ENGR 212: Programming Practice

Week 6

Recommending Items

Critic	Similarity	Night	S.xNight	Lady	S.xLady	Luck	S.xLuck
Rose	0.99	3.0	2.97	2.5	2.48	3.0	2.97
Seymour	0.38	3.0	1.14	3.0	1.14	1.5	0.57
Puig	0.89	4.5	4.02			3.0	2.68
LaSalle	0.92	3.0	2.77	3.0	2.77	2.0	1.85
Matthews	0.66	3.0	1.99	3.0	1.99		
Total			12.89		8.38		8.07
Sim. Sum			3.84		2.95		3.18
Total/Sim. Sum			3.35		2.83		2.53

```python
def getRecommendations(prefs,person,similarity=sim_pearson):
  totals={}
  simSums={}
  for other in prefs:
    # don't compare me to myself
    if other==person: continue
    sim=similarity(prefs,person,other)

    # ignore scores of zero or lower
    if sim<=0: continue

    for item in prefs[other]:
      # only score movies I haven't seen yet
      if item not in prefs[person]:
        # Similarity * Score
        totals.setdefault(item,0)
        totals[item]+=prefs[other][item]*sim
        # Sum of similarities
        simSums.setdefault(item,0)
        simSums[item]+=sim

  # Create the normalized list
  rankings=[(total/simSums[item],item) for item,total in totals.items()]

  # Return the sorted list
  rankings.sort()
  rankings.reverse()
  return rankings
```

Week 5/code

- Play around with getRecommendations function.

```
from recommendations import *
print getRecommendations(critics, ‘Ali’)
```

Matching Products

- Now you know how to find similar people and recommend products.
- What if you want to see which products are similar to each other?
- How would you do it?

Matching Products

- Determine similarity by
  - looking at **who** liked a particular **item** and

    looking at **what items/movies** that **they** liked
  - seeing the other **things** **they** liked.

    seeing the other **people** who liked the same **things**

Matching Products

- Just need to swap the people (i.e., critics) and the items (i.e., movies).

```
    {'Lisa Rose': {'Lady in the Water': 2.5, 'Snakes on a Plane': 3.5},
    'Gene Seymour': {'Lady in the Water': 3.0, 'Snakes on a Plane': 3.5}}

to:

    {'Lady in the Water':{'Lisa Rose':2.5,'Gene Seymour':3.0},
    'Snakes on a Plane':{'Lisa Rose':3.5,'Gene Seymour':3.5}} etc..
```

Matching Products

- The following function performs the necessary transformation:

```
def transformPrefs(prefs):
  result={ }
  for person in prefs:
    for item in prefs[person]:
      result.setdefault(item,{})
      # Flip item and person
      result[item][person]= prefs[person][item]
  return result
```

Matching Products

```
from recommendations import *

movies = transformPrefs(critics)

print topMatches(movies, 'Superman Returns')
```

Matching Products

- An online retailer might collect purchase histories for the purpose of recommending products to individuals.
- Switching the products and people would allow to search for people who might buy certain products. *- planning a marketing effort for a big clearance day.*
- New links on a link-recommendation site are seen by the people who are most likely to enjoy them.

Recommendation - Scalability Considerations

- Our recommendation engine requires the use of all the rankings from every user in order to create a dataset.
- OK for a few thousand people or items
  - how about for a very large site like Amazon, which has millions of customers and products?

Recommendation - Scalability Considerations

- Comparing a user with every other user and then comparing every product each user has rated can be very slow.
  RUNNING TIME COMPLEXITY
  (MAY BE QUADRATIC!)
- Also, a site that sells millions of products may have very little overlap between people, which can make it difficult to decide which people are similar.
  SPARSITY

Item-Based Filtering

- So far, we have seen **user-based** collaborative filtering.
- An alternative is known as **item-based** collaborative filtering.
- In cases with very large datasets, it allows many of the calculations to be performed in advance so that a user needing recommendations can get them more quickly.

Item-Based Filtering

- The technique:
  - precompute the most similar items for each item.
- To make recommendations to a user:
  - look at his top-rated items and
  - create a weighted list of the items most similar to those.

Item-Based Filtering

- The first step still requires you to examine all the data. How is this approach more efficient?
  - comparisons between items will not change as often as comparisons between users.
- So what?
  - you do not have to continuously calculate each item’s most similar items.
  - do it once, and reuse multiple times.
  - Do it at low-traffic times or <u>on a computer separate from your main application.</u>

Building the Item Similarity Dataset

```
def calculateSimilarItems(prefs, n=10):
   # Create a dictionary of items showing
   # which other items they are most similar to.
   result={}

   # Invert the preference matrix to be item-centric
   itemPrefs=transformPrefs(prefs)

   for item in itemPrefs:
       # Find the most similar items to this one
       scores=topMatches(itemPrefs,item,n,sim_distance)
       result[item]=scores

   return result
```

Building the Item Similarity Dataset

- This function first inverts the score dictionary using the transformPrefs function defined earlier, giving a list of items along with how they were rated by each user.
- It then loops over every item and passes the transformed dictionary to the topMatches function to get the most similar items along with their similarity scores.
- Finally, it creates and returns a dictionary of items along with a list of their most similar items.

Building the Item Similarity Dataset

```
>>> from recommendations import *
>>> itemsim = calculateSimilarItems(critics)
```

- This function only has to be run frequently enough to keep the item similarities up to date.
- Run it more often early on when the user base and number of ratings is small
  - *as the user base grows, the similarity scores between items will usually become more stable.*

Getting recommendations

- Algorithm
  - Get all the items that the user has ranked.
  - Find the similar items.
  - Weigh them according to how similar they are.
- Unlike our previous approach, the critics are not involved at all
  - Instead, there is a grid of
    - movies I’ve watched and rated vs. movies I haven’t watched.

Getting recommendations

Table 2-3. Item-based recommendations for Toby

Movie	Rating	Night	R.xNight	Lady	R.xLady	Luck	R.xLuck
Snakes	4.5	0.182	0.818	0.222	0.999	0.105	0.474
Superman	4.0	0.103	0.412	0.091	0.363	0.065	0.258
Dupree	1.0	0.148	0.148	0.4	0.4	0.182	0.182
Total		0.433	1.378	0.713	1.764	0.352	0.914
Normalized			3.183		2.598		2.473

Getting recommendations

Table 2-3. Item-based recommendations for Toby

Movie	Rating	Night	R.xNight	Lady	R.xLady	Luck	R.xLuck
Snakes	4.5	0.182	0.818	0.222	0.999	0.105	0.474
Superman	4.0	0.103	0.412	0.091	0.363	0.065	0.258
Dupree	1.0	0.148	0.148	0.4	0.4	0.182	0.182
Total		0.433	1.378	0.713	1.764	0.352	0.914
Normalized			3.183		2.598		2.473

Getting recommendations

Table 2-3. Item-based recommendations for Toby

Movie	Rating	Night	R.xNight	Lady	R.xLady	Luck	R.xLuck
Snakes	4.5	0.182	0.818	0.222	0.999	0.105	0.474
Superman	4.0	0.103	0.412	0.091	0.363	0.065	0.258
Dupree	1.0	0.148	0.148	0.4	0.4	0.182	0.182
Total		0.433	1.378	0.713	1.764	0.352	0.914
Normalized			3.183		2.598		2.473

Getting recommendations

Table 2-3. Item-based recommendations for Toby

Movie	Rating	Night	R.xNight	Lady	R.xLady	Luck	R.xLuck
Snakes	4.5	0.182	0.818	0.222	0.999	0.105	0.474
Superman	4.0	0.103	0.412	0.091	0.363	0.065	0.258
Dupree	1.0	0.148	0.148	0.4	0.4	0.182	0.182
Total		0.433	1.378	0.713	1.764	0.352	0.914
Normalized			3.183		2.598		2.473

Getting recommendations

Table 2-3. Item-based recommendations for Toby

Movie	Rating	Night	R.xNight	Lady	R.xLady	Luck	R.xLuck
Snakes	4.5	0.182	0.818	0.222	0.999	0.105	0.474
Superman	4.0	0.103	0.412	0.091	0.363	0.065	0.258
Dupree	1.0	0.148	0.148	0.4	0.4	0.182	0.182
Total		0.433	1.378	0.713	1.764	0.352	0.914
Normalized			3.183		2.598		2.473

Getting recommendations

```
def getRecommendedItems(prefs,itemSim,user):
  userRatings=prefs[user]
  scores={}
  totalSim={}
  # Loop over items rated by this user
  for (item,rating) in userRatings.items( ):

    # Loop over items similar to this one
    for (similarity,item2) in itemSim[item]:

      # Ignore if this user has already rated this item
      if item2 in userRatings: continue
      # Weighted sum of rating times similarity
      scores.setdefault(item2,0)
      scores[item2]+=similarity*rating
      # Sum of all the similarities
      totalSim.setdefault(item2,0)
      totalSim[item2]+=similarity

  # Divide each total score by total weighing to get an average
  rankings=[(score/totalSim[item],item) for item,score in scores.items( )]

  # Return the rankings from highest to lowest
  rankings.sort( )
  rankings.reverse( )
  return rankings
```

Getting recommendations

```
>>> from recommendations import *
>>> itemsim = calculateSimilarItems(critics)
>>> print getRecommendedItems(critics, itemsim, 'Toby')
```

User-Based or Item-Based Filtering?

- Data Set Size
  - Large datasets: Item-based filtering is significantly faster than user-based
- What about the item similarity table?
  - Sparsity: In the movie data, since every critic has rated nearly every movie, the dataset is dense (not sparse).
  - Item-based filtering usually outperforms user-based filtering in ***sparse*** datasets (in terms of accuracy)
  - They perform about equally in **dense** datasets.

User-Based or Item-Based Filtering?

- User-based filtering
  - simpler to implement → no extra steps
  - more appropriate with **smaller in-memory datasets that change very frequently.**
  - In some cases, showing people other users with similar preferences has its own value
    - maybe not on a shopping site, but *possibly on a link-sharing or music recommendation site.*

Structuring

&

Visualization

What to learn?

- Data clustering: discovering and visualizing groups of things, people, or ideas that are all closely related.
- Graphical visualization of generated groups.

Clustering

- Automatically detect groups of customers with similar buying patterns, in addition to regular demographic information.
- People of similar age and income may have vastly different styles of dress, but with the use of clustering, **“fashion islands”** can be discovered and used to develop a retail or marketing strategy.

Running application

- We will look at **blogs**, the topics they discuss, and their particular word usage *to show that blogs can be grouped according to their text and that words can be grouped by their usage.*

Why to cluster blogs?

- By clustering blogs based on word frequencies, it might be possible to determine:
  - "there are groups of blogs that frequently write about similar subjects or write in similar styles"
  - Then, this information can be used to search, catalog, and discover within the blogosphere.

1st step!

- Prepare data for clustering by determining
    - a common set of numerical attributes
    - that can be used to compare the items.

Example

	"china"	"kids"	"music"	"yahoo"
Gothamist	0	3	3	0
GigaOM	6	0	0	2
Quick Online Tips	0	2	2	22

Preprocessing

- Almost all blogs can be read online or via their RSS feeds.
- An RSS feed is a simple XML document that contains information about the blog and all the entries.
- The first step in generating word counts for each blog is to parse these feeds. Universal Feed Parser is an excellent module. Install via:
- sudo pip install feedparser

Install feedparser

- PyCharm (like any other module installation)

or

- pip install feedparser

Sample Data Set

- Highly referenced blogs with clean data (mostly text)
  - feedlist.txt
  - Available on LMS (/Lectures/Week 06/Code)

RSS Feed or Atom Feed

- RSS and Atom feeds always have a title and a list of entries.
- Each entry usually has either a summary or description tag that contains the actual text of the entries.

Get word counts

```
# Returns title and dictionary of word counts for an RSS feed
def getwordcounts(url):
    # Parse the feed
    d =feedparser.parse(url)
    wc ={}

    # Loopover  all the entries
    for e in d.entries:
        if 'summary' in e:
            summary = e.summary
        else:
            summary = e.description

        # Extract a list of words
        words = getwords(e.title+' '+summary)
        for wordin  words:
            wc.setdefault(word,0)
            wc[word] += 1
    return d.feed.title, wc
```

Tokenize: Get Words

```
import re

# Strips out all of the HTML and splits the words by
# nonalphabetical characters and returns them as a list.

def getwords(html):
    # Remove all the HTML tags
    txt=re.compile(r'<[^>]+>').sub('',html)

    # Split words by all non-alpha characters
    words=re.compile(r'[^A-Z^a-z]+').split(txt)

    # Convert to lowercase
    return [word.lower() for word in words if word!='']
```