Recommender systems and algorithms

Adaptive Decision Support Systems

Recommender systems

Basic architecture:

Input ("preference elicitation")

Clicks, rating, attribute weighing, critiquing, demographics

Processing ("recommender algorithm")

Knowledge-based, content-based, collaborative filtering

Output ("presentation")

Overall recommendations, related items, re-ranked (search) results

Recommender systems

Today we focus on the processing part

- The recommender algorithm

We will talk more about input and output later

- Keep in mind, these are actually more important than the algorithm!

Recommender systems

Basic data sources:

Users

Each user may have several attributes (e.g. demographics)

We may have a user model for each user

Items

Each item may have several attributes (e.g. price)

Ratings

Where users and items come together (can be implicit!)

Non-algorithmic

Non-algorithmic solutions can work surprisingly well

Random items

Low quality, but high serendipity!

Sort by popularity ("global Top N")

Easy and predictable, but can lead to supply chain problems!

Sort by attribute(s), filter by attribute(s)

Supports non-compensatory decision-making

Knowledge-based

Knowledge-based recommenders provide hand-crafted recommendations

If(season = summer & user_age = 4 & user_gender = m) then recommend 4T_swimming_trunks

Pros and cons:

- Works really well in high-risk domains
- Easy to mix user, context, system attributes
- Manual effort; focused, detailed input required

Content-based

Multi-attribute utility theory (MAUT) recommenders

Idea comes from marketing / behavioral psychology

Each item has a (normalized) value on a number of attributes v_{ia}

User assigns a weight to each attribute w_{ua}

The system calculates the utility of each item for this user:

$$U_{iu} = \sum v_{ia} \cdot w_{ua}$$

Rank items by U_{iu}

Content-based

Pros and cons:

- Works well when item space has well-defined attributes
- More “compensatory” than attribute filtering / sorting

How do we get users’ attribute weights?

- Ask them directly (difficult for domain novices)
- Derive (average?) from liked products
- Other methods: see lecture 9.1

Content-based

What if the items don't have well-defined attributes?

Use item descriptions! Perform TF-IDF to extract important keywords

Perform cosine similarity with user-preferred keywords (either asked, or derived from liked items)

Collaborative filtering

	i1	i2	i3	i4	i5	i6
u1			2		4	
u2		2	3			
u3	1					
u4	5	2				
u5			4		3	
u6				1		

User-user CF

Predict a rating by summing across all* other users' ratings, weighted by similarity:

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

$\bar{r}_u$ = average rating of a user (to accommodate rating styles)

w_{uv} = weight (similarity between users u and v)

V = neighborhood (usually limited to k most similar users)

User-user CF

Pros and cons:

- No user or item attributes needed
- Sparsity: impossible to recommend to users with few (overlapping) ratings; items with no ratings won't be recommended
- Computationally intensive (but doable with some tricks)

Item-item CF

Compute pairwise item similarities

- Fairly stable, so this doesn't have to happen on the fly!

To predict a rating for an unrated item, compute a weighted sum of rated "item neighbors"

- Can also do more heuristic methods, like computing neighbors of rated items and merging them

Item-item CF

$$s(i;u) = \frac{\sum_{j \in N(i;u)} w_{ij}(r_{uj} - \bar{r}_j)}{\sum_{j \in N(i;u)} |w_{ij}|} + \bar{r}_i$$

$\bar{r}_i$ = average rating of an *item*

w_{ij} = weight (similarity between *items* i and j)

N = neighborhood (usually limited to k most similar *items* rated by user u)

Item-item CF

Pros and cons:

- Works great if you have more users than items
- Similarities can be pre-computed, low similarities can be discarded
- Not just for users (also similar items, shopping cart, etc.)
- Not for time-bound stuff (see context-based)
- Usually not high in serendipity

Matrix Factorization

Idea: “preferences” are abstract mental concepts

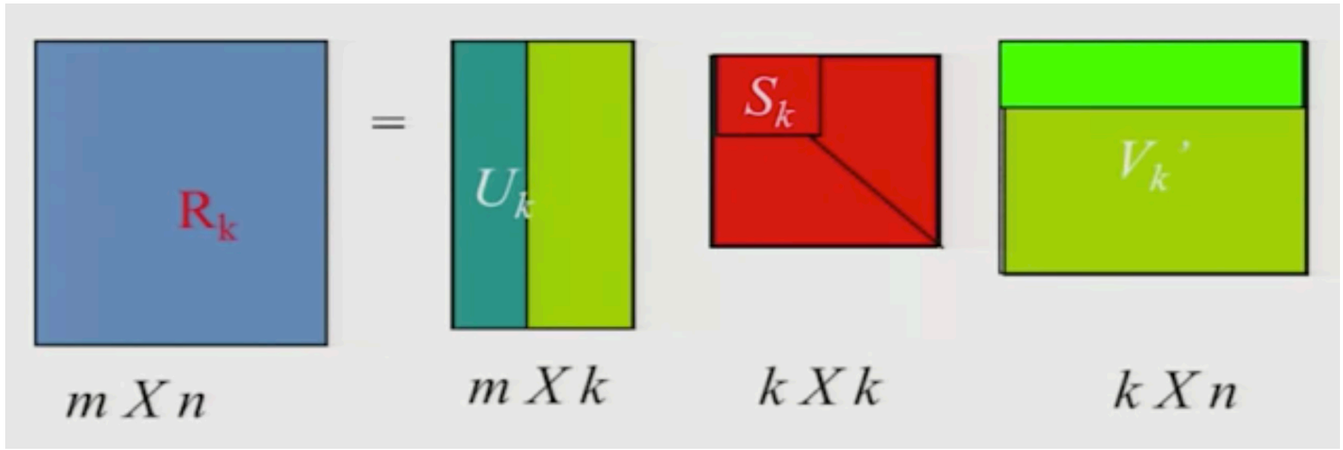
- Item ratings are not necessarily the best representation

Solution: extract latent features from the user-item rating matrix

- Technique: dimensional reduction (e.g. singular value decomposition)

Matrix Factorization

Use SVD to decompose R into U S V^T, truncate to k dimensions

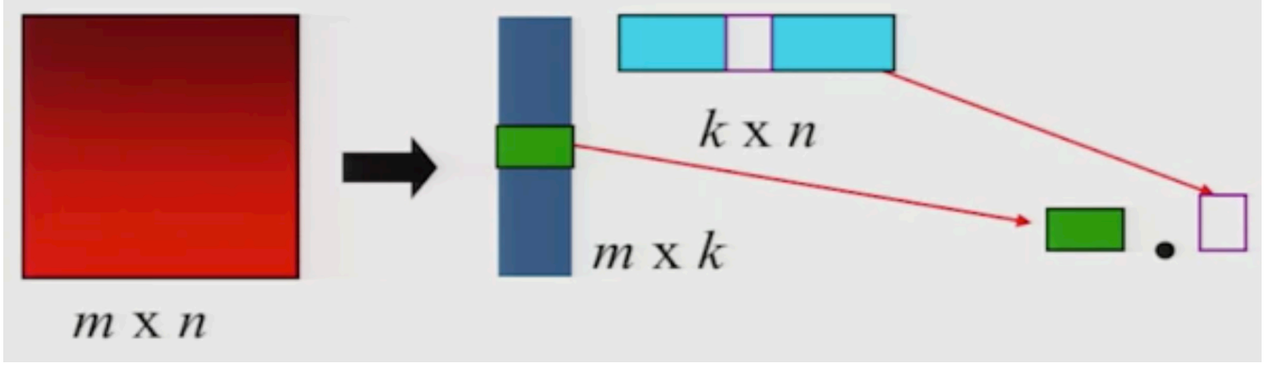


Here, U_k contains the feature weights of the users and V_k^T contains the feature values of the items; S_k has the importance of each feature on the diagonal

Note: these features are latent (not real, like in content-based methods!)

Matrix Factorization

The predicted rating for user *u* on item *i* is simply a summation over U S V^T for all *k* factors



Note: usually the SVD is done after subtracting a baseline, so this must be added back

Matrix Factorization

Pros and cons:

- Item representations can be pre-computed
- Low storage requirement
- Works well with sparsity
- Difficult to explain
- Model is hard to build (luckily this doesn't have to be done at runtime; new data can be folded in: $R\ V\ S^{-1} = U$)

Hybrid algorithms

Problem: cold start — items or users with no (or few) ratings

Solution 1: Use behavior (clicks) rather than ratings

But ratings are a more explicit form of preference than clicks!

Solution: combine different input signals — both implicit and explicit feedback

Change their relative weight as ratings accumulate

Hybrid algorithms

Problem: cold start — items or users with no (or few) ratings

Solution 2: Use content (e.g. attributes) rather than ratings

Example: Item-item CBF — use content to calculate w_{ij}

Again, CF and CBF can be combined into a single algorithm

Hybrid algorithms

Netflix Prize: reduce the RMSE* of the CineMatch algorithm by 10% (from 0.9525 to 0.8572 or less)

SVD: 0.8914, RBM: 0.8990

A linear blend of the two: 0.88

Final solution: an ensemble of 104 predictors developed by multiple teams

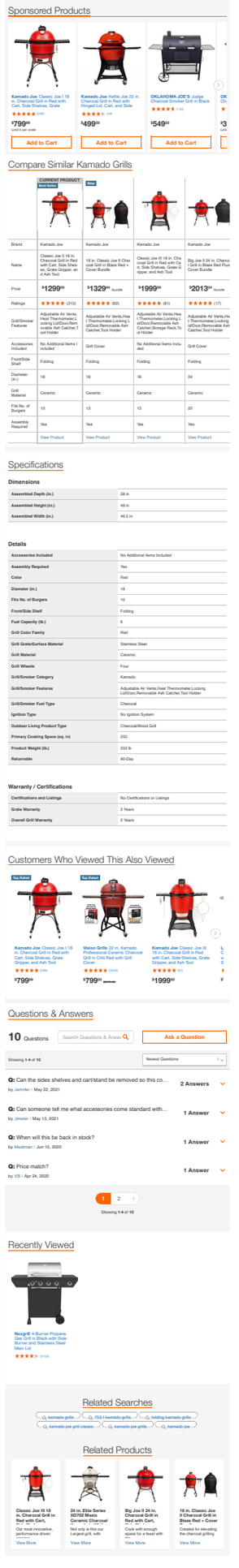
Combined using a neural network (meta-learner)

Other ensemble solutions: pre-filtering and re-ranking

Hybrid algorithms

Hybrids in real-life systems:

Multiple recommendation lists created by different algorithms (some can be external, i.e. ads)



Context-awareness

Problem: My taste in {music} depends on my {mood}

My interest in {restaurants} depends on my {location}

Solution: context-aware recommendations

Method 1: Filter or re-rank items based on the context

See also: personalized search results

Method 2: Create different user entries for different contexts

Use a blend of predictions to allow for multi-faceted contexts, e.g. $R_{(U|Clemson,happy)} = R_U + R_{(U|Clemson)} + R_{(U|happy)}$

Group recommenders

Problem: recommend a {restaurant} to a group of people

- How to take multiple preferences into account?
- Especially if they conflict?

Solutions:

- Aggregate the predictions: majority voting, average rating, fairness “zipper”, least misery, most pleasure, average without misery
- Concatenate the user models: concatenate the user data and use it to create a new user model

Group recommenders

Also possible: interactive solutions

- e.g. critiquing

Careful with explanations

- Social dynamics, privacy

Network-based

Idea: Leverage connections between people to inform the recommender

Example 1: limit the other users to “known” other users (social network)

Can help with explanation!

Example 2: use network distance as a trust measure

Can inform/replace calculation of w_{uv} in user-user CF

Can also be used to weigh w_{ij} calculation in item-item CF

Evaluation

How to evaluate algorithms?

Evaluation

Offline vs. online

- Offline: using a reference dataset
- Online: with live users

Accuracy vs. other metrics

- Accuracy: rating accuracy vs. ranking accuracy
- Other metrics: diversity, serendipity, coverage, user experience*

Accuracy

Basic setup: cross-validation

- Train the algorithm on a dataset, predict a hold-out set
- Variants: folds, leave-one-out
- Stratify by item frequency, user frequency?
- Take time into account?

Keep in mind: training data may be based on a system that has a recommender already!

- Persuasive effect of recommendation can bias the results

Rating accuracy

RMSE: $\sqrt{\text{mean}(r_{u,i} - \hat{r}_{u,i})^2}$

Basic version gives users with a large profile (too much) more weight

In a real system, we only really care about *good* predictions

- Being good at predicting the difference between 4 and 5 stars is more useful than predicting 1 vs. 2 stars
- But RMSE counts both errors equally!

Ranking accuracy

Tests the ability to rank top items in the correct order

- Precision@n and recall@n
- F_1-measure
- Mean Reciprocal Rank
- $NDCG_k$
- AUC_k

Think about fairness: is the accuracy evenly distributed?

Other metrics

Diversity: do users get recommendations that are diverse enough?

- Check based on item (latent) attributes

Serendipity: are the recommendations surprising?

- Novel (less-recommended) items

Coverage: is the entire item set recommended?

- Shannon entropy, Gini index, coverage

User experience: next week!

Before we adjourn

A few things to do before Wednesday

Before we adjourn

Fill out the “Knowledge and preference pre-test” on Canvas!

We will use your answers in the group formation process

Install the following tools/packages on your machine for the hands-on tutorial:

Python 3 (and a Python environment manager)

node.js v16

NPM v8

React v18

NVM, latest stable build (optional)

Questions…