This section is contributed to how 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 section 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.

There are different recommendation engines of online shopping sites like Amazon. Amazon tracks the purchasing habits of all its shoppers, and when a user 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.

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 section, we'll look at different ways of representing these cases so that they'll all work with the same set of algorithms.

Collaborative Filtering

One of the simplest way to get recommendations for products, movies, or entertaining web sites is to ask friends. Some of the friends have better "taste" than others, something we'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 we 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 ours. 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.

Collecting Preferences

The first thing we 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*.

This dictionary uses a ranking from 1 to 5 as a way to express how much each of these movie critics 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, we 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".

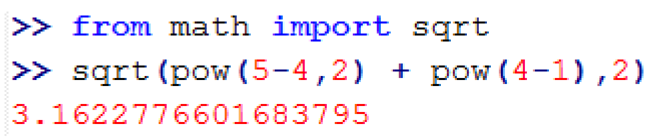
Finding Similar Users

After collecting data about the things people like, we need a way to determine how similar people are in their tastes. We do this by comparing each person with every other person and calculating a *similarity score*. There are a few ways to do this, and we have chosen: *Euclidean distance* and *Pearson correlation*.

Euclidean Distance Score

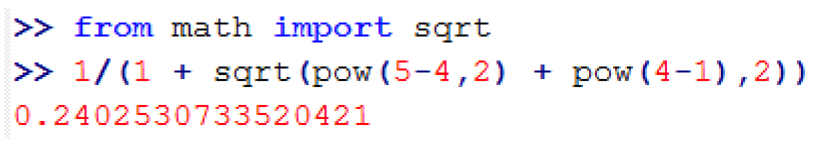
This is a simplest way to calculate a similarity score, which takes the items that people have ranked in common and uses them as axes for a chart.

To calculate the distance between two users 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, we 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(5-4,2) + pow(4-1),2)
3.1622776601683795
```

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



```
>> from math import sqrt
>> 1/(1 + sqrt(pow(5-4,2) + pow(4-1),2))
0.2402530733520421
```

This new function returns a value between 0 and 1, where a value of 1 means that two people have identical preference.

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.

```
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 are 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
```

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 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.

Which Similarity Metric Should We Use?

There are actually many more ways to measure similarity between two sets of data. The best one to use will depend on 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 report have an optional *similarity* parameter, which points to a function to make it easier to experiment: specify sim_pearson or sim_ vector to choose which similarity parameter to use. There are many other functions such as the *Jaccard coefficient* or *Manhattan distance* that we can use as similarity function.

Ranking the Critics

Now that we have functions for comparing two people, we can create a function that scores everyone against a given person and finds the closest matches. In this case, we are interested in learning which movie critics have tastes similar as ours so that we know whose advice we should take when deciding on a movie.

Recommending Items

Finding a good critic to read is great, but what we really want is a movie recommendation right now. We could just look at the person who has tastes most similar as ours and look for a movie he likes that we 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 we 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, we 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 us by the score they gave each movie.

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
Mathhews	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

The 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 we haven't rated. The columns beginning with S.x give the similarity multiplied by the rating, so a person who is similar to us will contribute more to the overall score than a person who is different from us. The Total row shows the sum of all these numbers.

We 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, we 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.

```
>>> reload(recommendations)
>>> recommendations.getRecommendations(recommendations.
critics,'Toby')
[(3.3477895267131013, 'The Night Listener'), (
2.832549918264161 4, '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')]
```

Matching Products

Now we know how to find similar people and recommend products for a given person, but what if we want to see which products are similar to each other? We may have encountered this on shopping web sites, particularly when the site hasn't collected a lot of information about a user.

In this case, we 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—we just need to swap the people and the items.

```
{'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..
```

It's not always clear that flipping people and items will lead to useful results, but in many cases it will allow us 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 we'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.

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 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.

We need to first invert the score dictionary using the transformPrefs function, 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.

```
>>> 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'),...
```

Remember, this function only has to be run frequently enough to keep the item similarities up to date. We 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.

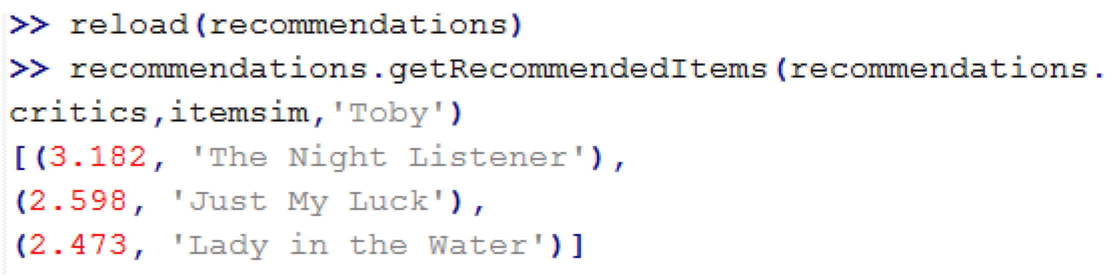
Getting Recommendations

We'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.

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
Normalize			3.183		2.598		2.483

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.



```
>> 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

We took a real dataset of movie ratings called *MovieLens*. MovieLens was developed by the GroupLens project at the University of Minnesota. We used dataset with 0.1 million entries.

The archive of MovieLens 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. We can get the movie titles, but the user data is anonymous, so we will be working with user IDs. The set contains rating of 1,682 movies by 943 users, each of whom rated at least 20 movies.

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 work 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.

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.

References:

1. Toby Segaran. 2007. *Programming Collective Intelligence* (First ed.). O'Reilly.
2. http://grouplens.org/datasets/movielens/
3. https://en.wikipedia.org/wiki/Metric_%28mathematics%29#Examples
4. G. Linden, B. Smith, J. York, Amazon.com recommendations: item-to-item collaborative filtering. IEEE Internet Comput. 7(1), 76–80 (2003)