Web Mining and Recommender Systems

Recommender Systems: Introduction
Why recommendation?

The goal of recommender systems is...
• To help people discover new content
Why recommendation?

The goal of recommender systems is...

- To help us find the content we were already looking for.

Are these recommendations good or bad?
Why recommendation?

The goal of recommender systems is...

• To discover which things go together
Why recommendation?

The goal of recommender systems is...

• To personalize user experiences in response to user feedback
Why recommendation?

The goal of recommender systems is...
• To recommend incredible products that are relevant to our interests
Why recommendation?

The goal of recommender systems is...

• To identify things that we **like**
The goal of recommender systems is...
• To help people discover new content
• To help us find the content we were already looking for
• To discover which things go together
• To personalize user experiences in response to user feedback
• To identify things that we like

To **model** people’s preferences, opinions, and behavior
Suppose we want to build a movie recommender.

e.g. which of these films will I rate highest?
Recommending things to people

We already have a few tools in our “supervised learning” toolbox that may help us

\[ f(\text{user features, movie features}) \rightarrow \text{star rating} \]
Recommending things to people

\[ f(\text{user features, movie features}) \rightarrow \text{star rating} \]
Recommending things to people

\[ f(\text{user features, movie features}) \rightarrow \text{star rating} \]

With the models we’ve seen so far, we can build predictors that account for...

- Do women give higher ratings than men?
- Do Americans give higher ratings than Australians?
- Do people give higher ratings to action movies?
- Are ratings higher in the summer or winter?
- Do people give high ratings to movies with Vin Diesel?

So what can’t we do yet?
Recommending things to people

\[ f(\text{user features, movie features}) \rightarrow \text{star rating} \]

Consider the following linear predictor (e.g. from week 1):

\[ f(\text{user features, movie features}) = \langle \phi(\text{user features}); \phi(\text{movie features}), \theta \rangle \]
Recommending things to people

\[ f(\text{user features, movie features}) \rightarrow \text{star rating} \]

Consider the following linear predictor (e.g. from week 1):

\[
\begin{align*}
f(\text{user features, movie features}) &= \\
&= \langle \phi(\text{user features}) ; \phi(\text{movie features}) , \theta \rangle \\
&= \langle \phi(\text{user features}) , \theta_{\text{user}} \rangle + \langle \phi(\text{movie features}) , \theta_{\text{movie}} \rangle
\end{align*}
\]
But this is essentially just two separate predictors!

\[ f(\text{user features}, \text{movie features}) = \]
\[ = \langle \phi(\text{user features}), \theta_{\text{user}} \rangle + \langle \phi(\text{movie features}), \theta_{\text{movie}} \rangle \]

That is, we’re treating user and movie features as though they’re \textbf{independent}!
Recommending things to people

But these predictors should (obviously?) **not** be independent

\[ f(\text{user features}, \text{movie features}) = f(\text{user}) + f(\text{movie}) \]

- do I tend to give high ratings?
- does the population tend to give high ratings to this genre of movie?

But what about a feature like “do I give high ratings to **this genre** of movie”? 

Recommending things to people

**Recommender Systems** go beyond the methods we’ve seen so far by trying to model the **relationships** between people and the items they’re evaluating.
Recommender Systems

1. (next) Collaborative filtering
   (performs recommendation in terms of user/user and item/item similarity)

2. (later) Latent-factor models
   (performs recommendation by projecting users and items into some low-dimensional space)

3. The Netflix Prize

4. Recommender Systems Evaluation
Recommender Systems – more advanced topics

- Incorporating complex *side-information* into recommender systems
- Recommendation in other contexts, e.g. social networks, online dating, etc.
  - Online advertising
- (even later) temporal factors, ethics, text, etc.
Web Mining and Recommender Systems

Similarity-based Recommender Systems
Defining similarity between users & items

Q: How can we measure the similarity between two users?
A: In terms of the items they purchased!

Q: How can we measure the similarity between two items?
A: In terms of the users who purchased them!
Defining similarity between users & items

e.g.: Amazon
Definitions

\[ I_u = \text{set of items purchased by user } u \]
\[ U_i = \text{set of users who purchased item } i \]
Definitions

Or equivalently...

\[ R = \begin{pmatrix} 1 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 1 \end{pmatrix} \]

\( R_u \) = binary representation of items purchased by \( u \)

\( R_{.,i} \) = binary representation of users who purchased \( i \)

\( I_u = \)

\( U_i = \)
0. Euclidean distance

Euclidean distance:
e.g. between two items i,j (similarly defined between two users)

\[ |U_i \setminus U_j| + |U_j \setminus U_i| = \|R_i - R_j\| \]
0. Euclidean distance

Euclidean distance:

e.g.: \( U_1 = \{1,4,8,9,11,23,25,34\} \)
\( U_2 = \{1,4,6,8,9,11,23,25,34,35,38\} \)
\( U_3 = \{4\} \)
\( U_4 = \{5\} \)

\[ |U_1 \setminus U_2| + |U_2 \setminus U_1| = \]
\[ |U_3 \setminus U_4| + |U_4 \setminus U_3| = 2 \]

**Problem:** favors small sets, even if they have few elements in common
1. Jaccard similarity

\[
\text{Jaccard}(A, B) =
\]

\[
\text{Jaccard}(U_i, U_j) =
\]

- Maximum of 1 if the two users purchased \textbf{exactly the same} set of items (or if two items were purchased by the same set of users)

- Minimum of 0 if the two users purchased \textbf{completely disjoint} sets of items (or if the two items were purchased by completely disjoint sets of users)
2. Cosine similarity

- **$U_{\text{harry potter}}$** (vector representation of users who purchased Harry Potter)

- **$U_{\text{pitch black}}$**

- $\theta = 0$ → $A$ and $B$ point in exactly the same direction

- $\cos(\theta) = 1$

- $\theta = 180$ → $A$ and $B$ point in opposite directions (won’t actually happen for 0/1 vectors)

- $\cos(\theta) = -1$

- $\theta = 90$ → $A$ and $B$ are orthogonal

- $\cos(\theta) = 0$
2. Cosine similarity

Why cosine?

- Unlike Jaccard, works for arbitrary vectors
- E.g. what if we have **opinions** in addition to purchases?

\[
R = \begin{pmatrix}
1 & 0 & \cdots & 1 \\
0 & 0 & \cdots & 1 \\
\vdots & \vdots & \ddots & \vdots \\
1 & 0 & \cdots & 1
\end{pmatrix}
\rightarrow
\begin{pmatrix}
-1 & 0 & \cdots & 1 \\
0 & 0 & \cdots & -1 \\
\vdots & \vdots & \ddots & \vdots \\
1 & 0 & \cdots & -1
\end{pmatrix}
\]

bought and **liked**

didn’t buy

bought and **hated**
2. Cosine similarity

E.g. our previous example, now with “thumbs-up/thumbs-down” ratings

\[ \cos(\theta) = 1 \]
(theta = 0) -> Rated by the same users, and they all agree

\[ \cos(\theta) = -1 \]
(theta = 180) -> Rated by the same users, but they completely disagree about it

\[ \cos(\theta) = 0 \]
(theta = 90) -> Rated by different sets of users
4. Pearson correlation

What if we have numerical ratings (rather than just thumbs-up/down)?

$$R = \begin{pmatrix} -1 & 0 & \cdots & 1 \\ 0 & 0 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & -1 \end{pmatrix} \begin{pmatrix} 4 & 0 & \cdots & 2 \\ 0 & 0 & \cdots & 3 \\ \vdots & \vdots & \ddots & \vdots \\ 5 & 0 & \cdots & 1 \end{pmatrix}$$

bought and **liked**
didn’t buy
bought and **hated**
4. Pearson correlation

What if we have numerical ratings (rather than just thumbs-up/down)?
4. Pearson correlation

What if we have numerical ratings (rather than just thumbs-up/down)?

- We wouldn’t want 1-star ratings to be parallel to 5-star ratings
- So we can subtract the average – values are then **negative** for below-average ratings and **positive** for above-average ratings

\[
\text{Sim}(u, v) = \frac{\sum_{i \in I_u \cap I_v} (R_{u,i} - \bar{R}_u)(R_{v,i} - \bar{R}_v)}{\sqrt{\sum_{i \in I_u \cap I_v} (R_{u,i} - \bar{R}_u)^2 \sum_{i \in I_u \cap I_v} (R_{v,i} - \bar{R}_v)^2}}
\]
4. Pearson correlation

Compare to the cosine similarity:

Pearson similarity (between users):

$$\text{Sim}(u, v) = \frac{\sum_{i \in I_u \cap I_v} (R_{u,i} - \bar{R}_u)(R_{v,i} - \bar{R}_v)}{\sqrt{\sum_{i \in I_u \cap I_v} (R_{u,i} - \bar{R}_u)^2 \sum_{i \in I_u \cap I_v} (R_{v,i} - \bar{R}_v)^2}}$$

Cosine similarity (between users):

$$\text{Sim}(u, v) = \frac{\sum_{i \in I_u \cap I_v} R_{u,i}R_{v,i}}{\sqrt{\sum_{i \in I_u \cap I_v} R_{u,i}^2 \sum_{i \in I_u \cap I_v} R_{v,i}^2}}$$

Note: slightly different from previous definition. Here similarity is determined only based on items both users have consumed.
4. Pearson correlation

\[
\text{Sim}(u, v) = \frac{\sum_{i \in I_u \cap I_v} R_{u,i} R_{v,i}}{\sqrt{\sum_{i \in I_u \cap I_v} R_{u,i}^2 \sum_{i \in I_u \cap I_v} R_{v,i}^2}}
\]

Consider all items in the denominator, or just shared items?

**Just shared:** two users should be considered maximally similar if they've rated shared items the same way. If only one user has rated an item, we have no evidence that the other user is different.

**All:** Two users who've rated items the same way *and only rated the same items* should be more similar than two users who've rated some different items.

Ultimately, these are *heuristics*, and either definition could be used depending on the situation.
Given a product: Let $U_i$ be the set of users who viewed it

Rank products according to: $\frac{|U_i \cap U_j|}{|U_i \cup U_j|}$ (or cosine/pearson)

How does Amazon generate their recommendations?

Linden, Smith, & York (2003)
Collaborative filtering in practice

• Amazon uses the cosine similarity
• Similarity is defined between users: the goal is to recommend items that have previously been purchased by similar customers (e.g. "customers who bought items in your shopping cart also bought")
• Main challenges involve scalability: how to cluster users so that we can quickly identify similar users
Note: (surprisingly) that we built something pretty useful out of nothing but interaction data – we didn’t look at any features of the products (or users!) whatsoever
But: we still have a few problems left to address...

1. This is actually kind of slow given a huge enough dataset – if one user purchases one item, this will change the rankings of every other item that was purchased by at least one user in common
2. Of no use for new users and new items (“cold-start” problems)
3. Won’t necessarily encourage diverse results
Web Mining and Recommender Systems

Similarity based recommender – implementation
Code on course webpage

Uses Amazon "Musical Instrument" data from https://s3.amazonaws.com/amazon-reviews-pds/tsv/index.txt
Code: Reading the data

Read the data:

```python
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_1_00.tsv.gz"

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

In [4]:
header = f.readline()
header = header.strip().split('t')
```
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**.
Build data structures representing the set of items for each user and users for each item:
The Jaccard similarity implementation follows the definition directly:

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

```python
In [12]: def jaccard(s1, s2):
    numer = len(s1.intersection(s2))
    denom = len(s1.union(s2))
    return numer / denom
```
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
Now we can implement the recommendation function itself:

```python
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]
```
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': 'B0006VM8HI',
            'product_parent': 'B003251968',
            '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\']
```
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, 'B00006ISSD'),
 (0.01694915254237288, 'B00006ISSB'),
 (0.015065913370998116, 'B000AJR482'),
 (0.01420454545454545, 'B00E7MVP3S'),
 (0.008955223880597015, 'B001255YLI'),
 (0.008849557522123894, 'B003EIRVO8'),
 (0.008333333333333333, 'B0015VEZ22'),
 (0.00821917808219178, 'B00006I5UH'),
 (0.00821390374331552, 'B00008BwM7'),
 (0.007656967840735069, 'B000H2BC4E')]
```
Code: Recommendation

Items that were recommended:

In [17]: itemNames[query]

Out[17]: 'AudioQuest LP record clean brush'

In [18]: [itemNames[x[1]] for x in mostSimilar(query)]

Out[18]: ['Shure SFG-2 Stylus Tracking Force Gauge',
           'Shure M97xe High-Performance Magnetic Phono Cartridge',
           'ART Pro Audio DJPRE II Phono Turntable Preamplifier',
           'Signstek Blue LCD Backlight Digital Long-Playing LP Turntable Stylus Force Scale Gauge Tester',
           'Audio Technica AT120E/T Standard Mount Phono Cartridge',
           'Technics: 45 Adaptor for Technics 1200 (SFWE010)',
           'GruvGlide GRUGGLIDE DJ Package',
           'STANTON MAGNETICS Record Cleaner Kit',
           'Shure M97xe High-Performance Magnetic Phono Cartridge',
           'Behringer PP400 Ultra Compact Phono Preamplifier']
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 \( i \) \( (\text{and store it}) \)
  - Sort all other items by (Jaccard) similarity
  - Return the most similar
Our more efficient implementation works as follows:

```python
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]
```
Which ought to recommend the same set of items, but **much** more quickly:
Web Mining and Recommender Systems

Similarity-based 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?
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:
This can be written as:

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

\[ Z = \sum_{j \in I_u \setminus \{i\}} \text{sim}(i, j) \]
Collaborative filtering for rating prediction

Other rating prediction functions...
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)

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
Our rating prediction code works as follows:

```python
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
```
As an example, select a rating for prediction:

```python
In [20]: dataset[1]
Out[28]: {'marketplace': 'US',
          'customer_id': '14640079',
          'review_id': 'R2SL08ALIYUNU',
          'product_id': 'B003LRN531',
          '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': '2015-06-31'}
```

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

```python
In [30]: predictRating(u, i)
Out[30]: 5.0
```
Similarly, we can evaluate accuracy across the entire corpus:

```python
def MSE(predictions, labels):
    differences = [(x-y)**2 for x, y in zip(predictions, labels)]
    return sum(differences) / len(differences)

alwaysPredictMean = [ratingMean for d in dataset]
cfPredictions = [predictRating(d['customer_id'], d['product_id']) for d in dataset]
labels = [d['star_rating'] for d in dataset]
MSE(alwaysPredictMean, labels)
MSE(cfPredictions, labels)
```

```
Out[35]: 1.4796142779564334
Out[36]: 1.614613004291603
```
Note that this is just a \textit{heuristic} for rating prediction

- In fact in this case it did \textit{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
Better heuristics?
Web Mining and Recommender Systems

Latent-factor models
Recap

1. Measuring similarity between users/items for **binary** prediction
   
   *Jaccard similarity*

2. Measuring similarity between users/items for **real-valued** prediction
   
   *cosine/Pearson similarity*

**Now:** Machine learning-based models for **real-valued** prediction *latent-factor models*
Latent factor models

So far we’ve looked at approaches that try to define some definition of user/user and item/item similarity

**Recommendation** then consists of

- Finding an item $i$ that a user likes (gives a high rating)
- Recommending items that are similar to it (i.e., items $j$ with a similar rating profile to $i$)
Latent factor models

What we’ve seen so far are **unsupervised** approaches and whether the work depends highly on whether we chose a “good” notion of similarity.

So, can we perform recommendations via **supervised** learning?
e.g. if we can model

\[ f(\text{user features, movie features}) \rightarrow \text{star rating} \]

Then recommendation will consist of identifying

\[ \text{recommendation}(u) = \arg \max_{i \in \text{unseen items}} f(u, i) \]
In 2006, Netflix created a dataset of **100,000,000** movie ratings. 
Data looked like:

\[(\text{userID, itemID, time, rating})\]

The goal was to reduce the (R)MSE at predicting ratings:

\[
\text{RMSE}(f) = \sqrt{\frac{1}{N} \sum_{u, i, t \in \text{test set}} (f(u, i, t) - r_{u, i, t})^2}
\]

Whoever first manages to reduce the RMSE by **10%** versus Netflix's solution wins **$1,000,000**.
This led to a lot of research on rating prediction by minimizing the Mean-Squared Error (it also led to a lawsuit against Netflix, once somebody managed to de-anonymize their data)

We’ll look at a few of the main approaches
Let’s start with the simplest possible model:

\[ f(u, i) = \alpha \]
Rating prediction

What about the 2nd simplest model?

\[ f(u, i) = \alpha + \beta_u + \beta_i \]

- \( \alpha \): how much does this user tend to rate things above the mean?
- \( \beta_u \): does this item tend to receive higher ratings than others?
- \( \beta_i \): user
- \( \beta_{pitch\ black} = -0.1 \)
- \( \beta_{julian} = -0.2 \)

\[ \alpha = 4.2 \]
The optimization problem becomes:

$$\arg \min_{\alpha, \beta} \sum_{u,i} (\alpha + \beta_u + \beta_i - R_{u,i})^2 + \lambda \left[ \sum_u \beta_u^2 + \sum_i \beta_i^2 \right]$$

Jointly convex in $\beta_i, \beta_u$. Can be solved by iteratively removing the mean and solving for $\beta$.
Jointly convex?
Differentiate:

$$\arg \min_{\alpha, \beta} \sum_{u,i} (\alpha + \beta_u + \beta_i - R_{u,i})^2 + \lambda \left[ \sum_u \beta_u^2 + \sum_i \beta_i^2 \right]$$
Differentiate:

\[ \frac{\partial \text{obj}}{\partial \beta_u} = \sum_{i \in I_u} 2(\alpha + \beta_u + \beta_i - R_{u,i}) + 2\lambda \beta_u \]

Two ways to solve:

1. "Regular" gradient descent
2. Solve \( \frac{\partial \text{obj}}{\partial \beta_u} = 0 \) (sim. for beta_i, alpha)
Differentiate:

$$\frac{\partial \text{obj}}{\partial \beta_u} = \sum_{i \in I_u} 2(\alpha + \beta_u + \beta_i - R_{u,i}) + 2\lambda \beta_u$$

Solve $$\frac{\partial \text{obj}}{\partial \beta_u} = 0$$:
Iterative procedure – repeat the following updates until convergence:

\[ \alpha = \frac{\sum_{u, i \in \text{train}} (R_{u, i} - (\beta_u + \beta_i))}{N_{\text{train}}} \]

\[ \beta_u = \frac{\sum_{i \in \text{I}_u} R_{u, i} - (\alpha + \beta_i)}{\lambda + |\text{I}_u|} \]

\[ \beta_i = \frac{\sum_{u \in \text{U}_i} R_{u, i} - (\alpha + \beta_u)}{\lambda + |\text{U}_i|} \]

(exercise: write down derivatives and convince yourself of these update equations!)
Looks good (and actually works surprisingly well), but doesn’t solve the basic issue that we started with

$$f(\text{user features}, \text{movie features}) =$$

$$= \langle \phi(\text{user features}), \theta_{\text{user}} \rangle + \langle \phi(\text{movie features}), \theta_{\text{movie}} \rangle$$

That is, we’re still fitting a function that treats users and items independently
Web Mining and Recommender Systems

Latent-factor models (part 2)
Recommending things to people

How about an approach based on **dimensionality reduction**?

My (user’s) “preferences”  HP’s (item) “properties”

I.e., let’s come up with low-dimensional representations of the users and the items so as to best explain the data
Dimensionality reduction

We already have some tools that ought to help us, e.g. from dimensionality reduction:

$$R = \begin{pmatrix}
5 & 3 & \ldots & 1 \\
4 & 2 & 1 \\
3 & 1 & 3 \\
2 & 2 & 4 \\
1 & 5 & 2 \\
\vdots & \vdots & \vdots \\
1 & 2 & \ldots & 1
\end{pmatrix}$$

What is the best low-rank approximation of $R$ in terms of the mean-squared error?
Dimensionality reduction

We can borrow some existing tools, e.g. the singular value decomposition, PCA (etc.):

The “best” rank-K approximation (in terms of the MSE) consists of taking the eigenvectors with the highest eigenvalues.
Dimensionality reduction

**But!** Our matrix of ratings is only partially observed; and it’s **really big**!

$$\begin{pmatrix}
5 & 3 & \cdots & \cdot \\
4 & 2 & 1 & \\
3 & \cdot & 3 & \\
\cdot & 2 & 4 & \\
1 & 5 & \cdot & \\
\cdot & \cdot & \cdot & \\
1 & 2 & \cdots & \cdot
\end{pmatrix}$$

SVD is **not defined** for partially observed matrices, and it is **not practical** for matrices with 1Mx1M+ dimensions.
Latent-factor models

Instead, let’s solve approximately using gradient descent

\[ R = \begin{pmatrix} 5 & 3 & \cdots & . \\ 4 & 2 & 1 \\ 3 & \cdot & 3 \\ \cdot & 2 & 4 \\ 1 & 5 & \cdot \\ \vdots & \vdots & \vdots \\ 1 & 2 & \cdots & . \end{pmatrix} \]

\[ R \approx U V^T \]

K-dimensional representation of each item
K-dimensional representation of each user
Instead, let’s solve approximately using gradient descent

$$R = \begin{pmatrix}
5 & 3 & \cdots & 1 \\
4 & 2 & 1 \\
3 & 3 \\
\cdot & 2 & 4 \\
1 & 5 & \cdot \\
\vdots & \cdot & \cdot \\
1 & 2 & \cdot \\
\end{pmatrix}$$
Latent-factor models

Let’s write this as:

$$f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i$$
Latent-factor models

Let’s write this as:

\[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]

Our optimization problem is then

\[
\arg\min_{\alpha, \beta, \gamma} \sum_{u,i} (\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i})^2 + \lambda \left[ \sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \| \gamma_i \|_2^2 + \sum_u \| \gamma_u \|_2^2 \right]
\]

error regularizer
**Problem:** this is certainly not convex
Oh well. We’ll just solve it approximately.
Again, two ways to solve:

1. "Regular" gradient descent
2. Solve $\frac{\partial \text{obj}}{\partial \gamma_u} = 0$ (sim. For beta_i, alpha, etc.)

(Solution 1 is much easier to implement, though Solution 2 might converge more quickly/easily)
Latent-factor models (Solution 1)

$$\arg\min_{\alpha, \beta, \gamma} \sum_{u,i} (\alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i - R_{u,i})^2 + \lambda \left[ \sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|^2_2 + \sum_u \|\gamma_u\|^2_2 \right]$$
Observation: if we know either the user or the item parameters, the problem becomes "easy"

\[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]

e.g. fix gamma_i – pretend we’re fitting parameters for features
Latent-factor models

(Harder solution): iteratively solve the following subproblems

**objective:**
\[
\arg\min_{\alpha, \beta, \gamma} \sum_u (\alpha + \beta_u + \beta_i + \gamma_i - R_{u,i})^2 + \lambda \left[ \sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|^2 + \sum_u \|\gamma_u\|^2 \right]
\]

\[= \arg\min_{\alpha, \beta, \gamma} \text{objective}(\alpha, \beta, \gamma)\]

1) fix \(\gamma_i\). Solve \(\arg\min_{\alpha, \beta, \gamma_u} \text{objective}(\alpha, \beta, \gamma)\)
2) fix \(\gamma_u\). Solve \(\arg\min_{\alpha, \beta, \gamma_i} \text{objective}(\alpha, \beta, \gamma)\)

3, 4, 5... (repeat until convergence)

Each of these subproblems is “easy” – just regularized least-squares, like we’ve been doing since we studied regression.
This procedure is called **alternating least squares.**
later we'll see how to do this using:

- High-level recommender systems libraries
- Tensorflow (next week?)
Observation: we went from a method which uses only features:

\[ f(\text{user features, movie features}) \rightarrow \text{star rating} \]

to one which completely ignores them:

\[
\arg\min_{\alpha, \beta, \gamma} \sum_{u,i} (\alpha + \beta_u + \beta_i + \gamma_i \cdot R_{u,i})^2 + \lambda \left[ \sum_u \beta_u^2 + \sum_i \beta_i^2 + \sum_i \|\gamma_i\|^2_2 + \sum_u \|\gamma_u\|^2_2 \right]
\]
Should we use features or not?

1) Argument **against** features:

In principle, the addition of features adds **no expressive power** to the model. We **could** have a feature like “is this an action movie?”, but if this feature were useful, the model would “discover” a latent dimension corresponding to action movies, and we wouldn’t need the feature anyway.

**In the limit**, this argument is valid: as we add more ratings per user, and more ratings per item, the latent-factor model should automatically discover any useful dimensions of variation, so the influence of observed features will disappear.
Latent-factor models

Should we use features or not?

2) Argument for features:

But! Sometimes we don’t have many ratings per user/item

Latent-factor models are next-to-useless if either the user or the item was never observed before

\[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]

reverts to zero if we’ve never seen the user before (because of the regularizer)
Latent-factor models

Should we use features or not?

2) Argument for features:

This is known as the cold-start problem in recommender systems. Features are not useful if we have many observations about users/items, but are useful for new users and items.

We also need some way to handle users who are active, but don’t necessarily rate anything, e.g. through implicit feedback.
Dimensionality reduction

Note that this is really a form of **dimensionality reduction**

What are the dimensions that *explain the most variance* in the data?

For connections to other dimensionality reduction techniques (mostly SVD), see textbook
Recently we’ve followed the programme below:

1. Measuring similarity between users/items for **binary** prediction (e.g. Jaccard similarity)
2. Measuring similarity between users/items for **real-valued** prediction (e.g. cosine/Pearson similarity)
3. Dimensionality reduction for **real-valued** prediction (latent-factor models)
4. **Finally** – dimensionality reduction for **binary** prediction
Web Mining and Recommender Systems

Implicit feedback models
One-class recommendation

Suppose we have binary (0/1) observations (e.g. purchases) or pos./neg. feedback (thumbs-up/down)

\[
R = \begin{pmatrix}
1 & 0 & \cdots & 1 \\
0 & 0 & & 1 \\
\vdots & \ddots & \ddots & \vdots \\
1 & 0 & \cdots & 1
\end{pmatrix}
\quad \text{or} \quad
\begin{pmatrix}
-1 & ? & \cdots & 1 \\
? & ? & & -1 \\
\vdots & \ddots & \ddots & \vdots \\
1 & ? & \cdots & -1
\end{pmatrix}
\]

- purchased
- didn’t purchase
- liked
- didn’t evaluate
- didn’t like
How can we use dimensionality reduction (latent factors) to predict binary outcomes?

• Previously we saw regression and logistic regression. These two approaches use the same type of linear function to predict real-valued and binary outputs.
• We can apply an analogous approach to binary recommendation tasks.

This is referred to as “one-class” recommendation.
Why can't we just apply logistic regression?

Why do we need a special approach? Compare to “traditional” approach of replacing “missing values” by 0:
Why can't we just apply logistic regression?
Why can't we just apply logistic regression?
Why can't we just apply logistic regression?

Why do we need a special approach? Compare to “traditional” approach of replacing “missing values” by 0:

• At test time, the model should assign positive scores to items that the user consumed
• But at training time, the model was penalized for not predicting zero!
• (Put differently, the "negative" items are exactly the ones we should be recommending!)
One-class recommendation

Two broad classes of strategy to dealing with one-class data:

1. Instance reweighting: try to figure out which negative (or positive) instances are "important"
2. Optimize *relative* scores rather than positive versus negative
Why can't we just apply logistic regression?

We need a special way to handle "negative" items (since they're not really "negative")

1. Try to figure out which negatives are "real" negatives, and weight instances differently (*instance reweighting*)

2. Try to use a ranking-based objective (*personalized ranking*)
1. Instance reweighting: try to figure out which negative (or positive) instances are "important"

Fit a function of the form:
1. Instance reweighting: try to figure out which negative (or positive) instances are "important"

Fit a function of the form:

$$\arg\min_{\gamma} \sum_{(u,i) \in T} c_{u,i}(p_{u,i} - \gamma_u \cdot \gamma_i)^2 + \lambda \Omega(\gamma)$$
How to choose $c$ (i.e., the importance of each sample)? A couple of heuristics:

1. (Hu et al. 2008): applied to positive instances

$$c_{u,i} = 1 + \alpha r_{u,i} \quad c_{u,i} = 1 + \alpha \log(1 + r_{u,i}/\epsilon)$$
How to choose c (i.e., the importance of each sample)? A couple of heuristics:

2. (Pan et al. 2008): applied to negative instances

\[ c_{u,i} = \alpha \times |I_u| \quad c_{u,i} = \alpha(m - |U_i|) \]

(negative instances should be weighted higher if the user has interacted with many items, etc.)
Instance reweighting
2. Bayesian Personalized Ranking

Idea: Rather than predicting that negative items are disliked, can we just predict that they're less liked than positive items?

\[ u : \]
**Goal:** Estimate a personalized ranking function for each user

- Compare pairs of items $i$ and $j$ together
- $i$ is an item $u$ consumed ("positive")
- $j$ is an item $u$ didn't consume
- Train such that $i$ should have a higher score than $j$ (for $u$)
Bayesian Personalized Ranking

Basic scheme:

• Our original dataset consists of positive \((u,i)\), e.g. purchased items for each user
• Augment this dataset by constructing many triples \((u,i,j)\) where \((u,i)\) is positive and \((u,j)\) is negative
• The model now has to make binary predictions as to whether \(i\) or \(j\) is the positive item
Bayesian Personalized Ranking

**Goal:** Estimate a personalized ranking function for each user

\[ i >_u j \]
Bayesian Personalized Ranking

What form should $x(u,i,j)$ take?
Bayesian Personalized Ranking

Goal is to count how many times we identified \( i \) as being "more preferable" than \( j \) for a user \( u \)

\[ \delta(\hat{x}_{uij} > 0) \]
Bayesian Personalized Ranking

We can think of this as maximizing the probability of correctly predicting pairwise preferences, i.e.,

\[ p(i \text{ is preferred over } j) = \sigma(\gamma_u \cdot \gamma_i - \gamma_u \cdot \gamma_j) \]

- As with logistic regression, we can now maximize the likelihood associated with such a model by gradient ascent
- In practice it isn’t feasible to consider all pairs of positive/negative items, so we proceed by stochastic gradient ascent – i.e., randomly sample a (positive, negative) pair and update the model according to the gradient w.r.t. that pair
Bayesian Personalized Ranking

\[
\max \ln \sigma (\gamma_u \cdot \gamma_i - \gamma_u \cdot \gamma_j)
\]
Recap

1. Measuring similarity between users/items for **binary** prediction
   - Jaccard similarity
2. Measuring similarity between users/items for **real-valued** prediction
   - cosine/Pearson similarity
3. Dimensionality reduction for **real-valued** prediction
   - latent-factor models
4. Dimensionality reduction for **binary** prediction
   - one-class recommender systems
Further reading:

One-class recommendation:
http://goo.gl/08Rh59

Amazon’s solution to collaborative filtering at scale:

An (expensive) textbook about recommender systems:

Cold-start recommendation (e.g.):
http://wanlab.poly.edu/recsys12/recsys/p115.pdf
Web Mining and Recommender Systems

Recommender Systems Libraries
Latent Factor Models (Surprise)

Fairly simple interface that implements the type of (rating prediction) model we've described

Reads data in .tsv format (various others are supported):

```python
reader = Reader(line_format='user item rating', sep='\t')
data = Dataset.load_from_file(dataDir + "goodreads_fantasy.tsv", reader=reader)
```

code from: https://cseweb.ucsd.edu/~jmcauley/pml/code/chap5.html
Create a model instance, train/test splits, and fit the model:

```python
model = SVD()

# Inbuilt functions to split into training and test fractions
trainset, testset = train_test_split(data, test_size=.25)

# Fit the model and extract predictions
model.fit(trainset)
predictions = model.test(testset)
```
Latent Factor Models (Surprise)

Make predictions and compute MSE from the fitted model

```
predictions[0].est

3.6334479463688463

MSE for model predictions (test set)

```sse = 0
for p in predictions:
    sse += (p.r_ui - p.est)**2
print(sse / len(predictions))
```

```
1.1883531641648757
```
Web Mining and Recommender Systems

Bayesian Personalized Ranking (Implicit)
A little more work to put the data in the right format. Start by reading the data in our usual formats:

```r
data = list(parseData(dataDir + "goodreads_reviews_fantasy_paranormal.json.gz"))
random.shuffle(data)
```

Example from the dataset:

```r
data[0]
```

```r
{
'book_id': '13451182',
'date_added': 'Sun Sep 09 18:58:45 -0700 2012',
'date_updated': 'Sun Oct 07 15:13:32 -0700 2012',
'n_comments': 1,
'n_votes': 0,
'rating': 1,
}
Bayesian Personalized Ranking (Implicit)

Build some utility data structures:

```python
userIDs, itemIDs = {},{}

for d in data:
    u, i = d['user_id'], d['book_id']
    if not u in userIDs: userIDs[u] = len(userIDs)
    if not i in itemIDs: itemIDs[i] = len(itemIDs)

nUsers, nItems = len(userIDs), len(itemIDs)

nUsers, nItems

(256088, 258212)
```
Bayesian Personalized Ranking (Implicit)

Build some sparse matrix data structures. Here we essentially build the (massive!) user-item interaction matrix describing which items users have interacted with:

```python
Xiu = scipy.sparse.lil_matrix((nItems, nUsers))
for d in data:
    Xiu[itemIDs[d['book_id']], userID[d['user_id']]] = 1

Xui = scipy.sparse.csr_matrix(Xiu.T)
```

Bayesian Personalized Ranking model with 5 latent factors

```python
model = bpr.BayesianPersonalizedRanking(factors = 5)
```
Bayesian Personalized Ranking (Implicit)

Fit the model, and get some recommendations from it:

```python
model.fit(Xiu)
```

Get recommendations for a particular user (the first one) and to get items related to (similar latent factors) to a particular item

```python
recommended = model.recommend(0, Xui)
related = model.similar_items(0)
```

```python
related
```

```
[(0, 1.0),
 (42098, 0.9885355),
 (142964, 0.9845209),
 (150861, 0.98274595),
 (231639, 0.9826295),
 (182330, 0.9813926),
 (280959, 0.9800054)]
```
Bayesian Personalized Ranking (Implicit)

Can also extract latent factors (e.g. for visualization):

```python
itemFactors = model.item_factors
userFactors = model.user_factors

itemFactors[0]

array([-0.74582803, -0.10878776, 0.32922822, 0.16516064, 0.38874012,
       0.7460656 ], dtype=float32)
```
Web Mining and Recommender Systems

Recommender Systems in Tensorflow
Recommender Systems in Tensorflow

(will come back to Tensorflow later, but code is in: https://cseweb.ucsd.edu/~jmcauley/pml/code/chap5.html)
Web Mining and Recommender Systems

More on recommender systems evaluation
Challenges in evaluating recommender systems

So far, we've mostly considered the Mean Squared Error when evaluating recommender systems; we haven't thought too hard about this since introducing linear regression.

What might be some problems with this choice?
Challenges in evaluating recommender systems

What might be some problems with the MSE? Consider e.g.

Label: 🌟🌟🌟🌟🌟 vs. 🌟🌟🌟🌟
Prediction: 🌟🌟🌟🌟 vs. 🌟

Which has a higher penalty? Which *should* have?
What might be some problems with the MSE? Consider e.g.

Label:

Model 1: 

VS.

Model 2:

Which has a higher penalty? Which *should* have?
Challenges in evaluating recommender systems

What might be some problems with the MSE? Consider e.g.

MSE assumed errors were normally distributed; what if they're more bimodal?

What *should* the correct prediction be in this case?
Challenges in evaluating recommender systems

More thoughts:

• The most popular items (or most active users) will dominate our MSE calculation; will less popular items (or users) receive "fair" consideration?

• A small change in the MSE can drastically change the ordering of the most relevant items; alternately a better MSE does not necessarily mean a better recommender
Just as we saw (e.g.) precision and recall when evaluating classifiers, we can consider ranking-based metrics for evaluation of recommender systems. A few we'll look at:

- Precision and Recall @ K (again)
- AUC (Area Under ROC Curve)
  - Mean Reciprocal Rank
- Cumulative Gain and NDCG (in textbook)
  - Beyond accuracy
Precision and Recall @ K

Much as we considered Precision and Recall (@K) when evaluating classifiers, they can also be used to evaluate ranked recommendation lists.

First, rank recommended items for each user by relevance:

\[ rank_u(i) < rank_u(j) \iff f(u, i) > f(u, j) \]

\[ rank_u(i) = rank_u(j) \iff i = j. \]
Next, count how many of the (withheld/test) interactions for a user are among the top K recommendations:
Next, count how many of the (withheld/test) interactions for a user are among the top K recommendations:

\[
\text{precision@} K(u) = \frac{|\{i \in I_u \mid \text{rank}_u(i) \leq K\}|}{K}
\]

Can then be defined for all users (likewise for recall@K):

\[
\text{precision@} K = \frac{1}{|U|} \sum_{u \in U} \text{precision@} K(u)
\]

\[
\text{recall@} K = \frac{1}{|U|} \sum_{u \in U} \frac{|\{i \in I_u \mid \text{rank}_u(i) \leq K\}|}{|I_u|}
\]
Mean Reciprocal Rank

How high is the rank of the relevant item?
  • An ideal algorithm should rank it first
  • An algorithm that ranks it 10th is somewhat worse
  • An algorithm that ranks it 100th is much worse
  • The further down the ranking we go the less difference it makes

(assuming only a single withheld "test" item $i_u$ for each user)
Mean Reciprocal Rank

1.0 = ideal algorithm; withheld item always ranked first
1/n = relevant item tends to be ranked in the n\(^{th}\) position
Does a ranker tend to give *positive* (e.g. purchased) items higher ranks than *negative* (e.g. not-purchased) items:
AUC
The AUC:
- Counts the fraction of times the algorithm gives a higher score to a positive than to a negative interaction
  - (1.0 = always correct; 0.5 = random)
  - Across all users:

\[
AUC = \frac{1}{|U|} \text{AUC}(u)
\]
Why the AUC?
Why the AUC?

- Doesn't force negative items to be rated as "negative" – just less positive than positive – this is desirable in implicit feedback settings
- Rewards the algorithm for ordering things correctly, which may be more important to users than (e.g.) predicting ratings correctly
- Is easy (compared to some other metrics) to optimize
Web Mining and Recommender Systems

"User-free" models of recommendation
So far we've studied (arguably) the simplest approaches for a variety of tasks:

1. Similarity-based models for binary data
2. Similarity-based models for real-valued data
3. Dimensionality reduction (latent factors) for real-valued data
4. Dimensionality reduction for binary data

Next we'll discuss a few alternate approaches to similar problems
Main goal in this section is to avoid having an explicit model of a user $\gamma_u$ Why?
Main goal in this section is to *avoid having an explicit model of a user gamma_u*

Why?

- Previous models had K parameters per user – very expensive in settings with many users!
- Poor performance for users with few interactions
- Requires continuous retraining as users continue to interact
User-free recommenders

Can we design algorithms that take a set of items as input, and generate recommended items as output?

As users provide more interactions, we just change the input to the algorithm – no need to retrain!

1. Sparse Linear Methods (SLIM)
2. Factored Item Similarity Models (FISM)
Adapts ideas from regression; model interactions as

\[ f(u, i) = R_u \cdot W_i \]

- Vector of interactions for user \( u \)
- (linear!) parameter vector
1. Sparse Linear Methods (SLIM) (Ning and Karypis, 2011)

Can be expanded as:

Challenge: $W$ is a (dense!) $|I| \times |I|$ matrix
1. Sparse Linear Methods (SLIM) (Ning and Karypis, 2011)

Solution: assume $W$ is \textit{sparse}, which is achieved through a regularizer:

$$\arg\min_W \|R - RW^T\|_2^2 + \lambda \Omega_2(W) + \lambda' \Omega_1(W)$$

s.t. $W_{i,j} \geq 0; \quad W_{i,i} = 0$. 
Sparse linear methods:
• Rapid inference time (compared to traditional methods, not compared to latent factor models)
• Better long-tail performance, i.e., works well for rarely-occurring items
2. Factored Item Similarity Models (FISM) (Kabbur et al. 2013)

**Idea:** a user is just the average over items they consume

Replace the user representation with an average of item representations

\[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]
2. Factored Item Similarity Models (FISM) (Kabbur et al. 2013)

Replace

\[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]

with
2. Factored Item Similarity Models (FISM) (Kabbur et al. 2013)

Replace

\[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]

with

\[ f(u, i) = \alpha + \beta_u + \beta_i + \frac{1}{|I_u \setminus \{i\}|} \sum_{j \in I_u \setminus \{i\}} \gamma'_j \cdot \gamma_i \]
2. Factored Item Similarity Models (FISM) (Kabbur et al. 2013)

Note that we have *two* item representations (instead of an item and a user representation)

\[ \gamma'_j \text{ and } \gamma_i \]

These are essentially a "query" and a "target" representation
Further reading:

• Sparse Linear Methods (SLIM) (Ning and Karypis, 2011)
• Factored Item Similarity Models (FISM) (Kabbur et al. 2013)
Won't spend a tonne of time teaching deep learning, but obviously it's made some headway into recommendation (just like everywhere else...)  

Here will just give a basic sense of some of the main ideas 

See textbook for details!
Why not deep learning for recommendation?

Why should we need deep learning to improve recommender systems?

\[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]

Deep Learning is normally used to uncover non-linear relationships/transforms among features, but this model doesn't have any features!

If latent factors can uncover any properties about users/items, what can a "deep" model learn?
Maximum inner product

$\gamma_u[1]$ (e.g. comedy)

$\gamma_u[0]$ (e.g. action)

Nearest neighbors

$\gamma_u[1]$ (e.g. comedy)

$\gamma_u[0]$ (e.g. action)
Why *not* deep learning for recommendation?

**Idea:** there's nothing sacred about the inner product in this function, and other choices might be better in some contexts

$$f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i$$

Maybe we can automatically learn what types of relationship would be most effective
1. Neural Collaborative Filtering (He et al. 2017)

**Idea:** use a multilayer perceptron to learn the ideal relationship between gamma_u and gamma_i
1. Neural Collaborative Filtering (He et al. 2017)

**Idea:** use a multilayer perceptron to learn the ideal relationship between gamma_u and gamma_i

- Arguably, this will help us to learn more complex interactions between users and items
- Some counterargument (e.g. Dacrema et al. 2019): it's actually hard for an MLP to learn an inner product function!
2. AutoRec (Sedhain et al. 2015)

Idea: *Autoencoders* are a technique to learn low-dimensional latent representations of feature vectors; can they be used for recommendation?
Autoencoder-based recommendation:

- Input is a vector of items a user has consumed (or a set of users who have consumed an item)
- Model is trained to encode these vectors
- At test time, find un-consumed items that have the highest score according to the decoder
- **Note:** this is a user-free model! The input/output is just a vector of items!
3. Convolutional and Recurrent Models

Plenty of other approaches based on Convolutional and Recurrent Neural Networks:

- CNNs often used as a way to incorporate rich content into recommender systems (e.g. images)
- RNNs (and Transformers, etc.) are often used as a way to incorporate sequential dynamics

We'll come back to these a little (but not much) later
Are deep-learning models "worth it"?

Some doubts as to whether deep learning-based recommenders are really "worth it":

• Some evidence that simpler models will work just as well if carefully tuned (Dacrema et al. 2019)
• Potentially adding a lot of parameters / training complexity for modest performance improvements
  • Also challenges re. efficient retrieval etc.
Further reading:

- He et al. (2017): Neural Collaborative Filtering
- He and Chua (2017): Neural Factorization Machines
- Cheng et al. (2016): Wide and Deep Learning for Recommendation
- Guo et al. (2017): Deep Factorization Machines
- Sedhain et al. (2015): AutoRec
Web Mining and Recommender Systems

What's still coming up?
Extensions of latent-factor models

So far we have a model that looks like:

\[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]

How might we extend this to:

- Incorporate features about users and items
- Handle implicit feedback
  - Change over time

See Yehuda Koren (+Bell & Volinsky)'s magazine article:
“Matrix Factorization Techniques for Recommender Systems”
IEEE Computer, 2009
1) Features about users and/or items

(simplest case) Suppose we have binary attributes to describe users or items

\[ A(u) = [1,0,1,1,0,0,0,0,0,1,0,1] \]

attribute vector for user \( u \)

- e.g. is female
- is male
- is between 18-24yo
1) Features about users and/or items

(simplest case) Suppose we have **binary attributes** to describe users or items

- Associate a **parameter vector** with each attribute
- Each vector encodes how much a particular feature “offsets” the given latent dimensions

\[ A(u) = [1,0,1,1,0,0,0,0,0,1,0,1] \]

attribute vector for user \( u \)

e.g. \( y_0 = [-0.2,0.3,0.1,-0.4,0.8] \)
~ “how does being male impact \( \gamma_u \)”
Extensions of latent-factor models

1) Features about users and/or items

(simplest case) Suppose we have **binary attributes** to describe users or items

- Associate a **parameter vector** with each attribute
- Each vector encodes how much a particular feature "offsets" the given latent dimensions
  - Model looks like:

\[ f(u, i) = \alpha + \beta_u + \beta_i + (\gamma_u + \sum_{a \in A(u)} \rho_a) \cdot \gamma_i \]

- Fit as usual:

\[
\arg\min_{\alpha, \beta, \gamma, \rho} \sum_{u, i \in \text{train}} (f(u, i) - r_{u,i})^2 + \lambda \Omega(\beta, \gamma)
\]

\[ \text{error} \quad \text{regularizer} \]
2) Implicit feedback

Perhaps many users will never actually rate things, but may still interact with the system, e.g. through the movies they view, or the products they purchase (but never rate)

• Adopt a similar approach – introduce a binary vector describing a user’s actions

\[ N(u) = [1,0,0,0,1,0,\ldots,0,1] \]

\[ e.g. y_0 = [-0.1,0.2,0.3,-0.1,0.5] \]

Clicked on “Love Actually” but didn’t watch
Extensions of latent-factor models

2) Implicit feedback

Perhaps many users will never actually rate things, but may still interact with the system, e.g. through the movies they view, or the products they purchase (but never rate)

• Adopt a similar approach – introduce a binary vector describing a user’s actions
  • Model looks like:

\[ f(u, i) = \alpha + \beta_u + \beta_i + (\gamma_u + \frac{1}{\|N(u)\|} \sum_{a \in N(u)} \rho_a) \cdot \gamma_i \]

normalize by the number of actions the user performed
3) Change over time

There are a number of reasons why rating data might be subject to temporal effects...
Extensions of latent-factor models

3) Change over time

Figure from Koren: “Collaborative Filtering with Temporal Dynamics” (KDD 2009)
Extensions of latent-factor models

3) Change over time

Figure from Koren: “Collaborative Filtering with Temporal Dynamics” (KDD 2009)

People tend to give higher ratings to older movies
Extensions of latent-factor models

3) Change over time

A few temporal effects from beer reviews
Extensions of latent-factor models

3) Change over time

There are a number of reasons why rating data might be subject to temporal effects...

- Changes in the interface
  People give higher ratings to older movies (or, people who watch older movies are a biased sample)
- The community’s preferences gradually change over time
- My girlfriend starts using my Netflix account one day
- I binge watch all 144 episodes of Buffy one week and then revert to my normal behavior
- I become a “connoisseur” of a certain type of movie
- Anchoring, public perception, seasonal effects, etc.

- e.g. “Collaborative filtering with temporal dynamics” Koren, 2009
- e.g. “Sequential & temporal dynamics of online opinion” Godes & Silva, 2012
- e.g. “Temporal recommendation on graphs via long- and short-term preference fusion” Xiang et al., 2010
- e.g. “Modeling the evolution of user expertise through online reviews” McAuley & Leskovec, 2013
Extensions of latent-factor models

3) Change over time

Each definition of temporal evolution demands a slightly different model assumption (we’ll see some in more detail later tonight!) but the basic idea is the following:

1) Start with our original model:
   \[ f(u, i) = \alpha + \beta_u + \beta_i + \gamma_u \cdot \gamma_i \]

2) And define some of the parameters as a function of time:
   \[ f(u, i, t) = \alpha + \beta_u(t) + \beta_i(t) + \gamma_u(t) \cdot \gamma_i \]

3) Add a regularizer to constrain the time-varying terms:
   \[ \arg\min_{\alpha, \beta, \gamma} \sum_{u,i,t \in \text{train}} (f(u, i, t) - r_{u,i,t})^2 + \lambda_1 \Omega(\beta, \gamma) + \lambda_2 \| \gamma(t) - \gamma(t + \delta) \| \]

parameters should change smoothly
Moral(s) of the story

How much do these extension help?

Moral: increasing complexity helps a bit, but changing the model can help a lot

Figure from Koren: “Collaborative Filtering with Temporal Dynamics” (KDD 2009)
So what actually happened with Netflix?

- The AT&T team “BellKor”, consisting of Yehuda Koren, Robert Bell, and Chris Volinsky were early leaders. Their main insight was how to effectively incorporate temporal dynamics into recommendation on Netflix.
- Before long, it was clear that no one team would build the winning solution, and Frankenstein efforts started to merge. Two frontrunners emerged, “BellKor’s Pragmatic Chaos”, and “The Ensemble”.
- The BellKor team was the first to achieve a 10% improvement in RMSE, putting the competition in “last call” mode. The winner would be decided after 30 days.
- After 30 days, performance was evaluated on the hidden part of the test set.
- Both of the frontrunning teams had the same RMSE (up to some precision) but BellKor’s team submitted their solution 20 minutes earlier and won $1,000,000

For a less rough summary, see the Wikipedia page about the Netflix prize, and the nytimes article about the competition: http://goo.gl/WNpy7o
Moral(s) of the story

Afterword

• Netflix had a class-action lawsuit filed against them after somebody de-anonymized the competition data
• $1,000,000 seems to be **incredibly cheap** for a company the size of Netflix in terms of the amount of research that was devoted to the task, and the potential benefit to Netflix of having their recommendation algorithm improved by 10%
• Other similar competitions have emerged, such as the Heritage Health Prize ($3,000,000 to predict the length of future hospital visits)

• But... the winning solution never made it into production at Netflix – it’s a monolithic algorithm that is very expensive to update as new data comes in*

*source: a friend of mine told me and I have no actual evidence of this claim
Moral(s) of the story

Finally...

Q: Is the RMSE really the right approach? Will improving rating prediction by 10% actually improve the user experience by a significant amount?
A: Not clear. Even a solution that only changes the RMSE slightly could drastically change which items are top-ranked and ultimately suggested to the user.

Q: But... are the following recommendations actually any good?
A1: Yes, these are my favorite movies!
or A2: No! There’s no diversity, so how will I discover new content?
Various extensions of latent factor models:

- Incorporating features
  * e.g. for cold-start recommendation
- Implicit feedback
  * e.g. when ratings aren’t available, but other actions are
- Incorporating temporal information into latent factor models
  * seasonal effects, short-term “bursts”, long-term trends, etc.
  - Missing-not-at-random
  * incorporating priors about items that were not bought or rated
  - The Netflix prize
Further reading:
Yehuda Koren’s, Robert Bell, and Chris Volinsky’s IEEE computer article:

Paper about the “Missing-at-Random” assumption, and how to address it:

Collaborative filtering with temporal dynamics:

Recommender systems and sales diversity: