

Chapter 3: Collaborative filtering

In what I like to call the awesome chapter 2 we learned the basics of collaborative filtering and recommendation systems. Users rate different items and we find users who have similar ratings. In the examples in that chapter, users rated items on a 5 or 10 point scale. As was mentioned in that chapter, there is some evidence to suggest that for many applications users typically don't use this fine-grain distinction and instead tend to give either the top rating or the lowest rating. The algorithms are general purpose and can be used with a variety of data.

One way of distinguishing types of user preferences is whether they are explicit or implicit.

Explicit ratings

Explicit ratings are when the user herself explicitly rates the item. For example, the star system on Amazon is such a system:

★★★★★ **Practical Knowledge**
I have found many of Seneca's letters to be useful in my life. I have no problem recommending this book to others.
Published 4 months ago by F. Spohr

★★★★★ **Simply a masterpiece.**
Reading this book is like spotting a light on the shore while sailing at night.

It is an amazing series of letters that will surprise you with the clarity of...

[Read more](#)

Published 6 months ago by Stefano Natoli

Here the user explicitly (intentionally) rates an item using number of stars.

Another example would be the thumb up / thumbs down ratings-- for example Pandora:







Implicit Ratings

For implicit ratings, we don't ask users to do any ratings—we just observe their behavior. An example of this is keeping track of what a user clicks on in the online New York Times.



After observing what a user clicks on for a few weeks you can imagine that we could develop a reasonable profile of the user—he doesn't like sports but seems to like technology news. If the user clicks on the article *Fastest Way to Lose Weight Discovered by Professional Trainers* and the article *Slow and Steady: How to lose weight and keep it off* perhaps he wishes to lose weight. If he clicks on the Office 2010 ad, he perhaps has an interest in that product. (By the way, the term used when a user clicks on an ad is called 'click through'.) Or consider what information we can gain from recording what products a user clicks on in Amazon. On your personalized Amazon front page this information is displayed:

More Items to Consider

You viewed	Customers who viewed this also viewed		
 Wolfgang Amadeus Phoenix Phoenix Audio CD \$11.98 \$9.99	 XX XX Audio CD \$14.98 \$9.99	 High Violet The National Audio CD \$14.98 \$9.88	 Broken Bells Broken Bells Audio CD \$11.98 \$9.99

So here in this example, Amazon keeps track of what people click on. It knows, for example, that users who viewed the Wolfgang Amadeus Phoenix product page, also viewed the XX product page.

Another implicit rating is what the user actually buys. Amazon also keeps track of this information. For example, on the page for Paolo Bacigalupi's *The Windup Girl* we get the following recommendations:

Customers Who Bought This Item Also Bought

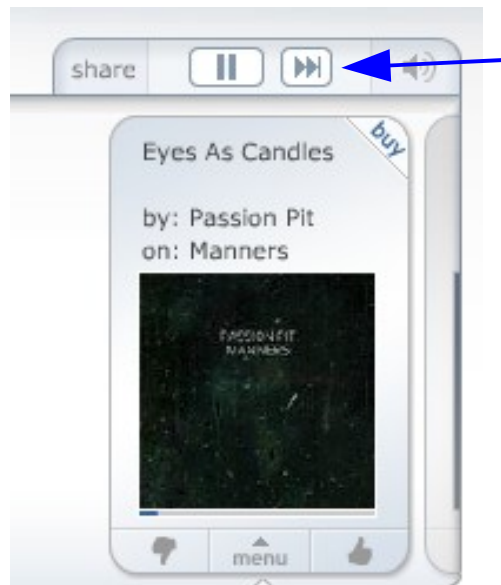


The screenshot shows four book recommendations under the heading "Customers Who Bought This Item Also Bought". Each recommendation includes a book cover, a "LOOK INSIDE!" button, the title, author, star rating, number of reviews, and price.

Book Title	Author	Star Rating	Reviews	Price
Boneshaker (Sci Fi Essential Books)	Cherie Priest	★★★★☆	(92)	\$10.87
The City & The City	China Mieville	★★★★☆	(109)	\$10.20
River of Gods	Ian McDonald	★★★★☆	(54)	\$10.87
Julian Comstock: A Story of 22nd-Century...	Robert Charles Wilson	★★★★☆	(21)	\$8.99

(Note the “Customers Who Bought This Item Also Bought” heading.) You would think that there could be the potential for some weird recommendations but this works surprisingly well.

One final example of implicit data is of Pandora recording when a user skips over a song:



Skipping over a song counts as a negative rating for that song.

Imagine what information a program can get by monitoring your behavior on iTunes.

Name	Time	Artist	Album	Genre	Rating	Play Count
<input checked="" type="checkbox"/> Between the Lines	4:35	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	16
<input checked="" type="checkbox"/> Love Song	4:19	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	15
<input checked="" type="checkbox"/> Vegas	4:08	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	15
<input checked="" type="checkbox"/> Bottle It Up	3:01	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	12
<input checked="" type="checkbox"/> Gravity	3:53	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	12
<input checked="" type="checkbox"/> One Sweet Love	4:21	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	11
<input checked="" type="checkbox"/> City	4:34	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	10
<input checked="" type="checkbox"/> Love on the Rocks	4:13	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	9
<input checked="" type="checkbox"/> Many the Miles	5:11	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	9
<input checked="" type="checkbox"/> Come Round Soon	3:33	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	8
<input checked="" type="checkbox"/> Morningside	3:58	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	7
<input checked="" type="checkbox"/> Fairytale	3:15	Sara Bareilles	Little Voice	Folk	☆☆☆☆☆	5
<input checked="" type="checkbox"/> 3rd Eye Vision	5:00	Mishka	Above the Bones	Reggae		4
<input checked="" type="checkbox"/> Wild Child	3:48	Enya	A Day Without Rain	New Age		3
<input checked="" type="checkbox"/> Only Time	3:38	Enya	A Day Without Rain	New Age		3
<input checked="" type="checkbox"/> Feeling Good	2:55	Nina Simone	The Jazz Plays Nina...	Jazz		3
<input checked="" type="checkbox"/> Awesome God	3:11	Youthful Praise	Awesome God	Christian ...		3

First, there's the fact that I added a song to iTunes. That indicates minimally that I was interested enough in the song to do so. Then there is the Play Count information. In the screen shot above, I've listened to Sara Bareilles' 'Between the Lines' 16 times. That suggests that I like that song. If I have a song in my library for awhile and only listened to it once, that might indicate that I don't like the song.

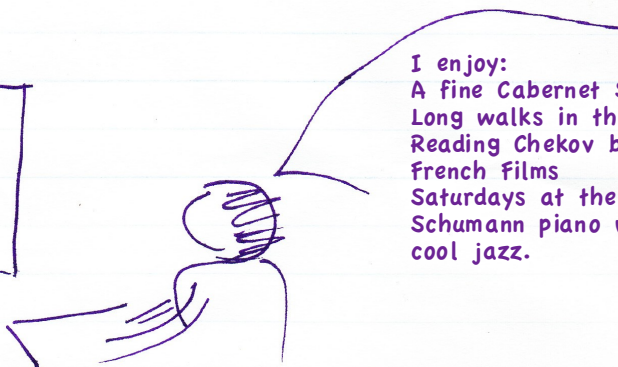
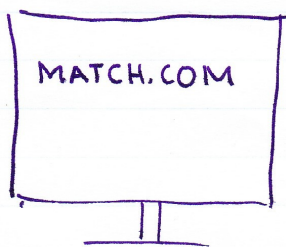
Which is more accurate: Explicit or Implicit ratings?

Do you think having a user explicitly give a rating to an item is more accurate?

Or do you think watching what a user buys or does (for ex., the play count) is a more accurate judge on what an individual likes?

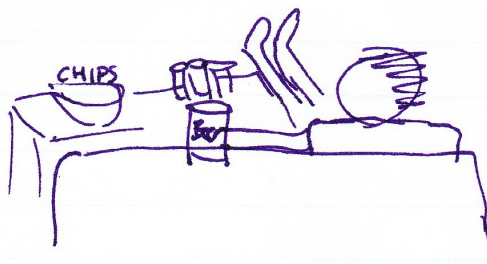
Which is more accurate: Explicit or Implicit ratings?

Explicit:



I enjoy:
A fine Cabernet Sauvignon
Long walks in the woods
Reading Chekov by the fire
French films
Saturdays at the Art Museum
Schumann piano works and
cool jazz.

Implicit:



What are the problems with explicit ratings?

Problem 1: People are lazy and don't rate items.

First, users will typically not bother to rate items. I have imagine most of you have bought a substantial amount of stuff on Amazon. I know I have. In the last month I bought a microHelicopter, a 1TB hard drive, a USB-SATA converter, a bunch of vitamins, two Kindle books (Murder City: Ciudad Juarez and the Global Economy's New Killing Fields and Fool Moon: The Dresden Files Book 2) and and the physical books No Place to Hide, Dr. Weil's 8 Weeks to Optimum Health, Anticancer: A new way of life, and Rework. That's twelve items. How many have I rated? Zero. I imagine most of you are the same. You don't rate the items you buy.

I have a gimp knee. I like hiking in the mountains and as a result own a number of trekking poles including some cheap ones I bought on Amazon that have taken a lot of abuse. Last year I flew to Austin for the 3 day Austin City Limits music festival. I aggravated my knee injury dashing from one flight to another and ended up going to REI to buy a somewhat pricey REI branded trekking pole. It broke in less than a day of walking on flat grass at a city park. Here I own \$10 poles that don't break during constant use of hiking around in the Rockies and this pricey model broke on flat ground. At the time of the festival, as I was fuming, I planned to rate and write a review of the pole on the REI site. Did I? No, I am too lazy. So even in this extreme case I didn't rate the item. I think there are a lot of lazy people like me. People in general are too lazy or unmotivated to rate products.

Problem 2: People may lie or give only partial information.

Let's say someone gets over that initial laziness and actually rates a product. That person may lie. This is illustrated in the drawing on the previous page. They can lie directly—giving inaccurate ratings or lie via omission—providing only partial information. Ben goes on a first date with Ann to see the 2010 Cannes Film Festival Winner, a Thai film, *Uncle Boonmee Who Can Recall His Past Lives*. They go with Ben's friend Dan and Dan's friend Clara. Ben thinks it was the worst film he ever saw. All the others absolutely loved it and gushed about it afterwards at the restaurant. It would not be surprising if Ben upped his rating of the film on online rating sites that his friends might see or just not rate the film.

Problem 3: People don't update their ratings.

Suppose I am motivated by writing this chapter to rate my Amazon purchases. That 1TB hard drive works well—it's very speedy and also very quiet. I rate it five stars. That microHelicopter is great. It is easy to fly and great fun and it survived multiple crashes. I rate it five stars. A month goes by. The hard drive dies and as a result I lose all my downloaded movies and music—a major bummer. The microHelicopter suddenly stops working—it looks like the motor is fried. Now I think both products suck. Chances are pretty good that I will not go to Amazon and update my ratings (laziness again). People still think I would rate both 5 stars.

Consider Mary, a college student. For some reason, she loves giving Amazon ratings. Ten years ago she rated her favorite music albums with five stars: *Giggling and Laughing: Silly Songs for Kids*, and *Sesame Songs: Sing Yourself Silly!* Her most recent ratings included 5 stars for *Wolfgang Amadeus Phoenix* and *The Twilight Saga: Eclipse Soundtrack*. Based on these recent ratings she ends up being the closest neighbor to another college student Jen. It would be odd to recommend *Giggling and Laughing: Silly Songs for Kids* to Jen. This is a slightly different type of update problem than the one above, but a problem none-the-less.

Problems with Implicit Ratings

Before turning the page, list what you think are the problems with implicit ratings.

A few pages ago I gave a list of items I bought at Amazon in the last month. It turns out I bought two of those items for other people. I bought the anticancer book for my cousin and the Rework book for my son. To see why this is a problem, let me come up with a more compelling example by going further back in my purchase history. I bought some kettlebells and the book *Enter the Kettlebell! Secret of the Soviet Supermen* as a gift for my son and a Plush Chase Border Collie stuffed animal for my wife because our 14 year old border collie died. Using purchase history as an implicit rating of what a person likes, might lead you to believe that people who like kettlebells, like stuffed animals, like microHelicopters, books on anticancer, and the Dresden File books. Amazon's purchase history can't distinguish between purchases for myself and purchases I make as gifts. Stephen Baker describes a related example:

Figuring out that a certain white blouse is business attire for a female baby boomer is merely step one for the computer. The more important task is to build a profile of the shopper who buys that blouse. Let's say it's my wife. She goes to Macy's and buys four or five items for herself. Underwear, pants, a couple of blouses, maybe a belt. All of the items fit that boomer profile. She's coming into focus. Then, on the way out she remembers to buy a birthday present for our 16-year-old niece. Last time we say her, this girl was wearing black clothing with a lot of writing on it, most of it angry. She told us she was a goth. So my wife goes into an "alternative" section and—what the hell—picks up one of those dog collars bristling with sharp spikes.

Baker 2008.60-61.

If we are attempting to build a profile of a person—what a particular person likes—this dog collar purchase is problematic.

Finally consider a couple sharing a Netflix account. He likes action flicks with lots of explosions and helicopters, she likes intellectual movies and romantic comedies. If we just look at rental history, we build an odd profile of someone liking two very different things.

Recall that I said my purchase of the book *Anticancer: A New Way of Life* was as a gift to my cousin. If we mine my purchase history a bit more we would see that I bought this book before. In fact in the last year, I purchased multiple copies of three books. One can imagine that I am making these multiple purchases not because I am losing the books, or that I am losing my mind and forgetting that I read the books. The most rational reason, is that I liked the books so much I am in a sense recommending these books to others by giving them as gifts. So we can gain a substantial amount of information from a person's purchase history.

So what can we use as implicit data (before turning the page, come up with a list of possibilities)

Implicit data

Web pages: clicking on the link to the page
 time spent looking at page
 repeated visits
 recommending or referring

Music players: what the person plays.
 skipping tunes
 number of times tune played.

Keep in mind that the algorithms described in chapter 2 can be used regardless of whether the data is explicit or implicit.

The problems of success

You have a successful streaming music service with a built in recommendation system. What could possibly go wrong?

Suppose you have one million users. Everytime you want to make a recommendation for someone you need to calculate one million distances (comparing that person to the one million other people). If we are making multiple recommendations per second, the number of calculations get extreme. Unless you throw a lot of iron at the problem the system will get slow. To say this in a more formal way, latency can be a major drawback of neighbor-based recommendation systems. Fortunately, there is a solution.

User-based filtering.

So far we have been doing user-based collaborative filtering. We are comparing a user with every other user to find the closest matches. There are two main problems with this approach:

1. **Scalability.** As we have just discussed, the computation increases as the number of users increases. User-based methods work fine for thousands of users, but scalability gets to be a problem when we have a million users.
2. **Sparcity.** Most recommendation systems have many users and many products but the average user rates a small fraction of the total products. For example, Amazon carries millions of books but the average user rates just a handful of books. Because of this the algorithms we covered in chapter 2 may not find any nearest neighbors.

Because of these two issues it might be better to do what is called item-based filtering.

Item-based filtering.

Suppose I have an algorithm that identifies products that are most similar to each other. For example, such an algorithm might find that Phoenix's album *Wolfgang Amadeus Phoenix* is similar to Passion Pit's album, *Manners*. If a user rates *Wolfgang Amadeus Phoenix* highly we could recommend the similar album *Manners*. Note that this is different than what we did for user-based filtering. In user-based filtering we had a user, found the most similar person (or users) to that user and used the ratings of that similar person to make recommendations. In item-based filtering, ahead of time we find the most similar items, and combine that with a user's rating of items to generate a recommendation.

Can you give me an example?

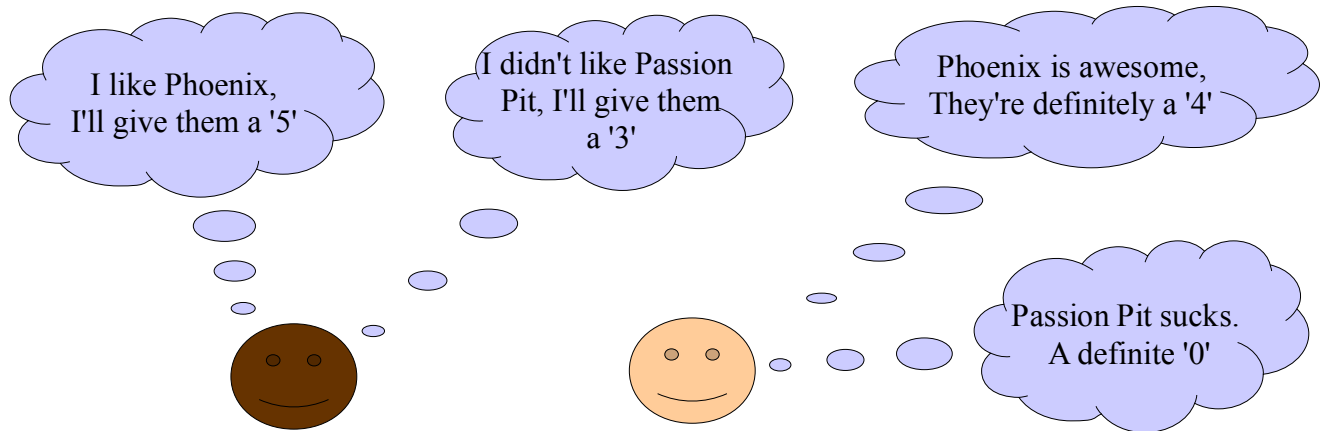
Suppose our streaming music site has m users and n bands, where the users rate bands. This is shown in the following table. As before the rows represent the users and the columns represent bands.

	USERS	...	Phoenix	...	Passion Pit	...	n
1	Marjorie Aaronson		5				
2	Jasmine Abbey				4		
3	Arturo Alvarez		1		2		
	...						
	...						
u	Cecilia De La Cueva		5		5		
	...						
	Lyle Kent		4				
	...						
m	Jordyn Zamora		4		5		

We would like to compute the similarity of Phoenix to Passion Pit. To do this we only use users who rated both bands as indicated by the blue squares. If we were doing user-based filtering we would determine the similarity between rows. For item-based filtering we are determining the similarity between columns—in this case between the Phoenix and Passion Pit columns.

Adjusted Cosine Similarity.

To compute the similarity between items we will use Cosine Similarity which I introduced in chapter 2. We already talked about grade inflation where a user gives higher ratings than expected.



To compensate for this grade inflation we will subtract the user's average rating from each rating. This gives us the adjusted cosine similarity formula shown here:

$$s(i, j) = \frac{\sum_{u \in U} (R_{u,i} - \bar{R}_u)(R_{u,j} - \bar{R}_u)}{\sqrt{\sum_{u \in U} (R_{u,i} - \bar{R}_u)^2} \sqrt{\sum_{u \in U} (R_{u,j} - \bar{R}_u)^2}}$$

This formula is from a seminal article in collaborative filtering: “Item-based collaborative filtering recommendation algorithms” by Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl (http://www.grouplens.org/papers/pdf/www10_sarwar.pdf)

So using the example above (reprinted here):

		Average Rating	Phoenix	...	Passion Pit	...	n
1	Marjorie Aaronson		5				
2	Jasmine Abbey				4		
3	Arturo Alvarez	3	1		2		
	...						
	...						
u	Cecilia De La Cueva	3.5	5		5		
	...						
	Lyle Kent		4				
	...						
m	Jordyn Zamora	4	4		5		

$$s(i, j) = \frac{(1-3)(2-3) + (5-3.5)(5-3.5) + (4-4)(5-4)}{\sqrt{(1-3)^2 + (5-3.5)^2 + (4-4)^2} \sqrt{(2-3)^2 + (5-3.5)^2 + (5-4)^2}}$$

$$s(i, j) = \frac{2 + 2.25 + 0}{\sqrt{6.25} \sqrt{4.25}} = \frac{4.25}{(2.5)(2.06)} = \frac{4.25}{5.15} = 0.825$$

Slope One

Another popular algorithm for item-based collaborative filtering is Slope One. A major advantage of Slope One is that it is simple and hence easy to implement. Slope One was introduced in the paper “Slope One Predictors for Online Rating-Based Collaborative Filtering” by Daniel Lemire and Anna Machlachlan (<http://www.daniel-lemire.com/fr/abstracts/SDM2005.html>). This is an awesome paper and well worth the read.

Here's the basic idea in a minimalist nutshell. Suppose Amy gave a rating of 3 to Lady Gaga and a rating of 4 to Phoenix. Ben gave a rating of 4 to Lady Gaga. We'd like to predict how Ben would rate Phoenix. In table form the problem might look like this:

	Lady Gaga	Phoenix
Amy	3	4
Ben	4	?

To guess what Ben might rate Phoenix we could reason as follows. Amy rated Phoenix one whole point better than Lady Gaga. We can predict then that Ben would rate Phoenix one point higher so we will predict that Ben will give Phoenix a '5'.

There are actually several Slope One algorithms. I will present the Weighted Slope One algorithm. Remember that a major advantage is that the approach is simple. What I present may look complex, but bear with me and things should become clear. You can consider Slope One to be in two parts. First, ahead of time (in batch mode, in the middle of the night or whenever) we are going to compute what is called the **deviation** between every pair of items. In the simple example above, this step will determine that Phoenix is rated 1 better than Lady Gaga. Now we have a nice database of item deviations. In the second phase we actually make predictions. A user comes along, Ben for example. He has never heard the band Phoenix and we want to predict how he would rate it. Using all the bands he did rate along with our database of deviations we are going to make a prediction. That's the broad brush picture



Part 1: Computing deviation

Let's make our previous example way more complex by adding one user and one band:

	Dr. Dog	Lady Gaga	Phoenix
Amy	4	3	4
Ben	5	2	?
Clara	?	3.5	4

The first step is to compute the deviations. The average deviation of an item i with respect to item j is:

$$dev_{j,i} = \sum_{u \in S_{j,i}(X)} \frac{u_j - u_i}{card(S_{j,i}(X))}$$

where $card(S)$ is how many elements are in S and X is the entire set of all ratings. So $card(S_{j,i}(X))$ is the number of people who have rated both j and i . Let's consider the deviation of Lady Gaga with respect to Dr. Dog. In this case, $card(S_{j,i}(X))$ is 2—there are 2 people (Amy and Ben) who rated both Dr. Dog and Lady Gaga. The $u_j - u_i$ numerator is (the rating for Dr. Dog) minus (the rating for Lady Gaga). So the deviation is:

$$dev = \frac{(4-3)}{2} + \frac{(5-2)}{2} = \frac{1}{2} + \frac{3}{2} = 2$$

Compute the rest of the values in this table:

	Dr. Dog	Lady Gaga	Phoenix
Dr. Dog	0		
Lady Gaga	2	0	
Phoenix			0

QUESTION:

Consider our streaming music site with one million users rating 200,000 artists. If we get a new user who rates 10 artists do we need to run the algorithm again to generate the deviations of all 200k x 200k pairs or is there an easier way?

ANSWERS

Phoenix with respect to Lady Gaga:

$$dev = \frac{3-4}{2} + \frac{3.5-4}{2} = -\frac{1}{2} + -\frac{.5}{2} = -.75$$

Phoenix with respect to Dr. Dog:

$$dev = \frac{0}{1} = 0$$

Compute the rest of the values in this table:

	Dr. Dog	Lady Gaga	Phoenix
Dr. Dog	0	-2	0
Lady Gaga	2	0	0.75
Phoenix	0	-0.75	0

QUESTION:

Consider our streaming music site with one million users rating 200,000 artists. If we get a new user who rates 10 artists do we need to run the algorithm again to generate the deviations of all 200k x 200k pairs or is there an easier way?

You don't need to run the algorithm on the entire dataset again. That's the beauty of this method. For a given pair of items we only need to keep track of the deviation and the total number of people rating both items.

For example, suppose I have a deviation of Dr. Dog with respect to Lady Gaga of 2 based on 9 people. I have a new person who rated Dr. Dog 1 and Lady Gaga 5 the updated deviation would be

$$((9 * 2) + 4) / 10 = 2.2$$

Okay, so now we have a big collection of deviations. How can we use that collection to make predictions. As I mentioned, we are using Weighted Slope One or wS1 for short. The formula is:

$$pwSI_{(u)j} = \frac{\sum_{i \in S(u)-\{j\}} (dev_{j,i} + u_i) c_{j,i}}{\sum_{i \in S(u)-\{j\}} c_{i,j}}$$

where $c_{j,i} = \text{card}(S_{j,i}(X))$

Let's say I am interested in answering the question: *How Ben will like Phoenix?*

Let's dissect the numerator. So, for every band that Ben has rated (except for Phoenix—the $\{j\}$ bit), we will look up the deviation of Phoenix to that band and we will add that to Ben's rating for that band. We multiply that by the cardinality of that pair. Let's step through this:

1. Ben has rated Dr. Dog and gave it a 5—that is the u .
2. The deviation of Phoenix with respect to Dr. Dog is 0—this is the $dev_{j,i}$.
3. $dev_{j,i} + u$ then is 5.
4. There was only one person that rated both Dr. Dog and Phoenix so $c_{j,i} = 1$
5. So $(dev_{j,i} + u)c_{j,i} = 5$
6. Ben has rated Lady Gaga and gave her a 2.
7. The deviation of Lady Gaga with respect to Phoenix is 0.75
8. $dev_{j,i} + u$ then is 2.75
9. Two people rated both Lady Gaga and Phoenix so $(dev_{j,i} + u)c_{j,i} = 5.5$
10. We sum up steps 5 and 9 to get 10.5 for the numerator

11. Dissecting the demoninator we get something like for every band that Ben has rated sum the cardinalities of those bands to Phoenix. So Ben has rated Dr. Dog and the cardinality of Dr. Dog and Phoenix (that is, the total number of people that rated both of them) is 1. Ben has rated Lady Gaga and her cardinality is also 2. So the demoninator is 3.

12. So our prediction of how well Ben will like Phoenix is 3.5.

Putting this into Python

I am going to extend the Python class developed in chapter 2. To save space I will not repeat the code for the recommender class here—just refer back to it (and remember that you can download the code at <http://guidetodatamining.com>). Recall that the data for that class was in the following format:

```
>>> users2 = {"Amy": {"Dr. Dog": 4, "Lady Gaga": 3, "Phoenix": 4},
              "Ben": {"Dr. Dog": 5, "Lady Gaga": 2},
              "Clara": {"Lady Gaga": 3.5, "Phoenix": 4}}
>>>
```

First computing the deviations.

Again, the formula for computing deviations is

$$dev_{j,i} = \sum_{u \in S_{j,i}(X)} \frac{u_j - u_i}{card(S_{j,i}(X))}$$

So the input to our computeDeviations function should be data in the format of users2 above. The output should be a representation of the following data:

	Dr. Dog	Lady Gaga	Phoenix
Dr. Dog	-	-2 (2)	0 (1)
Lady Gaga	2 (2)	-	0.75 (2)
Phoenix	0 (1)	-0.75 (2)	-

The number in the parentheses is the frequency (that is, the number of people that rated that pair of bands). So for each pair of bands we need to record both the deviation and the frequency.

The psuedoCode for our function could be

```
def computeDeviations(self):
    for each i in bands:
        for each j in bands:
            if i ≠ j:
                compute dev(j,i)
```

That psuedocode looks pretty nice but as you can see, there is a disconnect between the data format expected by the psuedo code and the format the data is really in (see users2 above as an example). As

code warriors we have two possibilities, either alter the format of the data, or revise the psuedocode. I am going to opt for the second approach. This revised psuedocode looks like

```
def computeDeviations(self):
    for each person in the data:
        get their ratings
        for each item & rating in that set of ratings:
            for each item2 & rating2 in that set of ratings:
                add the difference between the ratings to our computation
```

Let's construct the method step-by-step

Step 1:

```
def computeDeviations(self):
    #for each person in the data:
    #    get their ratings
    for ratings in self.data.values():
```

Python dictionaries (aka hash tables) are key value pairs. Self.data is a dictionary. The values method extracts just the values from the dictionary. Our data looks like

```
>>> users2 = {"Amy": {"Dr. Dog": 4, "Lady Gaga": 3, "Phoenix": 4},
              "Ben": {"Dr. Dog": 5, "Lady Gaga": 2},
              "Clara": {"Lady Gaga": 3.5, "Phoenix": 4}}
```

So the first time through the loop ratings = {"Dr. Dog": 4, "Lady Gaga": 3, "Phoenix": 4}

Step 2:

```
def computeDeviations(self):
    #for each person in the data:
    #    get their ratings
    for ratings in self.data.values():
        #for each item & rating in that set of ratings:
        for (item, rating) in ratings.items():
            self.frequencies.setdefault(item, {})
            self.deviations.setdefault(item, {})
```

In the recommender class init method I initialized self.frequencies and self.deviations to be dictionaries. The Python dictionary method setdefault takes 2 arguments: a key and an initialValue. This method does the following. If the key does not exist in the dictionary it is added to the dictionary with the value initialValue. Otherwise it returns the current value of the key.

Using the sample data above (users2) the first time through the loop in step one ratings is {"Dr. Dog": 4, "Lady Gaga": 3, "Phoenix": 4}. This next for loop iterates through that list.

Step 3:

```
def computeDeviations(self):
    #for each person in the data:
    #    get their ratings
    for ratings in self.data.values():
        #for each item & rating in that set of ratings:
        for (item, rating) in ratings.items():
            self.frequencies.setdefault(item, {})
            self.deviations.setdefault(item, {})
            #for each item2 & rating2 in that set of ratings:
            for (item2, rating2) in ratings.items():
                if item != item2:
                    #add the difference between the ratings to our computation
                    self.frequencies[item].setdefault(item2, 0)
                    self.deviations[item].setdefault(item2, 0.0)
                    self.frequencies[item][item2] += 1
                    self.deviations[item][item2] += rating - rating2
```

The code added in this step computes the difference between two ratings and adds that to the self.deviations running sum. Again, using the data:

```
{"Dr. Dog": 4, "Lady Gaga": 3, "Phoenix": 4}
```

when we are in the outer loop where item = “Dr Dog” and rating = 4 and in the inner loop where item2 = “Lady Gaga” and rating2 = 3 the last line of the code above adds 1 to self.deviations[“Dr. Dog”][“Lady Gaga”].

Step 4:

Finally, we need to iterate through self.deviations to divide each deviation by its associated frequency.

```
def computeDeviations(self):
    #for each person in the data:
    #    get their ratings
    for ratings in self.data.values():
        #for each item & rating in that set of ratings:
        for (item, rating) in ratings.items():
            self.frequencies.setdefault(item, {})
            self.deviations.setdefault(item, {})
            #for each item2 & rating2 in that set of ratings:
            for (item2, rating2) in ratings.items():
                if item != item2:
                    #add the difference between the ratings to our computation
                    self.frequencies[item].setdefault(item2, 0)
                    self.deviations[item].setdefault(item2, 0.0)
                    self.frequencies[item][item2] += 1
                    self.deviations[item][item2] += rating - rating2
    for (item, ratings) in self.deviations.items():
        for item2 in ratings:
            ratings[item2] /= self.frequencies[item][item2]
```

That's it! Even with comments we implemented

$$dev_{j,i} = \sum_{u \in S_{j,i}(X)} \frac{u_j - u_i}{card(S_{j,i}(X))}$$

in only 18 lines of code.

When I run this method on the data I have been using in this example:

```
>>> users2 = {"Amy": {"Dr. Dog": 4, "Lady Gaga": 3, "Phoenix": 4},
              "Ben": {"Dr. Dog": 5, "Lady Gaga": 2},
              "Clara": {"Lady Gaga": 3.5, "Phoenix": 4}}
```

I get

```
>>> r = recommender(users2)
>>> r.computeDeviations()
>>> r.deviations
{'Lady Gaga': {'Dr. Dog': -2.0, 'Phoenix': -0.75}, 'Dr. Dog': {'Lady Gaga': 2.0,
'Phoenix': 0.0}, 'Phoenix': {'Lady Gaga': 0.75, 'Dr. Dog': 0.0}}
```

which should look remarkably like what we computed by hand.

By the way, I need to give a shout-out to Bryan O-Sullivan and his blog *teideal glic deisbhéalach* which presented a Python implement of Slope One. The code presented here is based on his work.

Weighted Slope 1: the recommendation component

Now it is time to code the recommendation component:

$$pwSI_{(u)j} = \frac{\sum_{i \in S(u) - \{j\}} (dev_{j,i} + u_i) c_{j,i}}{\sum_{i \in S(u) - \{j\}} c_{i,j}}$$

The big question I have is can we beat the 18 line implementation of computeDeviations. First, let's parse that formula and put it into English and/or psuedo code. You try:

The formula in psuedo English:

Here's my version of the formula:

I would like to make recommendations for a particular user. I have that user's recommendations in the form

```
{"Dr. Dog": 5, "Lady Gaga": 2}
```

for every `userItem` and `userRating` in the user's recommendations:

- for every `diffItem` that the user didn't rate (`item2 ≠ item`):
 - add the deviation of `diffItem` with respect to `userItem` to the `userRating` of the `userItem`. Multiply that by the number of people that rated both `userItem` and `diffItem`.
 - Add that to the running sum for `diffItem`
 - Also keep a running sum for the number of people that rated `diffItem`.

Finally, for every `diffItem` that is in our results list, divide the total sum of that item by the total frequency of that item and return the results.

And here is my conversion of that to Python

```
def slopeOneRecommendations(self, userRatings):
    recommendations = {}
    frequencies = {}
    # for every item and rating in the user's recommendations
    for (userItem, userRating) in userRatings.items():
        #for every item in our dataset that the user didn't rate
        for (diffItem, diffRatings) in self.deviations.items():
            if diffItem not in userRatings and userItem in self.deviations[diffItem]:
                freq = self.frequencies[diffItem][userItem]
                recommendations.setdefault(diffItem, 0.0)
                frequencies.setdefault(diffItem, 0)
                # add to the running sum representing the numerator of the formula
                recommendations[diffItem] +=
                    (diffRatings[userItem] + userRating) * freq
                # keep a running sum of the frequency of diffitem
                frequencies[diffItem] += freq
    recommendations = [(k, v / frequencies[k])
                       for (k, v) in recommendations.items()]
    # finally sort and return
    recommendations.sort(key=lambda artistTuple: artistTuple[1], reverse = True)
    return recommendations
```

And here is a simple test of the complete Slope One implementation:

```
>>> r = recommender(users2)
>>> r.computeDeviations()
>>> g = users2['Ben']
>>> r.slopeOneRecommendations(g)
[['Phoenix', 3.5]]
>>>
```

This results matches what we calculated by hand. So the recommendation part of the algorithm weighs in at 18 lines. So in 36 lines of Python code we implemented the Slope One algorithm. With Python you can write pretty compact code.

MovieLens data set

Let's try out the Slope One recommender on a different dataset. The MovieLens dataset—collected by the GroupLens Research Project at the University of Minnesota—contains user ratings of movies. The data set is available for download at www.grouplens.org. The data set is available in three sizes; for the demo here I am using the smallest one which contains 100,000 ratings (1-5) from 943 users on 1,682 movies. I wrote a short function that will import this data into the recommender class.

Let's give it a try.

First, I will load the data into the Python recommender object:

```
>>> r = recommender(0)
>>> r.loadMovieLens('/Users/raz/Downloads/ml-data_0/')
102625
```

I will be using the info from User 1. Just to peruse the data, I will look at the top 50 items the user 1 rated:

```
>>> r.showUserTopItems('1', 50)
When Harry Met Sally... (1989) 5
Jean de Florette (1986) 5
Godfather, The (1972) 5
Big Night (1996) 5
Manon of the Spring (Manon des sources) (1986) 5
Sling Blade (1996) 5
Breaking the Waves (1996) 5
Terminator 2: Judgment Day (1991) 5
Searching for Bobby Fischer (1993) 5
```

```

Maya Lin: A Strong Clear Vision (1994) 5
Mighty Aphrodite (1995) 5
Bound (1996) 5
Full Monty, The (1997) 5
Chasing Amy (1997) 5
Ridicule (1996) 5
Nightmare Before Christmas, The (1993) 5
Three Colors: Red (1994) 5
Professional, The (1994) 5
Priest (1994)5
Terminator, The (1984) 5
Graduate, The (1967) 5
Dead Poets Society (1989) 5
Amadeus (1984) 5
...

```

Now I will do the first step of Slope One: computing the deviations:

```
>>> r.computeDeviations()
```

Finally, let's get recommendations for User 1:

```
>>> g = r.data['1']
```

```
>>> r.slopeOneRecommendations(g)
```

```

[('Entertaining Angels: The Dorothy Day Story (1996)', 6.375), ('Aiqing wansui (1994)',
5.8490566037735849), ('Boys, Les (1997)', 5.6449704142011834), ('Someone Else's America (1995)',
5.3913043478260869), ('Santa with Muscles (1996)', 5.3809523809523814), ('Great Day in Harlem, A
(1994)', 5.2758620689655169), ('Little City (1998)', 5.2363636363636363), ('Pather Panchali (1955)',
5.2096128170894529), ('Saint of Fort Washington, The (1993)', 5.2017543859649127), ('Some
Mother's Son (1996)', 5.154471544715447), ('Faust (1994)', 5.1274509803921573), ('Angel Baby
(1995)', 5.0216718266253872), ('Butcher Boy, The (1998)', 4.9661016949152543), ('Spanish Prisoner,
The (1997)', 4.9661016949152543), ...

```

Compare that to the recommendations we get from Pearson:

```
>>> r.recommend('1')
```

```

[('Everyone Says I Love You (1996)', 4.0), ('Eve's Bayou (1997)', 3.0), ('Washington Square
(1997)', 2.0), ('Sweet Hereafter, The (1997)', 2.0), ('Air Force One (1997)', 1.0)]

```

PROJECT

1. See how well the Slope One recommender recommends movies for you. Rate 10 movies or so (ones that are in the MovieLens dataset). Does the recommender recommend movies you might like?
2. Implement Adjusted Cosine Similarity. Compare its performance to Slope One.
3. (harder) I run out of memory (I have 8GB on my desktop) when I try to run this on the Book Crossing Dataset. Recall that there are 270,000 books that are rated. So we would need a $270,000 \times 270,000$ dictionary to store the deviations. That's roughly 73 billion dictionary entries. How sparse is this dictionary for the MovieLens dataset? Alter the code so we can handle larger datasets.