Programming Collective Intelligence

Building Smart Web 2.0 Applications

Toby Segaran

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

CHAPTER 2

Making Recommendations

To begin the tour of collective intelligence, I'm going to show you ways to use the preferences of a group of people to make recommendations to other people. There are many applications for this type of information, such as making product recommendations for online shopping, suggesting interesting web sites, or helping people find music and movies. This chapter shows you how to build a system for finding people who share tastes and for making automatic recommendations based on things that other people like.

You've probably come across recommendation engines before when using an online shopping site like Amazon. Amazon tracks the purchasing habits of all its shoppers, and when you log onto the site, it uses this information to suggest products you might like. Amazon can even suggest movies you might like, even if you've only bought books from it before. Some online concert ticket agencies will look at the history of shows you've seen before and alert you to upcoming shows that might be of interest. Sites like *reddit.com* let you vote on links to other web sites and then use your votes to suggest other links you might find interesting.

From these examples, you can see that preferences can be collected in many different ways. Sometimes the data are items that people have purchased, and opinions about these items might be represented as yes/no votes or as ratings from one to five. In this chapter, we'll look at different ways of representing these cases so that they'll all work with the same set of algorithms, and we'll create working examples with movie critic scores and social bookmarking.

Collaborative Filtering

You know that the low-tech way to get recommendations for products, movies, or entertaining web sites is to ask your friends. You also know that some of your friends have better "taste" than others, something you've learned over time by observing whether they usually like the same things as you. As more and more options become available, it becomes less practical to decide what you want by asking a small group of people, since they may not be aware of all the options. This is why a set of techniques called *collaborative filtering* was developed.

A collaborative filtering algorithm usually works by searching a large group of people and finding a smaller set with tastes similar to yours. It looks at other things they like and combines them to create a ranked list of suggestions. There are several different ways of deciding which people are similar and combining their choices to make a list; this chapter will cover a few of these.

The term *collaborative filtering* was first used by David Goldberg at Xerox PARC in 1992 in a paper called "Using collaborative filtering to weave an information tapestry." He designed a system called *Tapestry* that allowed people to annotate documents as either interesting or uninteresting and used this information to filter documents for other people.

There are now hundreds of web sites that employ some sort of collaborative filtering algorithm for movies, music, books, dating, shopping, other web sites, podcasts, articles, and even jokes.

Collecting Preferences

The first thing you need is a way to represent different people and their preferences. In Python, a very simple way to do this is to use a *nested dictionary*. If you'd like to work through the example in this section, create a file called *recommendations.py*, and insert the following code to create the dataset:

```
# A dictionary of movie critics and their ratings of a small
# set of movies
critics={'Lisa Rose': {'Lady in the Water': 2.5, 'Snakes on a Plane': 3.5,
 'Just My Luck': 3.0, 'Superman Returns': 3.5, 'You, Me and Dupree': 2.5,
 'The Night Listener': 3.0},
'Gene Seymour': {'Lady in the Water': 3.0, 'Snakes on a Plane': 3.5,
 'Just My Luck': 1.5, 'Superman Returns': 5.0, 'The Night Listener': 3.0,
 'You, Me and Dupree': 3.5},
'Michael Phillips': {'Lady in the Water': 2.5, 'Snakes on a Plane': 3.0,
 'Superman Returns': 3.5, 'The Night Listener': 4.0},
'Claudia Puig': {'Snakes on a Plane': 3.5, 'Just My Luck': 3.0,
 'The Night Listener': 4.5, 'Superman Returns': 4.0,
 'You, Me and Dupree': 2.5},
'Mick LaSalle': {'Lady in the Water': 3.0, 'Snakes on a Plane': 4.0,
 'Just My Luck': 2.0, 'Superman Returns': 3.0, 'The Night Listener': 3.0,
 'You, Me and Dupree': 2.0},
'Jack Matthews': {'Lady in the Water': 3.0, 'Snakes on a Plane': 4.0,
 'The Night Listener': 3.0, 'Superman Returns': 5.0, 'You, Me and Dupree': 3.5},
'Toby': {'Snakes on a Plane':4.5,'You, Me and Dupree':1.0,'Superman Returns':4.0}}
```

You will be working with Python interactively in this chapter, so you should save *recommendations.py* somewhere where the Python interactive interpreter can find it. This could be in the *python/Lib* directory, but the easiest way to do it is to start the Python interpreter in the same directory in which you saved the file.

This dictionary uses a ranking from 1 to 5 as a way to express how much each of these movie critics (and I) liked a given movie. No matter how preferences are expressed, you need a way to map them onto numerical values. If you were building a shopping site, you might use a value of 1 to indicate that someone had bought an item in the past and a value of 0 to indicate that they had not. For a site where people vote on news stories, values of –1, 0, and 1 could be used to represent "disliked," "didn't vote," and "liked," as shown in Table 2-1.

Table 2-1. Possible mappings of user actions to numerical scores

Concert tickets		Online shopping		Site recommender	
Bought	1	Bought	2	Liked	1
Didn't buy	0	Browsed	1	No vote	0
		Didn't buy	0	Disliked	–1

Using a dictionary is convenient for experimenting with the algorithms and for illustrative purposes. It's easy to search and modify the dictionary. Start your Python interpreter and try a few commands:

```
c:\code\collective\chapter2> python
Python 2.4.1 (#65, Mar 30 2005, 09:13:57) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>> from recommendations import critics
>> critics['Lisa Rose']['Lady in the Water']
2.5
>> critics['Toby']['Snakes on a Plane']=4.5
>> critics['Toby']
{'Snakes on a Plane':4.5, 'Superman Returns':4.0, 'You, Me and Dupree':1.0}
```

Although you can fit a large number of preferences in memory in a dictionary, for very large datasets you'll probably want to store preferences in a database.

Finding Similar Users

After collecting data about the things people like, you need a way to determine how similar people are in their tastes. You do this by comparing each person with every other person and calculating a *similarity score*. There are a few ways to do this, and in this section I'll show you two systems for calculating similarity scores: *Euclidean distance* and *Pearson correlation*.

Euclidean Distance Score

One very simple way to calculate a similarity score is to use a Euclidean distance score, which takes the items that people have ranked in common and uses them as axes for a chart. You can then plot the people on the chart and see how close together they are, as shown in Figure 2-1.

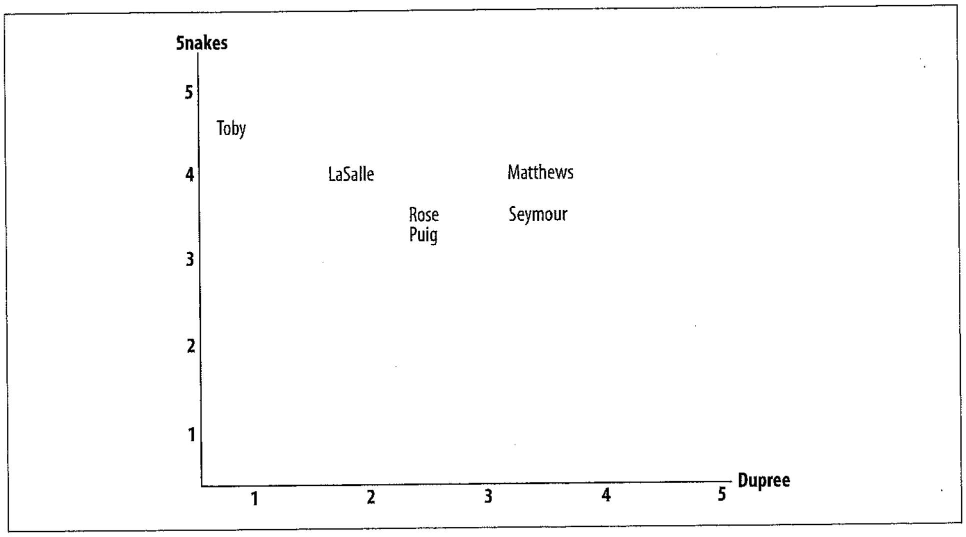


Figure 2-1. People in preference space

This figure shows the people charted in *preference space*. Toby has been plotted at 4.5 on the Snakes axis and at 1.0 on the Dupree axis. The closer two people are in the preference space, the more similar their preferences are. Because the chart is two-dimensional, you can only look at two rankings at a time, but the principle is the same for bigger sets of rankings.

To calculate the distance between Toby and LaSalle in the chart, take the difference in each axis, square them and add them together, then take the square root of the sum. In Python, you can use the `pow(n,2)` function to square a number and take the square root with the `sqrt` function:

```
>> from math import sqrt
>> sqrt(pow(4.5-4,2)+pow(1-2,2))
1.1180339887498949
```

This formula calculates the distance, which will be smaller for people who are more similar. However, you need a function that gives higher values for people who are similar. This can be done by adding 1 to the function (so you don't get a division-by-zero error) and inverting it:

```
>> 1/(1+sqrt(pow(4.5-4,2)+pow(1-2,2)))
0.47213595499957939
```

This new function always returns a value between 0 and 1, where a value of 1 means that two people have identical preferences. You can put everything together to create a function for calculating similarity. Add the following code to *recommendations.py*:

```
from math import sqrt

# Returns a distance-based similarity score for person1 and person2
def sim_distance(prefs,person1,person2):
  # Get the list of shared_items
  si={}
  for item in prefs[person1]:
    if item in prefs[person2]:
       si[item]=1

  # if they have no ratings in common, return 0
  if len(si)==0: return 0

  # Add up the squares of all the differences
  sum_of_squares=sum([pow(prefs[person1][item]-prefs[person2][item],2)
                      for item in si])

  return 1/(1+sqrt(sum_of_squares))
```

This function can be called with two names to get a similarity score. In your Python interpreter, run the following:

```
>>> import recommendations
>>> reload(recommendations)
>>> recommendations.sim_distance(recommendations.critics,
...    'Lisa Rose','Gene Seymour')
0.148148148148
```

This gives you a similarity score between Lisa Rose and Gene Seymour. Try it with other names to see if you can find people who have more or less in common.

Pearson Correlation Score

A slightly more sophisticated way to determine the similarity between people's interests is to use a Pearson correlation coefficient. The correlation coefficient is a measure of how well two sets of data fit on a straight line. The formula for this is more complicated than the Euclidean distance score, but it tends to give better results in situations where the data isn't well normalized—for example, if critics' movie rankings are routinely more harsh than average.

To visualize this method, you can plot the ratings of two of the critics on a chart, as shown in Figure 2-2. *Superman* was rated 3 by Mick LaSalle and 5 by Gene Seymour, so it is placed at (3,5) on the chart.

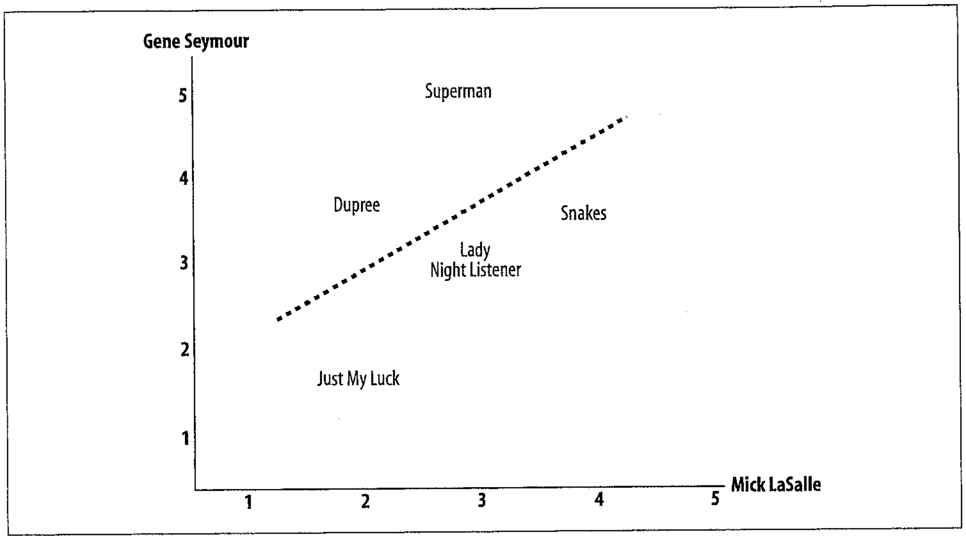


Figure 2-2. Comparing two movie critics on a scatter plot

You can also see a straight line on the chart. This is called the *best-fit line* because it comes as close to all the items on the chart as possible. If the two critics had identical ratings for every movie, this line would be diagonal and would touch every item in the chart, giving a perfect correlation score of 1. In the case illustrated, the critics disagree on a few movies, so the correlation score is about 0.4. Figure 2-3 shows an example of a much higher correlation, one of about 0.75.

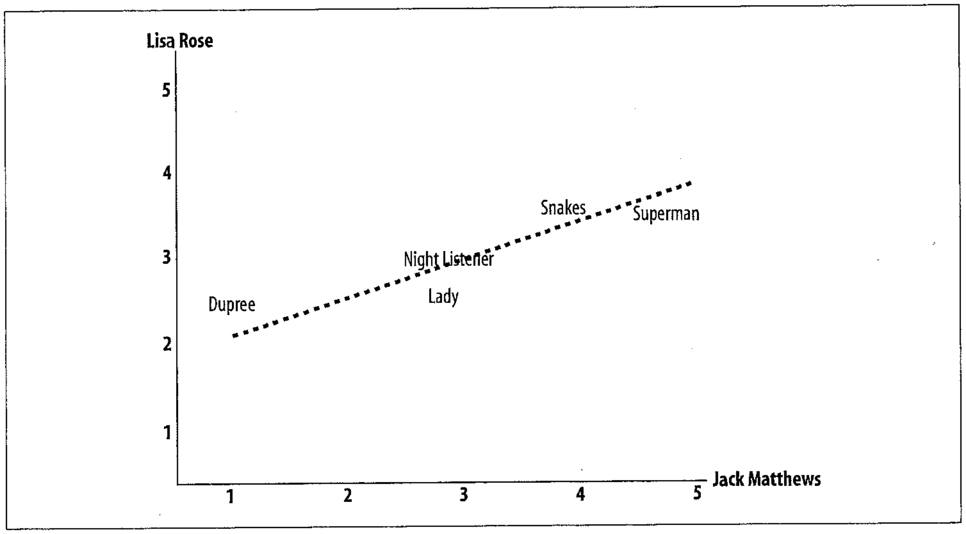


Figure 2-3. Two critics with a high correlation score

One interesting aspect of using the Pearson score, which you can see in the figure, is that it corrects for *grade inflation*. In this figure, Jack Matthews tends to give higher scores than Lisa Rose, but the line still fits because they have relatively similar preferences. If one critic is inclined to give higher scores than the other, there can still be perfect correlation if the difference between their scores is consistent. The Euclidean distance score described earlier will say that two critics are dissimilar because one is consistently harsher than the other, even if their tastes are very similar. Depending on your particular application, this behavior may or may not be what you want.

The code for the Pearson correlation score first finds the items rated by both critics. It then calculates the sums and the sum of the squares of the ratings for the two critics, and calculates the sum of the products of their ratings. Finally, it uses these results to calculate the Pearson correlation coefficient, shown in bold in the code below. Unlike the distance metric, this formula is not very intuitive, but it does tell you how much the variables change together divided by the product of how much they vary individually.

To use this formula, create a new function with the same signature as the sim_distance function in *recommendations.py*:

```
# Returns the Pearson correlation coefficient for p1 and p2
def sim_pearson(prefs,p1,p2):
  # Get the list of mutually rated items
  si={}
  for item in prefs[p1]:
    if item in prefs[p2]: si[item]=1

  # Find the number of elements
  n=len(si)

  # if they have no ratings in common, return 0
  if n==0: return 0

  # Add up all the preferences
  sum1=sum([prefs[p1][it] for it in si])
  sum2=sum([prefs[p2][it] for it in si])

  # Sum up the squares
  sum1Sq=sum([pow(prefs[p1][it],2) for it in si])
  sum2Sq=sum([pow(prefs[p2][it],2) for it in si])

  # Sum up the products
  pSum=sum([prefs[p1][it]*prefs[p2][it] for it in si])

  # Calculate Pearson score
  num=pSum-(sum1*sum2/n)
  den=sqrt((sum1Sq-pow(sum1,2)/n)*(sum2Sq-pow(sum2,2)/n))
  if den==0: return 0

  r=num/den

  return r
```

This function will return a value between –1 and 1. A value of 1 means that the two people have exactly the same ratings for every item. Unlike with the distance metric, you don't need to change this value to get it to the right scale. Now you can try getting the correlation score for Figure 2-3:

```
>>> reload(recommendations)
>>> print recommendations.sim_pearson(recommendations.critics,
...  'Lisa Rose','Gene Seymour')
0.396059017191
```

Which Similarity Metric Should You Use?

I've introduced functions for two different metrics here, but there are actually many more ways to measure similarity between two sets of data. The best one to use will depend on your application, and it is worth trying Pearson, Euclidean distance, or others to see which you think gives better results.

The functions in the rest of this chapter have an optional *similarity* parameter, which points to a function to make it easier to experiment: specify `sim_pearson` or `sim_distance` to choose which similarity parameter to use. There are many other functions such as the *Jaccard coefficient* or *Manhattan distance* that you can use as your similarity function, as long as they have the same signature and return a float where a higher value means more similar.

You can read about other metrics for comparing items at *http://en.wikipedia.org/wiki/Metric_%28mathematics%29#Examples*.

Ranking the Critics

Now that you have functions for comparing two people, you can create a function that scores everyone against a given person and finds the closest matches. In this case, I'm interested in learning which movie critics have tastes simliar to mine so that I know whose advice I should take when deciding on a movie. Add this function to *recommendations.py* to get an ordered list of people with similar tastes to the specified person:

```
# Returns the best matches for person from the prefs dictionary.
# Number of results and similarity function are optional params.
def topMatches(prefs,person,n=5,similarity=sim_pearson):
  scores=[(similarity(prefs,person,other),other)
                  for other in prefs if other!=person]

  # Sort the list so the highest scores appear at the top
  scores.sort( )
  scores.reverse( )
  return scores[0:n]
```

This function uses a Python *list comprehension* to compare me to every other user in the dictionary using one of the previously defined distance metrics. Then it returns the first *n* items of the sorted results.

Calling this function with my own name gives me a list of movie critics and their similarity scores:

```
>> reload(recommendations)
>> recommendations.topMatches(recommendations.critics,'Toby',n=3)
[(0.99124070716192991, 'Lisa Rose'), (0.92447345164190486, 'Mick LaSalle'),
 (0.89340514744156474, 'Claudia Puig')]
```

From this I know that I should be reading reviews by Lisa Rose, as her tastes tend to be similar to mine. If you've seen any of these movies, you can try adding yourself to the dictionary with your preferences and see who your favorite critic should be.

Recommending Items

Finding a good critic to read is great, but what I really want is a movie recommendation right now. I could just look at the person who has tastes most similar to mine and look for a movie he likes that I haven't seen yet, but that would be too permissive. Such an approach could accidentally turn up reviewers who haven't reviewed some of the movies that I might like. It could also return a reviewer who strangely liked a movie that got bad reviews from all the other critics returned by `topMatches`.

To solve these issues, you need to score the items by producing a weighted score that ranks the critics. Take the votes of all the other critics and multiply how similar they are to me by the score they gave each movie. Table 2-2 shows how this process works.

Table 2-2. Creating recommendations for Toby

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

This table shows correlation scores for each critic and the ratings they gave the three movies (*The Night Listener*, *Lady in the Water*, and *Just My Luck*) that I haven't rated. The columns beginning with S.x give the similarity multiplied by the rating, so a person who is similar to me will contribute more to the overall score than a person who is different from me. The Total row shows the sum of all these numbers.

You could just use the totals to calculate the rankings, but then a movie reviewed by more people would have a big advantage. To correct for this, you need to divide by the sum of all the similarities for critics that reviewed that movie (the Sim. Sum row in the table). Because *The Night Listener* was reviewed by everyone, its total is divided by the sum of all the similarities. *Lady in the Water*, however, was not reviewed by Puig, so the movie's score is divided by the sum of all the other similarities. The last row shows the results of this division.

The code for this is pretty straightforward, and it works with either the Euclidean distance or the Pearson correlation score. Add it to *recommendations.py*:

```
# Gets recommendations for a person by using a weighted average
# of every other user's rankings
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] or prefs[person][item]==0:
        # 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
```

This code loops through every other person in the prefs dictionary. In each case, it calculates how similar they are to the person specified. It then loops through every item for which they've given a score. The line in bold shows how the final score for an item is calculated—the score for each item is multiplied by the similarity and these products are all added together. At the end, the scores are normalized by dividing each of them by the similarity sum, and the sorted results are returned.

Now you can find out what movies I should watch next:

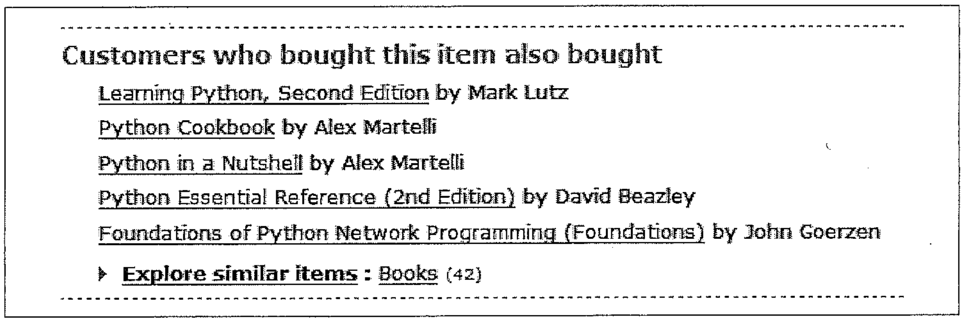
```
>>> reload(recommendations)
>>> recommendations.getRecommendations(recommendations.critics,'Toby')
[(3.3477895267131013, 'The Night Listener'), (2.8325499182641614, 'Lady in the
Water'), (2.5309807037655645, 'Just My Luck')]
>>> recommendations.getRecommendations(recommendations.critics,'Toby',
...    similarity=recommendations.sim_distance)
[(3.5002478401415877, 'The Night Listener'), (2.7561242939959363, 'Lady in the
Water'), (2.4619884860743739, 'Just My Luck')]
```

Not only do you get a ranked list of movies, but you also get a guess at what my rating for each movie would be. This report lets me decide if I want to watch a movie at all, or if I'd rather do something else entirely. Depending on your application, you may decide not to give a recommendation if there's nothing that would meet a given user's standards. You'll find that the results are only affected very slightly by the choice of similarity metric.

You've now built a complete recommendation system, which will work with any type of product or link. All you have to do is set up a dictionary of people, items, and scores, and you can use this to create recommendations for any person. Later in this chapter you'll see how you can use the del.icio.us API to get real data for recommending web sites to people.

Matching Products

Now you know how to find similar people and recommend products for a given person, but what if you want to see which products are similar to each other? You may have encountered this on shopping web sites, particularly when the site hasn't collected a lot of information about you. A section of Amazon's web page for the book *Programming Python* is shown in Figure 2-4.



Customers who bought this item also bought

Learning Python, Second Edition by Mark Lutz

Python Cookbook by Alex Martelli

Python in a Nutshell by Alex Martelli

Python Essential Reference (2nd Edition) by David Beazley

Foundations of Python Network Programming (Foundations) by John Goerzen

▸ **Explore similar items** : Books (42)

Figure 2-4. Amazon shows products that are similar to Programming Python

In this case, you can determine similarity by looking at who liked a particular item and seeing the other things they liked. This is actually the same method we used earlier to determine similarity between people—you just need to swap the people and the items. So you can use the same methods you wrote earlier if you transform the dictionary from:

```
{'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..
```

Add a function to *recommendations.py* to do this 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
```

And now call the topMatches function used earlier to find the set of movies most similar to *Superman Returns*:

```
>> reload(recommendations)
>> movies=recommendations.transformPrefs(recommendations.critics)
>> recommendations.topMatches(movies,'Superman Returns')
[(0.657, 'You, Me and Dupree'), (0.487, 'Lady in the Water'), (0.111, 'Snakes on a
Plane'), (-0.179, 'The Night Listener'), (-0.422, 'Just My Luck')]
```

Notice that in this example there are actually some negative correlation scores, which indicate that those who like *Superman Returns* tend to dislike *Just My Luck*, as shown in Figure 2-5.

To twist things around even more, you can get recommended critics for a movie. Maybe you're trying to decide whom to invite to a premiere?

```
>> recommendations.getRecommendations(movies,'Just My Luck')
[(4.0, 'Michael Phillips'), (3.0, 'Jack Matthews')]
```

It's not always clear that flipping people and items will lead to useful results, but in many cases it will allow you to make interesting comparisons. An online retailer might collect purchase histories for the purpose of recommending products to individuals. Reversing the products with the people, as you've done here, would allow them to search for people who might buy certain products. This might be very useful in planning a marketing effort for a big clearance of certain items. Another potential use is making sure that new links on a link-recommendation site are seen by the people who are most likely to enjoy them.

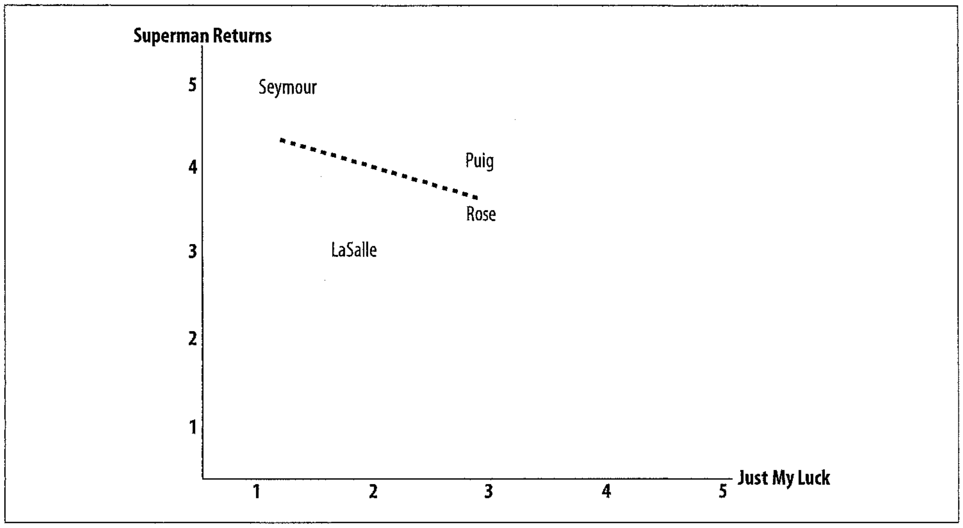
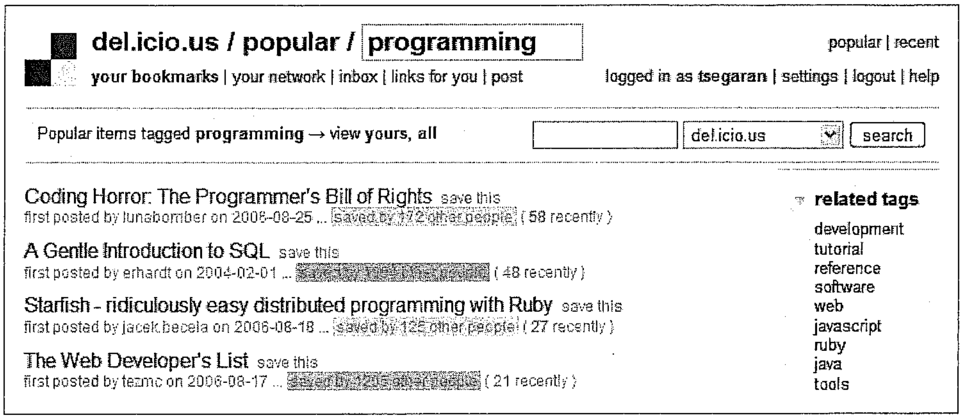


Figure 2-5. Superman Returns and Just My Luck have a negative correlation

Building a del.icio.us Link Recommender

This section shows you how to retrieve data from one of the most popular online bookmarking sites, and how to use that data to find similar users and recommend links they haven't seen before. This site, which you can access at *http://del.icio.us*, allows people to set up an account and post links that interest them for later reference. You can visit the site and look at links that other people have posted, and also browse "popular" links that have been posted by many different people. A sample page from del.icio.us is shown in Figure 2-6.



del.icio.us / popular / programming

popular | recent

your bookmarks | your network | inbox | links for you | post

logged in as **tsegaran** | settings | logout | help

Popular items tagged **programming** → **view yours, all**

del.icio.us

search

Coding Horror: The Programmer's Bill of Rights save this
first posted by lunabomber on 2006-08-25 ... saved by 172 other people (58 recently)

A Gentle Introduction to SQL save this
first posted by erhardt on 2004-02-01 ... [illegible] (48 recently)

Starfish - ridiculously easy distributed programming with Ruby save this
first posted by jacek.becela on 2006-08-18 ... saved by 125 other people (27 recently)

The Web Developer's List save this
first posted by teamc on 2006-08-17 ... saved by 1205 other people (21 recently)

related tags

development
tutorial
reference
software
web
javascript
ruby
java
tools

Figure 2-6. The del.icio.us popular page for programming

Unlike some link-sharing sites, del.icio.us doesn't (at the time of writing) include any way to find similar people or recommend links you might like. Fortunately, you can use the techniques discussed in this chapter to add that functionality yourself.

The del.icio.us API

Data from del.icio.us is made available through an API that returns data in XML format. To make things even easier for you, there is a Python API that you can download from *http://code.google.com/p/pydelicious/source* or *http://oreilly.com/catalog/9780596529321*. (Note that with some Python installations you may need to install elementtree from http://effbot.org/downloads/#elementtree to get pydelicious to work.)

To work through the example in this section, you'll need to download the latest version of this library and put it in your Python library path. (See Appendix A for more information on installing this library.)

This library has several simple calls to get links that people have submitted. For example, to get a list of recent popular posts about programming, you can use the get_popular call:

```
>> import pydelicious
>> pydelicious.get_popular(tag='programming')
[{'count': '', 'extended': '', 'hash': '', 'description': u'How To Write
Unmaintainable Code', 'tags': '', 'href': u'http://thc.segfault.net/root/phun/
unmaintain.html', 'user': u'dorsia', 'dt': u'2006-08-19T09:48:56Z'}, {'count': '',
'extended': '', 'hash': '', 'description': u'Threading in C#', 'tags': '', 'href':
u'http://www.albahari.com/threading/', 'user': u'mmihale', 'dt': u'2006-05-17T18:09:
24Z'},
...etc...
```

You can see that it returns a list of dictionaries, each one containing a URL, description, and the user who posted it. Since you are working from live data, your results will look different from the examples. There are two other calls you'll be using, get_urlposts, which returns all the posts for a given URL, and get_userposts, which returns all the posts for a given user. The data for these calls is returned in the same way.

Building the Dataset

It's not possible to download the full set of all user posts from del.icio.us, so you'll need to choose a subset of them. You could do this any way you like, but to make the example show interesting results, it would be good to find people who post frequently and have some similar posts.

One way to do this is to get a list of users who recently posted a popular link with a specified tag. Create a new file called *deliciousrec.py* and enter the following code:

```
from pydelicious import get_popular,get_userposts,get_urlposts

def initializeUserDict(tag,count=5):
  user_dict={}
  # get the top "count" popular posts
  for p1 in get_popular(tag=tag)[0:count]:
    # find all users who posted this
    for p2 in get_urlposts(p1['href']):
      user=p2['user']
      user_dict[user]={}
  return user_dict
```

This will give you a dictionary with some users, each referencing an empty dictionary waiting to be filled with links. The API only returns the last 30 people to post the link, so the function gathers users from the first 5 links to build a larger set.

Unlike the movie critic dataset, there are only two possible ratings in this case: 0 if the user did not post this link, and 1 if he did. Using the API, you can now create a function to fill in ratings for all the users. Add this code to *deliciousrec.py*:

```
import time
def fillItems(user_dict):
  all_items={}
  # Find links posted by all users
  for user in user_dict:
    for i in range(3):
      try:
        posts=get_userposts(user)
        break
      except:
        print "Failed user "+user+", retrying"
        time.sleep(4)
    for post in posts:
      url=post['href']
      user_dict[user][url]=1.0
      all_items[url]=1

  # Fill in missing items with 0
  for ratings in user_dict.values():
    for item in all_items:
      if item not in ratings:
        ratings[item]=0.0
```

This can be used to build a dataset similar to the critics dictionary you created by hand at the beginning of this chapter:

```
>> from deliciousrec import *
>> delusers=initializeUserDict('programming')
>> delusers ['tsegaran']={} # Add yourself to the dictionary if you use delicious
>> fillItems(delusers)
```

The third line adds the user tsegaran to the list. You can replace tsegaran with your own username if you use del.icio.us.

The call to fillItems may take several minutes to run, as it is making a few hundred requests to the site. Sometimes the API blocks requests that are repeated too rapidly. In this case, the code pauses and retries the user up to three times.

Recommending Neighbors and Links

Now that you've built a dataset, you can apply the same functions that you used before on the movie critic dataset. To select a user at random and find other users who have tastes similar to his, enter this code in your Python session:

```
>> import random
>> user=delusers.keys()[random.randint(0,len(delusers)-1)]
>> user
u'veza'
>> recommendations.topMatches(delusers,user)
[(0.083, u'kuzz99'), (0.083, u'arturoochoa'), (0.083, u'NickSmith'), (0.083,
u'MichaelDahl'), (0.050, u'zinggoat')]
```

You can also get recommendations for links for this user by calling getRecommendations. This will return all the items in order, so it's better to restrict it to the top 10:

```
>> recommendations.getRecommendations(delusers,user)[0:10]
[(0.278, u'http://www.devlisting.com/'),
(0.276, u'http://www.howtoforge.com/linux_ldap_authentication'),
(0.191, u'http://yarivsblog.com/articles/2006/08/09/secret-weapons-for-startups'),
(0.191, u'http://www.dadgum.com/james/performance.html'),
(0.191, u'http://www.codinghorror.com/blog/archives/000666.html')]
```

Of course, as demonstrated earlier, the preferences list can be transposed, allowing you to frame your searches in terms of links rather than people. To find a set of links similar to one that you found particularly interesting, you can try:

```
>> url=recommendations.getRecommendations(delusers,user)[0][1]
>> recommendations.topMatches(recommendations.transformPrefs(delusers),url)
[(0.312, u'http://www.fonttester.com/'),
(0.312, u'http://www.cssremix.com/'),
(0.266, u'http://www.logoorange.com/color/color-codes-chart.php'),
(0.254, u'http://yotophoto.com/'),
(0.254, u'http://www.wpdfd.com/editorial/basics/index.html')]
```

That's it! You've successfully added a recommendation engine to del.icio.us. There's a lot more that could be done here. Since del.icio.us supports searching by tags, you can look for tags that are similar to each other. You can even search for people trying to manipulate the "popular" pages by posting the same links with multiple accounts.

Item-Based Filtering

The way the recommendation engine has been implemented so far requires the use of all the rankings from every user in order to create a dataset. This will probably work well for a few thousand people or items, but a very large site like Amazon has millions of customers and products—comparing a user with every other user and then comparing every product each user has rated can be very slow. 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.

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

The procedure for item-based filtering draws a lot on what we have already discussed. The general technique is to precompute the most similar items for each item. Then, when you wish to make recommendations to a user, you look at his top-rated items and create a weighted list of the items most similar to those. The important difference here is that, although the first step requires you to examine all the data, *comparisons between items will not change as often as comparisons between users*. This means you do not have to continuously calculate each item's most similar items—you can do it at low-traffic times or on a computer separate from your main application.

Building the Item Comparison Dataset

To compare items, the first thing you'll need to do is write a function to build the complete dataset of similar items. Again, this does not have to be done every time a recommendation is needed—instead, you build the dataset once and reuse it each time you need it.

To generate the dataset, add the following function to *recommendations.py*:

```
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)
  c=0
  for item in itemPrefs:
    # Status updates for large datasets
    c+=1
    if c%100==0: print "%d / %d" % (c,len(itemPrefs))
    # Find the most similar items to this one
    scores=topMatches(itemPrefs,item,n=n,similarity=sim_distance)
    result[item]=scores
  return result
```

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.

In your Python session, build the item similarity dataset and see what it looks like:

```
>>> reload(recommendations)
>>> itemsim=recommendations.calculateSimilarItems(recommendations.critics)
>>> itemsim
{'Lady in the Water': [(0.40000000000000002, 'You, Me and Dupree'),
                       (0.2857142857142857, 'The Night Listener'),...
 'Snakes on a Plane': [(0.22222222222222221, 'Lady in the Water'),
                       (0.18181818181818182, 'The Night Listener'),...
etc.
```

Remember, this function only has to be run frequently enough to keep the item similarities up to date. You will need to do this more often early on when the user base and number of ratings is small, but as the user base grows, the similarity scores between items will usually become more stable.

Getting Recommendations

Now you're ready to give recommendations using the item similarity dictionary without going through the whole dataset. You're going to get all the items that the user has ranked, find the similar items, and weight them according to how similar they are. The items dictionary can easily be used to get the similarities.

Table 2-3 shows the process of finding recommendations using the item-based approach. Unlike Table 2-2, the critics are not involved at all, and instead there is a grid of movies I've rated versus movies I haven't rated.

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		2.473		2.598

Each row has a movie that I have already seen, along with my personal rating for it. For every movie that I haven't seen, there's a column that shows how similar it is to the movies I have seen—for example, the similarity score between *Superman* and *The Night Listener* is 0.103. The columns starting with R.x show my rating of the movie multiplied by the similarity—since I rated *Superman* 4.0, the value next to Night in the Superman row is 4.0×0.103 = 0.412.

The total row shows the total of the similarity scores and the total of the R.x columns for each movie. To predict what my rating would be for each movie, just divide the total for the R.x column by the total for the similarity column. My predicted rating for *The Night Listener* is thus 1.378/0.433 = 3.183.

You can use this functionality by adding one last function to *recommendations.py*:

```
def getRecommendedItems(prefs,itemMatch,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 itemMatch[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 weighting 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
```

You can try this function with the similarity dataset you built earlier to get the new recommendations for Toby:

```
>> reload(recommendations)
>> recommendations.getRecommendedItems(recommendations.critics,itemsim,'Toby')
[(3.182, 'The Night Listener'),
 (2.598, 'Just My Luck'),
 (2.473, 'Lady in the Water')]
```

The Night Listener still comes in first by a significant margin, and *Just My Luck* and *Lady in the Water* have changed places although they are still close together. More importantly, the call to getRecommendedItems did not have to calculate the similarities scores for all the other critics because the item similarity dataset was built in advance.

Using the MovieLens Dataset

For the final example, let's look at a real dataset of movie ratings called *MovieLens*. MovieLens was developed by the GroupLens project at the University of Minnesota. You can download the dataset from *http://www.grouplens.org*, click on "Data Sets", and then "Movie Lens Data Sets". There are two datasets here. Download the 100,000 dataset in either *tar.gz* format or *zip* format, depending on your platform.

The archive contains several files, but the ones of interest are *u.item*, which contains a list of movie IDs and titles, and *u.data*, which contains actual ratings in this format:

```
196  242   3       881250949
186  302   3       891717742
22   377   1       878887116
244  51    2       880606923
166  346   1       886397596
298  474   4       884182806
```

Each line has a user ID, a movie ID, the rating given to the movie by the user, and a timestamp. You can get the movie titles, but the user data is anonymous, so you'll just be working with user IDs in this section. The set contains ratings of 1,682 movies by 943 users, each of whom rated at least 20 movies.

Create a new method called loadMovieLens in *recommendations.py* to load this dataset:

```
def loadMovieLens(path='/data/movielens'):

  # Get movie titles
  movies={}
  for line in open(path+'/u.item'):
    (id,title)=line.split('|')[0:2]
    movies[id]=title

  # Load data
  prefs={}
  for line in open(path+'/u.data'):
    (user,movieid,rating,ts)=line.split('\t')
    prefs.setdefault(user,{})
    prefs[user][movies[movieid]]=float(rating)
  return prefs
```

In your Python session, load the data and look at some ratings for any arbitrary user:

```
>>> reload(recommendations)
>>> prefs=recommendations.loadMovieLens()
>>> prefs['87']
{'Birdcage, The (1996)': 4.0, 'E.T. the Extra-Terrestrial (1982)': 3.0,
 'Bananas (1971)': 5.0, 'Sting, The (1973)': 5.0, 'Bad Boys (1995)': 4.0,
 'In the Line of Fire (1993)': 5.0, 'Star Trek: The Wrath of Khan (1982)': 5.0,
'Speechless (1994)': 4.0, etc...
```

Now you can get user-based recommendations:

```
>>> recommendations.getRecommendations(prefs,'87')[0:30]
[(5.0, 'They Made Me a Criminal (1939)'), (5.0, 'Star Kid (1997)'),
 (5.0, 'Santa with Muscles (1996)'), (5.0, 'Saint of Fort Washington (1993)'),
 etc...]
```

Depending on the speed of your computer, you may notice a pause when getting recommendations this way. This is because you're working with a much larger dataset now. The more users you have, the longer user-based recommendations will take. Now try doing item-based recommendations instead:

```
>>> itemsim=recommendations.calculateSimilarItems(prefs,n=50)
100 / 1664
200 / 1664
etc...
>>> recommendations.getRecommendedItems(prefs,itemsim,'87')[0:30]
[(5.0, "What's Eating Gilbert Grape (1993)"), (5.0, 'Vertigo (1958)'),
 (5.0, 'Usual Suspects, The (1995)'), (5.0, 'Toy Story (1995)'),etc...]
```

Although building the item similarity dictionary takes a long time, recommendations are almost instantaneous after it's built. Furthermore, the time it takes to get recommendations will not increase as the number of users increases.

This is a great dataset to experiment with to see how different scoring methods affect the outcomes, and to understand how item-based and user-based filtering perform differently. The GroupLens web site has a few other datasets to play with, including books, jokes, and more movies.

User-Based or Item-Based Filtering?

Item-based filtering is significantly faster than user-based when getting a list of recommendations for a large dataset, but it does have the additional overhead of maintaining the item similarity table. Also, there is a difference in accuracy that depends on how "sparse" the dataset is. In the movie example, since every critic has rated nearly every movie, the dataset is dense (not sparse). On the other hand, it would be unlikely to find two people with the same set of del.icio.us bookmarks—most bookmarks are saved by a small group of people, leading to a sparse dataset. Item-based filtering usually outperforms user-based filtering in sparse datasets, and the two perform about equally in dense datasets.

> To learn more about the difference in performance between these algorithms, check out a paper called "Item-based Collaborative Filtering Recommendation Algorithms" by Sarwar et al. at *http://citeseer.ist.psu.edu/sarwar01itembased.html.*

Having said that, user-based filtering is simpler to implement and doesn't have the extra steps, so it is often more appropriate with smaller in-memory datasets that change very frequently. Finally, in some applications, showing people which other users have preferences similar to their own has its own value—maybe not something you would want to do on a shopping site, but possibly on a link-sharing or music recommendation site.

You've now learned how to calculate similarity scores and how to use these to compare people and items. This chapter covered two different recommendation algorithms, user-based and item-based, along with ways to persist people's preferences and use the del.icio.us API to build a link recommendation system. In Chapter 3, you'll see how to build on some of the ideas from this chapter by finding groups of similar people using unsupervised clustering algorithms. Chapter 9 will look at alternative ways to match people when you already know the sort of people they like.

Exercises

1. *Tanimoto score*. Find out what a Tanimoto similarity score is. In what cases could this be used as the similarity metric instead of Euclidean distance or Pearson coefficient? Create a new similarity function using the Tanimoto score.
2. *Tag similarity*. Using the del.icio.us API, create a dataset of tags and items. Use this to calculate similarity between tags and see if you can find any that are almost identical. Find some items that could have been tagged "programming" but were not.
3. *User-based efficiency*. The user-based filtering algorithm is inefficient because it compares a user to all other users every time a recommendation is needed. Write a function to precompute user similarities, and alter the recommendation code to use only the top five other users to get recommendations.
4. *Item-based bookmark filtering*. Download a set of data from del.icio.us and add it to the database. Create an item-item table and use this to make item-based recommendations for various users. How do these compare to the user-based recommendations?
5. *Audioscrobbler*. Take a look at *http://www.audioscrobbler.net*, a dataset containing music preferences for a large set of users. Use their web services API to get a set of data for making and building a music recommendation system.