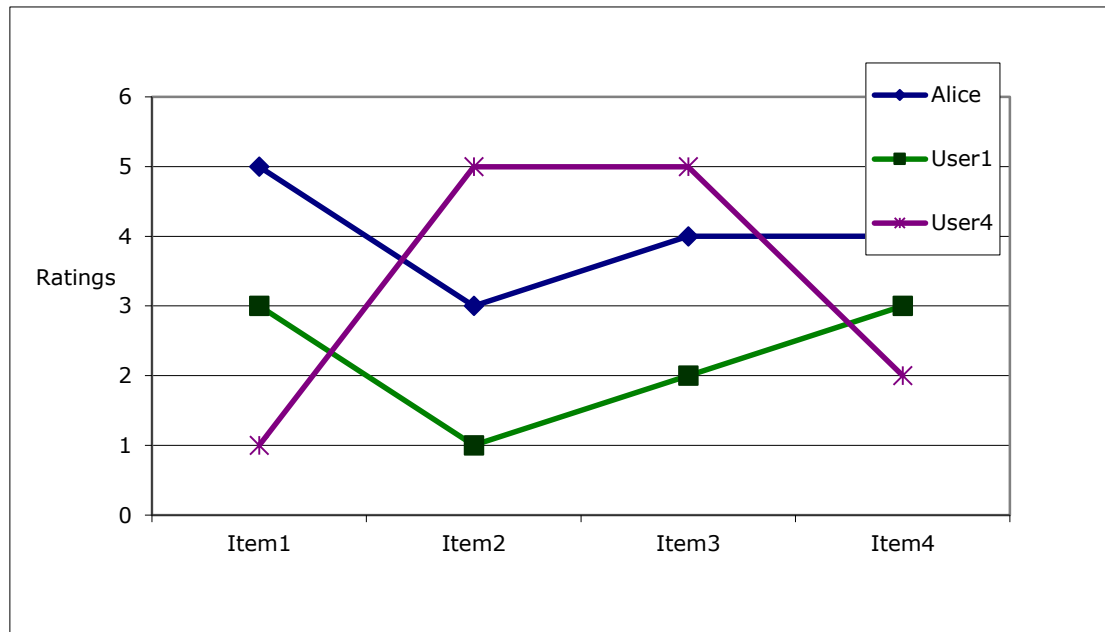
sim = 0,00

sim = 0,70

sim = -0,79

Pearson correlation

- Takes differences in rating behavior into account



- Works well in usual domains, compared with alternative measures
 - such as cosine similarity

Measuring user similarity (2)

- **A popular similarity measure in user-based CF: Jaccard Similarity**
 - a, b : users
 - $N(a)$: collections of items user a interacted with positive feedback
 - $N(b)$: collections of items user b interacted with positive feedback

$$\textit{sim}(a, b) = \frac{|N(a) \cap N(b)|}{|N(a) \cup N(b)|}$$

Measuring user similarity (3)

- **A popular similarity measure in user-based CF: Cosine Similarity**
 - a, b : users
 - $N(a)$: collections of items user a interacted with positive feedback
 - $N(b)$: collections of items user b interacted with positive feedback

$$\text{sim}(a, b) = \frac{|N(a) \cap N(b)|}{\sqrt{|N(a)||N(b)|}}$$

Measuring user similarity (3)

- A popular similarity measure in user-based CF: Cosine Similarity

A	a	b	d
B	a	c	
C	b	e	
D	c	d	e

$$w_{AB} = \frac{|\{a,b,d\} \cap \{a,c\}|}{\sqrt{|\{a,b,d\}| |\{a,c\}|}} = \frac{1}{\sqrt{6}}$$

$$w_{AC} = \frac{|\{a,b,d\} \cap \{b,e\}|}{\sqrt{|\{a,b,d\}| |\{b,e\}|}} = \frac{1}{\sqrt{6}}$$

$$w_{AD} = \frac{|\{a,b,d\} \cap \{c,d,e\}|}{\sqrt{|\{a,b,d\}| |\{c,d,e\}|}} = \frac{1}{3}$$

Making predictions

- A common prediction function:

$$pred(a, p) = \bar{r}_a + \frac{\sum_{b \in N} sim(a, b) * (r_{b,p} - \bar{r}_b)}{\sum_{b \in N} sim(a, b)}$$



- Calculate, whether the neighbors' ratings for the unseen item i are higher or lower than their average
- Combine the rating differences – use the similarity with a as a weight
- Add/subtract the neighbors' bias from the active user's average and use this as a prediction

Measuring user similarity (4)

- **A improved similarity measure in user-based CF:**

- Improved Cosine Similarity (User-IIF)
- Motivation: The similarity of interest is better illustrated by the fact that two users have done the same thing with less popular items
- a, b : users
- $N(a)$: collections of items user a interacted with positive feedback
- $N(b)$: collections of items user b interacted with positive feedback
- $N(i)$: collections of users who have interacted with item i

$$\text{sim}(a, b) = \frac{\sum_{i \in N(a) \cap N(b)} \frac{1}{\log(1 + |N(i)|)}}{\sqrt{|N(a)| |N(b)|}}$$

- **This formula penalizes the effect of popular items on similarity measure with $\frac{1}{\log(1 + |N(i)|)}$**

Making predictions

- To improve the novelty of the recommended results, we can reduce the weight of the popular items in the recommended results
- Simply, we can use formula like this:

$$pred(a, p) = \frac{pred(a, p)}{\log(1 + \alpha * popularity(p))}$$



- To achieve novelty in recommendation results, it is not enough to reduce the weight of popular items at the end, but to consider novelty in all parts of the recommendation engine

$$r_{b,p} - \overline{r_b} = \frac{r_{b,p} - \overline{r_b}}{\log(1 + \alpha * popularity(p))}$$

Measuring user similarity (5)

- **An improved similarity measure in user-based CF:**

- Introduce time decay factor
- Motivation: The longer it takes for user a and user b to act on item i , the less similar the two users' interests will be.
- a, b : users
- $N(a)$: collections of items user a interacted with positive feedback
- $N(b)$: collections of items user b interacted with positive feedback

$$sim(a, b) = \frac{\sum_{i \in N(a) \cap N(b)} \frac{1}{1 + \alpha |t_{ai} - t_{bi}|}}{\sqrt{|N(a)| |N(b)|}}$$

- This formula introduces the effect of time factor on similarity measure with $\frac{1}{1 + \alpha |t_{ai} - t_{bi}|}$

Making predictions

- Prediction function with time decay factor
- If you think about the recent interests of users that are similar to user a 's interests, then function can be:



$$pred(a, p) = \left(\overline{r}_a + \frac{\sum_{b \in N} sim(a, b) * (r_{b,p} - \overline{r}_b) * \frac{1}{1 + \alpha |t_0 - t_{b,p}|}}{\sum_{b \in N} sim(a, b)} \right)$$

More Discussions about UCF

Improving the metrics / prediction function

- **Not all neighbor ratings might be equally "valuable"**
 - Agreement on commonly liked items is not so informative as agreement on controversial items
 - **Possible solution:** Give more weights to items that have a higher variance
- **Value of number of co-rated items**
 - Use "significance weighting", by e.g., linearly reducing the weight when the number of co-rated items is low
- **Case amplification**
 - Intuition: Give more weights to "very similar" neighbors, i.e., where the similarity value is close to 1.

Common Normalizations

- Subtract user mean rating
- Convert to z-score (1 = 1 standard deviation above mean)
- Subtract item or item-user mean

Selecting Neighborhoods

- All the neighbors
- Threshold similarity or distance
- Random neighbors
- Top-N neighbors by similarity or distance
- Neighbors in a cluster

How Many Neighbors?

- **In theory, the more the better**
 - If you have a good similarity metric
- **In practice, noise from dissimilar neighbors decreases usefulness**
- **Depends on datasets**
 - 5-100

Scoring from Neighborhoods

- Average
 - Weighted average
 - Multiple linear regression
-
- Weighted average is common, simple, and works well

The motivation of item-based collaborative filtering

- **User-User CF was great, except ...**
- **Issues of Sparsity**
 - With large item sets, small numbers of ratings, too often there are points where no recommendation can be made (for a user, for an item to a set of users, etc.)
 - Many solutions proposed here, including “filterbots”, item-item, and dimensionality reduction
- **Computational performance**
 - With millions of users (or more), computing all pairs correlations is expensive
 - Even incremental approaches are expensive
 - And user profiles could change quickly – needed to compute in real time to keep users happy

Insight of Item-based collaborative filtering

- **Item-Item similarity is fairly stable**
 - This is dependent on having many more users than items
 - Average item has many more ratings than an average user
 - Intuitively, items don't generally change rapidly – at least not in ratings space (special case for time-bound items)
- **Item similarity is a route to compute a prediction of a user's item preference**

Implementation Process

- **Two step process:**
 - Compute similarity between pairs of items
 - Correlation between rating vectors
 - co-rated cases only (only useful for multi-level ratings)
 - Cosine of item rating vectors
 - can be used with multi-level or unary ratings
 - or adjusted ratings (normalize before computing cosine)
 - Some use conditional probability (unary)
 - Predict user-item rating
 - Weighted sum of rated “item-neighbors”
 - Linear regression to estimate rating

Item-based collaborative filtering

- **Basic idea:**
 - Use the similarity between items (and not users) to make predictions
- **Example:**
 - Look for items that are similar to Item5
 - Take Alice's ratings for these items to predict the rating for Item5

	Item1	Item2	Item3	Item4	Item5
Alice	5	3	4	4	?
User1	3	1	2	3	3
User2	4	3	4	3	5
User3	3	3	1	5	4
User4	1	5	5	2	1

Core Assumptions/Limitations

- **Item-item relationships need to be stable ...**
 - Mostly a corollary of stable user preferences
 - Could have special cases that are difficult (e.g., calendars, short-lived books, etc.)
 - Many of these issues are general temporal issues
- **Main limitation/complaint: lower serendipity**
 - This is a user/researcher complaint, not fully studied; intuition is clear

The cosine similarity measure

- Ratings are seen as vectors in n-dimensional space
- Similarity is calculated based on the angle between the vectors

$$\text{sim}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}| * |\vec{b}|}$$



- **Adjusted cosine similarity**
 - take average user ratings into account, transform the original ratings
 - U : set of users who have rated both items a and b

$$\text{sim}(\vec{a}, \vec{b}) = \frac{\sum_{u \in U} (r_{u,a} - \bar{r}_a)(r_{u,b} - \bar{r}_b)}{\sqrt{\sum_{u \in U} (r_{u,a} - \bar{r}_a)^2} \sqrt{\sum_{u \in U} (r_{u,b} - \bar{r}_b)^2}}$$



Inverse User Frequency

- **An improved similarity measure in item-based CF:**

- Improved Cosine Similarity (Item CF-IUF)
- Motivation: Active users should contribute less to item similarity than inactive users
- a, b : items
- $N(a)$: collections of users item a interacted with positive feedback
- $N(b)$: collections of users item b interacted with positive feedback
- $N(u)$: collections of items which have interacted with user u

$$\text{sim}(a, b) = \frac{\sum_{u \in N(a) \cap N(b)} \frac{1}{\log(1 + |N(u)|)}}{\sqrt{|N(a)| |N(b)|}}$$

- **This formula penalizes the effect of popular items on similarity measure with $\frac{1}{\log(1 + |N(u)|)}$**

Measuring item similarity

- **An improved similarity measure in item-based CF:**

- Introduce time decay factor
- Motivation: The longer it takes for user u to act on item a and b , the less similar the two items will be.
- a, b : items
- $N(a)$: collections of users item a interacted with positive feedback
- $N(b)$: collections of users item b interacted with positive feedback
- If a system's user interest changes quickly, take a larger alpha, otherwise a smaller alpha

$$\text{sim}(a, b) = \frac{\sum_{u \in N(a) \cap N(b)} \frac{1}{1 + \alpha |t_{ua} - t_{ub}|}}{\sqrt{|N(a)| |N(b)|}}$$

- This formula introduces the effect of time factor on similarity measure with $\frac{1}{1 + \alpha |t_{ua} - t_{ub}|}$

Making predictions

- A common prediction function:

$$\mathit{pred}(u, p) = \frac{\sum_{i \in \mathit{ratedItem}(u)} \mathit{sim}(i, p) * r_{u,i}}{\sum_{i \in \mathit{ratedItem}(u)} \mathit{sim}(i, p)}$$



- Neighborhood size is typically also limited to a specific size
- Not all neighbors are taken into account for the prediction
- An analysis of the MovieLens dataset indicates that "in most real-world situations, a neighborhood of 20 to 50 neighbors seems reasonable" (Herlocker et al. 2002)

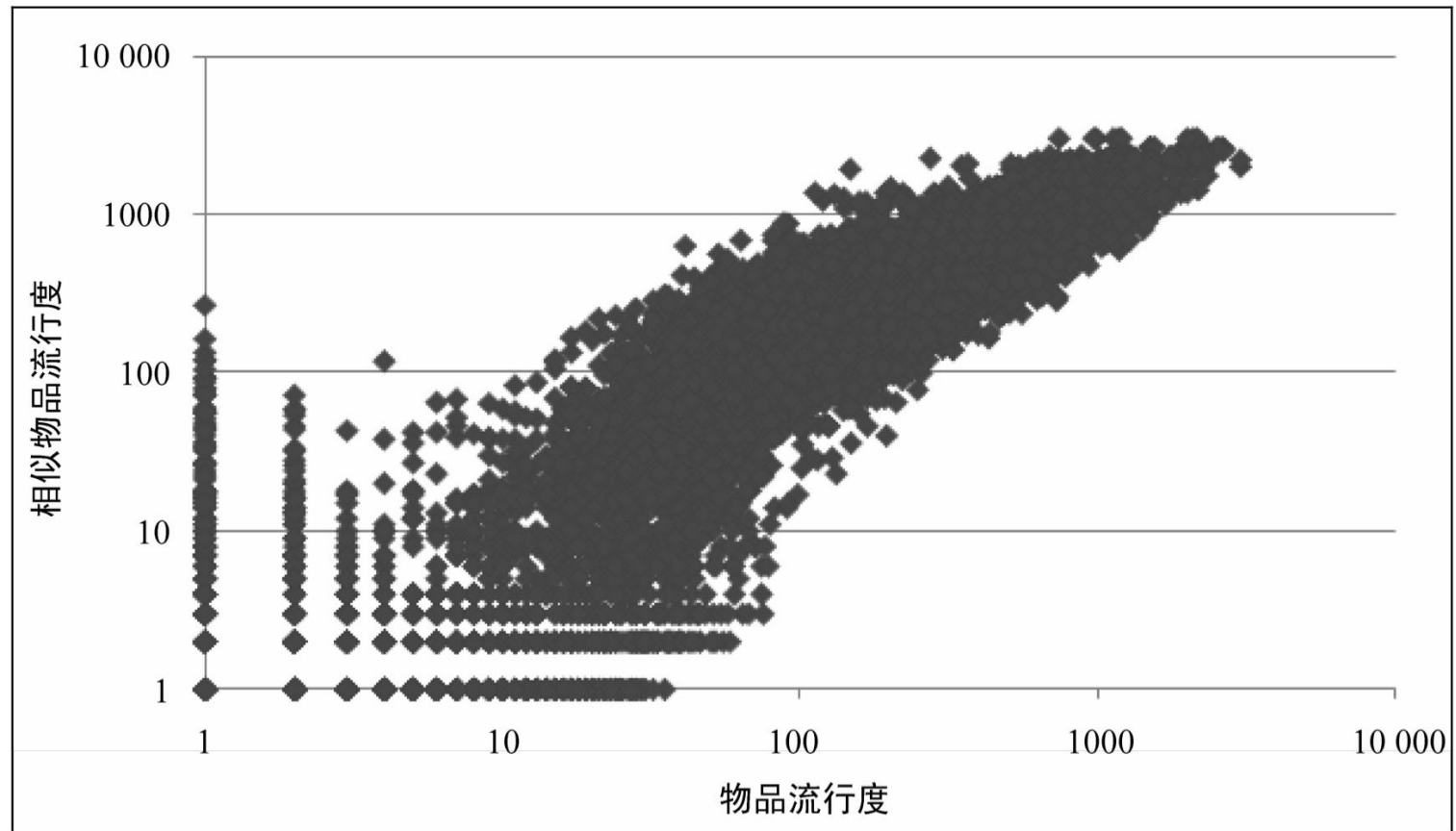
Making predictions

- Prediction Function with time decay factor
- In general, the user's current behavior should have more to do with the user's recent behavior, then function can be:

$$pred(u, p) = \frac{\sum_{i \in ratedItem(u)} sim(i, p) * r_{u,i} * \frac{1}{1 + \beta |t_0 - t_{ui}|}}{\sum_{i \in ratedItem(u)} sim(i, p)}$$



Making predictions



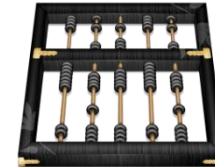
Making predictions

- Hot items tend to be similar to hot items, while less popular items tend to be similar to less popular items
- Considering that the recommendation system is to introduce unfamiliar items to users, it is assumed that if users know item i and have behaviors towards item i , then those items which is similar to item i and more popular than item i should have a relatively high probability to know, so the weight of such items can be reduced:

$$sim(i, p) = \frac{sim(i, p)}{\log(1 + \alpha * popularity(p))} (popularity(p) > popularity(i))$$

Making predictions

$$pred(u, p) = \frac{pred(u, p)}{\log(1 + \alpha * popularity(p))}$$
$$r_{u,i} = \frac{r_{u,i}}{\log(1 + \alpha * popularity(i))}$$



- The formulas above have been introduced in user-based cf, here we just review these two formulas which is proposed to improve novelty

Normalization of Item Similarity

- Evaluation of Item-based Top-N Recommendation Algorithms(George Karypis et.al 2001) indicates if the similarity matrix of Item CF is normalized by the maximum value, the accuracy of recommendation can be improved, which can be formulated as

$$\mathit{sim}(a, b)' = \frac{\mathit{sim}(a, b)}{\mathit{max}_b \mathit{sim}(a, b)}$$

- In fact, the benefits of normalization are not only to increase the accuracy of recommendations, but also to improve the coverage and diversity of recommendations

Pre-processing for item-based filtering

- **Item-based filtering does not solve the scalability problem itself**
- **Pre-processing approach by Amazon.com (in 2003)**
 - Calculate all pair-wise item similarities in advance
 - The neighborhood to be used at run-time is typically rather small, because only items are taken into account which the user has rated
 - Item similarities are supposed to be more stable than user similarities
- **Memory requirements**
 - Up to $O(N^2)$ pair-wise similarities to be memorized (N = number of items) in theory
 - In practice, this is significantly lower (items with no co-ratings)
 - Further reductions possible
 - Minimum threshold for co-ratings
 - Limit the neighborhood size (might affect recommendation accuracy)

Picking Neighbors for item-based filtering

- Score formula has neighborhood $N(i; u)$
- Neighbors are usually k most similar items
 - That the user has rated (hence u)
- Good value of k important
 - k too small -> inaccurate scores
 - k too large -> too much noise
 - $k = 20$ often works well

Item-Item Top-N

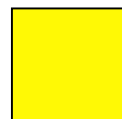
- **Item-Item similarity model can be used to compute top-N directly:**
 - Simplify model by limiting items to small “neighborhoods” of k most-similar items (e.g., 20)
 - For a profile set of items, compute/merge/sort the k-most similar items for each profile item
 - Straightforward matrix operation from Deshpande and Karypis (Mukund Deshpande and George Karypis. 2004. Item-based top-N recommendation algorithms. ACM Trans. Inf. Syst. 22, 1 (January 2004), 143-177.)

Item-Item CF ($|N|=2$)

		users											
		1	2	3	4	5	6	7	8	9	10	11	12
movies	1	1		3			5			5		4	
	2			5	4			4			2	1	3
	3	2	4		1	2		3		4	3	5	
	4		2	4		5			4			2	
	5			4	3	4	2					2	5
	6	1		3		3			2			4	



- unknown rating



- rating between 1 to 5

Item-Item CF ($|N|=2$)

		users											
		1	2	3	4	5	6	7	8	9	10	11	12
movies	1	1		3		?	5			5		4	
	2			5	4			4			2	1	3
	3	2	4		1	2		3		4	3	5	
	4		2	4		5			4			2	
	5			4	3	4	2					2	5
	6	1		3		3			2			4	



- estimate rating of movie 1 by user 5

Item-Item CF ($|N|=2$)

		users												
		1	2	3	4	5	6	7	8	9	10	11	12	sim(1,m)
movies	1	1		3		?	5			5		4		1.00
	2			5	4			4			2	1	3	-0.18
	<u>3</u>	2	4		1	2		3		4	3	5		<u>0.41</u>
	4		2	4		5			4			2		-0.10
	5			4	3	4	2					2	5	-0.31
	<u>6</u>	1		3		3			2			4		<u>0.59</u>

Neighbor selection:
Identify movies similar to
movie 1, rated by user 5

Here we use Pearson correlation as similarity:
1) Subtract mean rating m_i from each movie i
 $m_1 = (1+3+5+5+4)/5 = 3.6$
 row 1: [-2.6, 0, -0.6, 0, 0, 1.4, 0, 0, 1.4, 0, 0.4, 0]
 2) Compute cosine similarities between rows

Item-Item CF ($|N|=2$)

		users												
		1	2	3	4	5	6	7	8	9	10	11	12	sim(1,m)
movies	1	1		3		?	5			5		4		1.00
	2			5	4			4			2	1	3	-0.18
	<u>3</u>	2	4		1	2		3		4	3	5		<u>0.41</u>
	4		2	4		5			4			2		-0.10
	5			4	3	4	2					2	5	-0.31
	<u>6</u>	1		3		3			2			4		<u>0.59</u>

Compute similarity weights:

$s_{1,3}=0.41$, $s_{1,6}=0.59$

Item-Item CF ($|N|=2$)

		users											
		1	2	3	4	5	6	7	8	9	10	11	12
movies	1	1		3		2.6	5			5		4	
	2			5	4			4			2	1	3
	<u>3</u>	2	4		1	2		3		4	3	5	
	4		2	4		5			4			2	
	5			4	3	4	2					2	5
	<u>6</u>	1		3		3			2			4	

Predict by taking weighted average:

$$r_{1.5} = (0.41 \cdot 2 + 0.59 \cdot 3) / (0.41 + 0.59) = 2.6$$

$$r_{ix} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{jx}}{\sum s_{ij}}$$

Benefits of Item-based collaborative filtering

- **It actually works quite well**
 - Good prediction accuracy
 - Good performance on top-N predictions
- **Efficient implementation**
 - At least in cases where $|U| \gg |I|$
 - Benefits of precomputability
- **Broad applicability and flexibility**
 - As easy to apply to a shopping cart as to a user profile

Early Experiences

- **Item-Item Very Successful in Commercial Applications**
 - Amazon used Item-Item widely
 - Claims great success in recommendation
 - Helps people find products of interest
- **Big Disappointment in MovieLens**
 - MovieLens users were switched, and many complained
 - Claimed that recommendations were too obvious
 - Lack of bold recommendations and predictions

What Was Learned?

- **Item-Item is much faster and more stable for domains with many more users than items**
 - Stability makes it possible to pre-compute and store item-item correlations
- **Item-Item is also substantially more “conservative” in its recommendations and predictions**
 - Can be good for shopping, consumption tasks
 - May be frustrating for browsing/entertainment

The problem of item-based collaborative filtering

- **Item-Item is not the same as User-User**
 - Very difficult for item-item to discover highly different items to recommend
 - User-User by default will elevate items that a close neighbor loves, even without much evidence
 - Another consequence was that item-item predictions tended to be less extreme (since they were grounded in more data)

The extensibility of item-based collaborative filtering

- Item-item is good for extending directly
- Simple parts with well-defined interfaces provide a lot of flexibility
- It's easy to understand what extensions do

The extensibility of item-based collaborative filtering: example

- **Goal: incorporate user trustworthiness into item relatedness computation**
 - User's global reputation, not per-user trust
- **Solution: weight users by trust before computing item similarities**
- **High-trust users have more impact**
- **Massa and Avesani. 2004. 'Trust-Aware Collaborative Filtering for Recommender Systems'**

The extensibility of item-based collaborative filtering: example

- **Recommending research papers: useful to consider items as users who purchase the paper's citations**
 - Same idea can apply to web pages
- **Goal: incorporate paper 'importance' into recommender**
- **Solution: weight papers by the paper's PageRank (or HITS hub score)**
- **Ekstrand et al., 2010. Automatically Building Research Reading Lists.**

Restructuring: Item-Item CBF

- Basic item-item algorithm structure doesn't care how similarity is computed
- So why not use content-based similarity?
- Resulting algorithm really isn't a collaborative filter
- But it can work pretty well!
- Example: using Lucene to compare documents as neighborhood & similarity function

The Difference between UserCF and ItemCF

- UserCF's recommendation results focus on hot topics that reflect a small group of users with similar interests, while ItemCF's recommendation results focus on maintaining users' historical interests.
- In other words, UserCF's recommendations are more social, reflecting the popularity of items in the user's small interest group, while ItemCF's recommendations are more personalized, reflecting the user's own interest inheritance.

The choice between UserCF and ItemCF

- **One consideration: similarity computation**
 - Computational complexity
 - Ensure reliable computation
- **When number of users much larger than items**
 - E.g., Amazon
 - **ItemCF**
- **When number of items much larger than users**
 - E.g., paper recommender system
 - **UserCF**

Pros/Cons of Collaborative Filtering

- **+ Works for any kind of item**
 - No feature selection needed
- **- Cold Start:**
 - Need enough users in the system to find a match
- **- Sparsity:**
 - The user/ratings matrix is sparse
 - Hard to find users that have rated the same items
- **- First rater:**
 - Cannot recommend an item that has not been previously rated
 - New items, Esoteric items
- **- Popularity bias:**
 - Cannot recommend items to someone with unique taste
 - Tends to recommend popular items

More on ratings – Explicit ratings

- Probably the most precise ratings
 - Most commonly used (1 to 5, 1 to 7 Likert response scales)
 - Research topics
 - Optimal granularity of scale; indication that 10-point scale is better accepted in movie dom.
 - An even more fine-grained scale was chosen in the joke recommender discussed by Goldberg et al. (2001), where a continuous scale (from -10 to +10) and a graphical input bar were used
 - No precision loss from the discretization
 - User preferences can be captured at a finer granularity
 - Users actually "like" the graphical interaction method
 - Multidimensional ratings (multiple ratings per movie such as ratings for actors and sound)
 - Main problems
 - Users not always willing to rate many items
 - number of available ratings could be too small → sparse rating matrices → poor recommendation quality
 - How to stimulate users to rate more items?
-

More on ratings – Implicit ratings

- Typically collected by the web shop or application in which the recommender system is embedded
- When a customer buys an item, for instance, many recommender systems interpret this behavior as a positive rating
- Clicks, page views, time spent on some page, downloads ...
- Implicit ratings can be collected constantly and do not require additional efforts from the side of the user
- Main problem
 - One cannot be sure whether the user behavior is correctly interpreted
 - For example, a user might not like all the books he or she has bought; the user also might have bought a book for someone else
- Implicit ratings can be used in addition to explicit ones; question of correctness of interpretation

Data sparsity problems

- **Cold start problem**

- How to recommend new items? What to recommend to new users?

- **Straightforward approaches**

- Ask/force users to rate a set of items
 - Use another method (e.g., content-based, demographic or simply non-personalized) in the initial phase

- **Alternatives**

- Use better algorithms (beyond nearest-neighbor approaches)
 - Example:
 - In nearest-neighbor approaches, the set of sufficiently similar neighbors might be too small to make good predictions
 - Assume "transitivity" of neighborhoods to expand the neighbor size

Example algorithms for sparse datasets

■ Recursive CF (Zhang and Pu 2007)

- Assume there is a very close neighbor n of u who however has not rated the target item i yet.
- Idea:
 - Apply CF-method recursively and predict a rating for item i for the neighbor
 - Use this predicted rating instead of the rating of a more distant direct neighbor

	Item1	Item2	Item3	Item4	Item5
Alice	5	3	4	4	?
User1	3	1	2	3	?
User2	4	3	4	3	5
User3	3	3	1	5	4
User4	1	5	5	2	1

sim = 0.85

Predict rating for User1

Code Reading

Slides from Julian (UCSD)

Code: reading the data

Read the data (slightly larger dataset than before):

```
In [1]: import gzip
        from collections import defaultdict
        import random
        import numpy
        import scipy.optimize
```

```
In [2]: path = "/home/jmcauley/datasets/mooc/amazon/amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz"
```

```
In [3]: f = gzip.open(path, 'rt', encoding="utf8")
```

```
In [4]: header = f.readline()
        header = header.strip().split('\t')
```

Code: reading the data

Our goal is to make recommendations of products based on users' purchase histories. The only information needed to do so is **user and item IDs**

```
In [5]: dataset = []
```

```
In [6]: for line in f:
        fields = line.strip().split('\t')
        d = dict(zip(header, fields))
        d['star_rating'] = int(d['star_rating'])
        d['helpful_votes'] = int(d['helpful_votes'])
        d['total_votes'] = int(d['total_votes'])
        dataset.append(d)
```

```
In [7]: dataset[0]
```

```
Out[7]: {'marketplace': 'US',
        'customer_id': '45610553',
        'review_id': 'RMDCHH62VB6',
        'product_id': 'B00HH62VB6',
        'product_parent': '618218723',
        'product_title': 'AGPtek® 10 Isolated Output 9V 12V 18V Guitar Pedal Board Power Supply Effect Pedals
        with Isolated Short Cricuit / Overcurrent Protection',
```

Code: useful data structures

Build data structures representing the set of items for each user and users for each item:

```
In [8]: # Useful data structures
```

```
In [9]: usersPerItem = defaultdict(set)  
itemsPerUser = defaultdict(set)
```

U_i

I_u

```
In [10]: itemNames = {}
```

```
In [11]: for d in dataset:  
    user,item = d['customer_id'], d['product_id']  
    usersPerItem[item].add(user)  
    itemsPerUser[user].add(item)  
    itemNames[item] = d['product_title']
```

Code: Jaccard similarity

The Jaccard similarity implementation follows the definition directly:

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

```
In [12]: def Jaccard(s1, s2):  
         numer = len(s1.intersection(s2))  
         denom = len(s1.union(s2))  
         return numer / denom
```

Recommendation

We want a recommendation function that return **items similar to a candidate item i** . Our strategy will be as follows:


- Find the set of users who purchased i
- Iterate over all other items other than i
- For all other items, compute their similarity with i
(*and store it*)
- Sort all other items by (Jaccard) similarity
 - Return the most similar

Code: recommendation

Now we can implement the recommendation function itself:

```
In [13]: def mostSimilar(i):  
    similarities = []  
    users = usersPerItem[i]  
    for i2 in usersPerItem:  
        if i2 == i: continue  
        sim = Jaccard(users, usersPerItem[i2])  
        similarities.append((sim,i2))  
    similarities.sort(reverse=True)  
    return similarities[:10]
```

$Jaccard(U_i, U_j) = \frac{|U_i \cap U_j|}{|U_i \cup U_j|}$



Code: recommendation

Next, let's use the code to make a recommendation.
The query is just a product ID:

```
In [14]: dataset[2]
```

```
Out[14]: {'marketplace': 'US',  
          'customer_id': '6111003',  
          'review_id': 'RIZR67JKUDBI0',  
          'product_id': 'B0006VMBHI',  
          'product_parent': '603261968',  
          'product_title': 'AudioQuest LP record clean brush',  
          'product_category': 'Musical Instruments',  
          'star_rating': 3,  
          'helpful_votes': 0,  
          'total_votes': 1,  
          'vine': 'N',  
          'verified_purchase': 'Y',  
          'review_headline': 'Three Stars',  
          'review_body': 'removes dust. does not clean',  
          'review_date': '2015-08-31'}
```

```
In [15]: query = dataset[2]['product_id']
```

Code: recommendation

Next, let's use the code to make a recommendation.
The query is just a product ID:

```
In [16]: mostSimilar(query)
```

```
Out[16]: [(0.028446389496717725, 'B00006I5SD'),  
(0.01694915254237288, 'B00006I5SB'),  
(0.015065913370998116, 'B000AJR482'),  
(0.014204545454545454, 'B00E7MVP3S'),  
(0.008955223880597015, 'B001255YL2'),  
(0.008849557522123894, 'B003EIRV08'),  
(0.008333333333333333, 'B0015VEZ22'),  
(0.00821917808219178, 'B00006I5UH'),  
(0.008021390374331552, 'B00008BWM7'),  
(0.007656967840735069, 'B000H2BC4E')]
```

Recommending more efficiently

Our implementation was not very efficient. The slowest component is the iteration over all other items:

- Find the set of users who purchased i
- **Iterate over all other items other than i**
- For all other items, compute their similarity with i
(*and store it*)
 - Sort all other items by (Jaccard) similarity
 - Return the most similar

This can be done more efficiently as most items will have no overlap

Recommending more efficiently

In fact it is sufficient to iterate over **those items purchased by one of the users who purchased i**

- Find the set of users who purchased i
 - **Iterate over all users who purchased i**
 - Build a candidate set from all items those users consumed
 - For items in this set, compute their similarity with i (*and store it*)
 - Sort all other items by (Jaccard) similarity
 - Return the most similar
-

Code: faster implementation

Our more efficient implementation works as follows:

```
In [19]: def mostSimilarFast(i):  
    similarities = []  
    users = usersPerItem[i]  
    candidateItems = set()  
    for u in users:  
        candidateItems = candidateItems.union(itemsPerUser[u])  
    for i2 in candidateItems:  
        if i2 == i: continue  
        sim = Jaccard(users, usersPerItem[i2])  
        similarities.append((sim,i2))  
    similarities.sort(reverse=True)  
    return similarities[:10]
```

Collaborative filtering for rating prediction

In the previous section we provided code to make recommendations based on the **Jaccard similarity**

How can the same ideas be used for rating prediction?

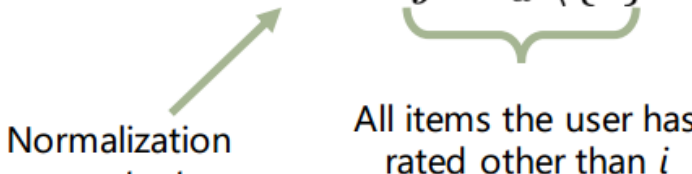
Collaborative filtering for rating prediction

A simple heuristic for rating prediction works as follows:

- The user (u)'s rating for an item i is a weighted combination of all of their previous ratings for items j
- The weight for each rating is given by the Jaccard similarity between i and j

Collaborative filtering for rating prediction

This can be written as:

$$r(u, i) = \frac{1}{Z} \sum_{j \in I_u \setminus \{i\}} r_{u,j} \cdot \text{sim}(i, j)$$


Normalization
constant

All items the user has
rated other than i

$$Z = \sum_{j \in I_u \setminus \{i\}} \text{sim}(i, j)$$


Code: CF for rating prediction

Now we can adapt our previous recommendation code to predict ratings

```
In [22]: # More utility data structures
```

```
In [23]: reviewsPerUser = defaultdict(list)
reviewsPerItem = defaultdict(list)
```

List of reviews per user and per item




```
In [24]: for d in dataset:
    user,item = d['customer_id'], d['product_id']
    reviewsPerUser[user].append(d)
    reviewsPerItem[item].append(d)
```

```
In [25]: ratingMean = sum([d['star_rating'] for d in dataset]) / len(dataset)
```

```
In [26]: ratingMean
```

```
Out[26]: 4.251102772543146
```

We'll use the mean rating as a baseline for comparison



Code: CF for rating prediction

Our rating prediction code works as follows:

In [27]: `def predictRating(user,item):`

`ratings = []`

`similarities = []`

`for d in reviewsPerUser[user]:`

`i2 = d['product_id']`

`if i2 == item: continue`

`ratings.append(d['star_rating'])`

`similarities.append(Jaccard(usersPerItem[item],usersPerItem[i2]))`

`if (sum(similarities) > 0):`


`weightedRatings = [(x*y) for x,y in zip(ratings,similarities)]`

`return sum(weightedRatings) / sum(similarities)`

`else:`

`# User hasn't rated any similar items`

`return ratingMean`

$$r(u,i) = \frac{1}{Z} \sum_{j \in I_u \setminus \{i\}} r_{u,j} \cdot \text{sim}(i,j)$$


Code: CF for rating prediction

As an example, select a rating for prediction:

```
In [28]: dataset[1]
```

```
Out[28]: {'marketplace': 'US',  
         'customer_id': '14640079',  
         'review_id': 'RZSL0BALIYUNU',  
         'product_id': 'B003LRN53I',  
         'product_parent': '986692292',  
         'product_title': 'Sennheiser HD203 Closed-Back DJ Headphones',  
         'product_category': 'Musical Instruments',  
         'star_rating': 5,  
         'helpful_votes': 0,  
         'total_votes': 0,  
         'vine': 'N',  
         'verified_purchase': 'Y',  
         'review_headline': 'Five Stars',  
         'review_body': 'Nice headphones at a reasonable price.',  
         'review_date': '2015-08-31'}
```

```
In [29]: u,i = dataset[1]['customer_id'], dataset[1]['product_id']
```

```
In [30]: predictRating(u, i)
```

```
Out[30]: 5.0
```

Code: CF for rating prediction

Similarly, we can evaluate accuracy across the entire corpus:

```
In [31]: def MSE(predictions, labels):  
         differences = [(x-y)**2 for x,y in zip(predictions,labels)]  
         return sum(differences) / len(differences)  
  
In [32]: alwaysPredictMean = [ratingMean for d in dataset]  
  
In [33]: cfPredictions = [predictRating(d['customer_id'], d['product_id']) for d in dataset]  
  
In [34]: labels = [d['star_rating'] for d in dataset]  
  
In [35]: MSE(alwaysPredictMean, labels)  
Out[35]: 1.4796142779564334  
  
In [36]: MSE(cfPredictions, labels)  
Out[36]: 1.6146130004291603
```

Collaborative filtering for rating prediction

Note that this is just a **heuristic** for rating prediction

- In fact in this case it did *worse* (in terms of the MSE) than always predicting the mean
 - We could adapt this to use:
 1. A different similarity function (e.g. cosine)
 2. Similarity based on users rather than items
 3. A different weighting scheme