CSE 258 – Lecture 7
Web Mining and Recommender Systems
Announcements

• Assignment 1 is out
• It will be due in week 8 on Monday at 5pm
• HW3 will help you set up an initial solution
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**
Why recommendation?

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
Recommending things to people

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

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

Movie features: genre, actors, rating, length, etc.

User features: age, gender, location, etc.

<table>
<thead>
<tr>
<th>Product Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Genres</strong></td>
</tr>
<tr>
<td><strong>Director</strong></td>
</tr>
<tr>
<td><strong>Starring</strong></td>
</tr>
<tr>
<td><strong>Supporting actors</strong></td>
</tr>
<tr>
<td><strong>Studio</strong></td>
</tr>
<tr>
<td><strong>MPAA rating</strong></td>
</tr>
<tr>
<td><strong>Captions and subtitles</strong></td>
</tr>
<tr>
<td><strong>Rental rights</strong></td>
</tr>
<tr>
<td><strong>Purchase rights</strong></td>
</tr>
<tr>
<td><strong>Format</strong></td>
</tr>
</tbody>
</table>

A. Phillips
Reviewer ranking: #17,230,554

90% helpful votes received on reviews (151 of 167)

ABOUT ME
Enjoy the reviews...

ACTIVITIES
Reviews (16)
Public Wish List (2)
Listmania Lists (2)
Tagged Items (1)
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

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

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

$$f(\text{user features, movie features}) = \langle \phi(\text{user features}); \phi(\text{movie features}), \theta \rangle$$

$$= \langle \phi(\text{user}), \theta^{\text{user}} \rangle + \langle \phi(\text{movie}), \theta^{\text{movie}} \rangle$$
Recommending things to people

But this is essentially just two separate predictors!

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

user predictor

movie predictor

That is, we’re treating user and movie features as though they’re independent!
But these predictors should (obviously?) **not** be independent

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

- 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.

- Preference toward “action”
- Preference toward “special effects”
- HP’s (item) “properties”
- My (user’s) “preferences”

 Compatibility

- Is the movie action-heavy?
- Are the special effects good?
Today

**Recommender Systems**

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

2. Assignment 1

3. (next lecture) Latent-factor models
   (performs recommendation by projecting users and items into some low-dimensional space)

4. (next lecture) The Netflix Prize
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...

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

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

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

\[ I_u = \{ i \mid R_{ui} = 1 \} \quad U_i = \{ u \mid R_{ui} = 1 \} \]
Euclidean distance:
e.g. between two items $i, j$ (similarly defined between two users)

$$|U_i \setminus U_j| + |U_j \setminus U_i^*| = \| R_i - R_j \|$$
Euclidean distance:

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

\[ |U_1 \setminus U_2| + |U_2 \setminus U_1| = 3 \]
\[ |U_3 \setminus U_4| + |U_3 \setminus U_4| = 2 \]

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

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

\[ \text{Jaccard}(U_i, U_j) = \frac{|u_i \cap u_j|}{|u_i \cup u_j|} \]

→ Maximum of 1 if the two users purchased **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 **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)

\[ \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|} \]

\[ \cos(\theta) = \begin{cases} 1 & \text{if } \theta = 0 \Rightarrow A \text{ and } B \text{ point in exactly the same direction} \\ -1 & \text{if } \theta = 180 \Rightarrow A \text{ and } B \text{ point in opposite directions (won't actually happen for 0/1 vectors)} \\ 0 & \text{if } \theta = 90 \Rightarrow A \text{ and } B \text{ are orthogonal} \end{cases} \]
2. Cosine similarity

Why cosine?

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

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

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

\[ \cos(\theta) = 1 \]

(\( \theta = 0 \) \( \rightarrow \) Rated by the same users, and they all agree)

\[ \cos(\theta) = -1 \]

(\( \theta = 180 \) \( \rightarrow \) Rated by the same users, but they completely disagree about it)

\[ \cos(\theta) = 0 \]

(\( \theta = 90 \) \( \rightarrow \) Rated by different sets of users)
What if we have numerical ratings (rather than just thumbs-up/down)?

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

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

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

$$\text{Cosine}(A, B) = \frac{A \cdot B}{\|A\| \|B\|}$$

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.
Collaborative filtering in practice

How does Amazon generate their recommendations?

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)

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

Can also use similarity functions to estimate ratings:

\[ r(u, i) = \frac{1}{\sum_{j \in I_u} \text{Sim}(i, j)} \sum_{j \in I_u} r_{uj} \text{Sim}(i, j) \times \sum_{j \in I_u} \text{Sim}(i, j) \]
Collaborative filtering in practice

**Note:** (surprisingly) that we built something pretty useful out of **nothing but rating data** – we didn’t look at any features of the products 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
Questions
CSE 258 – Lecture 7
Web Mining and Recommender Systems

Similarity based recommender - implementation
Code on:
http://jmcauley.ucsd.edu/code/week4.py

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

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

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

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']
```
The Jaccard similarity implementation follows the definition directly:

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

In [12]:
```python
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
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:

```python
In [14]: dataset[2]
Out[14]: {'marketplace': 'US',
          'customer_id': '6111003',
          'review_id': 'RIZR67JUKUBI0',
          'product_id': 'B0006VMBHI',
          'product_parent': '6032619690',
          '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'}
```

```python
In [15]: query = dataset[2][‘product_id’]
```
Next, let’s use the code to make a recommendation. The query is just a product ID:
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 M97x E High-Performance Magnetic Phono Cartridge',
              'ART Pro Audio DJPKE 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 GRUVGLIDE DJ Package',
              'STANTON MAGNETICS Record Cleaner Kit',
              'Shure M97x E 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.
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
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:

In [20]: `mostSimilarFast(query)`

Out[20]: 
```
[(0.028446389496717725, 'B00006I55D'),
 (0.01694915254237288, 'B00006I5SB'),
 (0.015065913370998116, 'B000AJR482'),
 (0.01420454545454544, 'B00E7MVP3S'),
 (0.008955223880597015, 'B001255YL2'),
 (0.008849557522123894, 'B003EIRVO8'),
 (0.00833333333333333, 'B0015VEZ22'),
 (0.00821917808219178, 'B00006I5UH'),
 (0.008021390374331552, 'B00008BWM7'),
 (0.007656967840735069, 'B000H2BC4E')]```
Similarity based recommender 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?
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 \(j\)
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 $i$

$$Z = \sum_{j \in I_u \setminus \{i\}} \text{sim}(i, j)$$
Now we can adapt our previous recommendation code to predict ratings.

```python
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
```

\[ r(u, i) = \frac{1}{Z} \sum_{j \in I_u \setminus \{i\}} r_{u,j} \cdot \text{sim}(i, j) \]
As an example, select a rating for prediction:

```python
In [28]: dataset[1]
Out[28]: {'marketplace': 'US',
         'customer_id': '14640079',
         'review_id': 'RZSL0B8ALIYUNU',
         '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
```
Similarly, we can evaluate accuracy across the entire corpus:

```python
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
```
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
Questions?