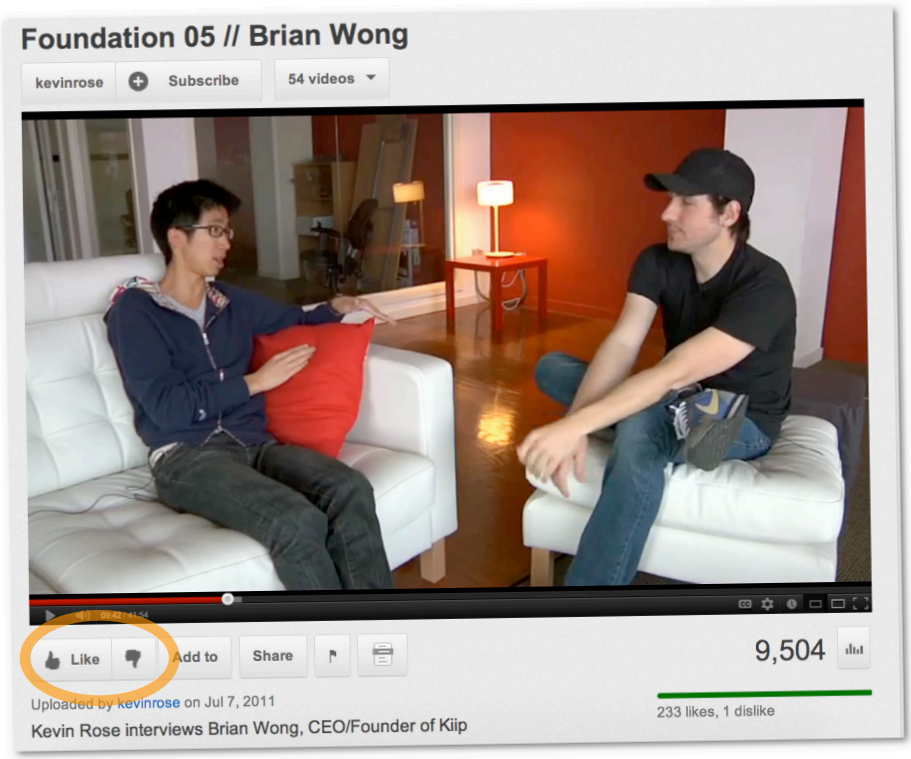
Implicit ratings and item based filtering

In chapter 2 we learned the basics of collaborative filtering and recommendation systems. The algorithms described in that chapter are general purpose and could be used with a variety of data. Users rated different items on a five or ten point scale and the algorithms found other users who had similar ratings. As was mentioned, there is some evidence to suggest users typically do not use this fine-grain distinction and instead tend to either give the top rating or the lowest one. This all-or-nothing rating strategy can sometimes lead to unusable results. In this chapter we will examine ways to fine tune collaborative filtering to produce more accurate recommendations in an efficient manner.

Explicit ratings

One way of distinguishing types of user preferences is whether they are explicit or implicit. Explicit ratings are when the user herself explicitly rates the item. One example of this is the thumbs up / thumbs down rating on sites such as Pandora and YouTube.



And Amazon's star system:

Most Recent Customer Reviews

★★★★★ **excellent translation, excellent sutra**

Very clear translation, minimal old untranslated Buddhist terms. This translation reminds me of a Cleary translation; true to the meaning, clear language, great flow. Read more

Published 29 days ago by M. Gonzales

★★★★☆ **The Long Sutra**

Gene Reeves did an excellent job in translating The Lotus Sutra from the Chinese into English. It is here, the entire work, not abridged.

Read more

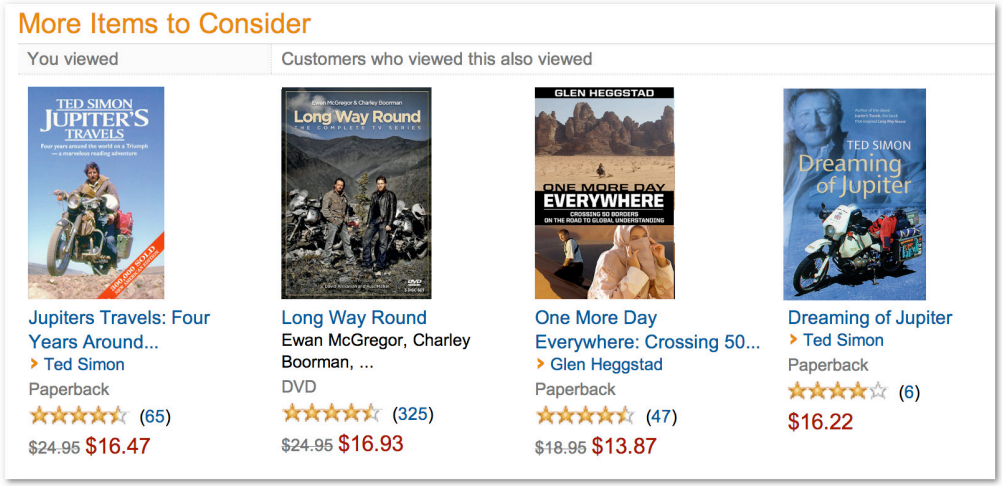
Published 8 months ago by Eric Maroney

Implicit Ratings

For implicit ratings, we don't ask users to give 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.

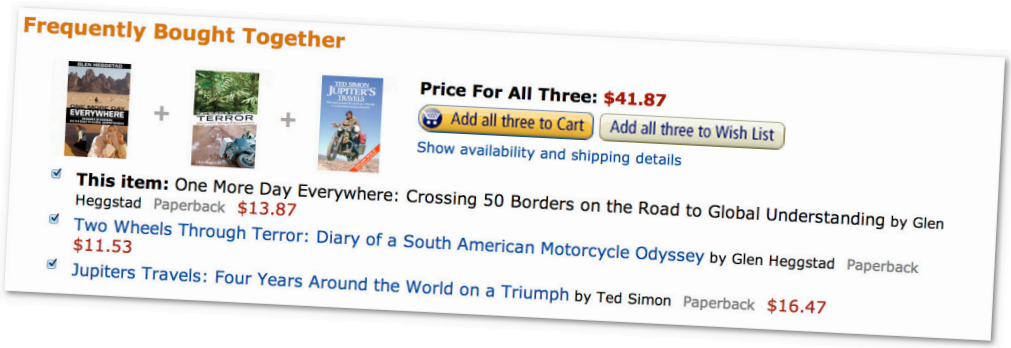
After observing what a user clicks on for a few weeks you can imagine that we could develop a reasonable profile of that user—she doesn't like sports but seems to like technology news. If the user clicks on the article "Fastest Way to Lose Weight Discovered by Professional Trainers" and the article "Slow and Steady: How to lose weight and keep it off" perhaps she wishes to lose weight. If she clicks on the iPhone ad, she perhaps has an interest in that product. (By the way, the term used when a user clicks on an ad is called 'click through'.)

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:



In this example, Amazon keeps track of what people click on. It knows, for example, that people who viewed the book *Jupiter's Travels: Four years around the world on a Triumph* also viewed the DVD *Long Way Round*, which chronicles the actor Ewan McGregor as he travels with his mate around the world on motorcycles. As can be seen in the Amazon screenshot above, this information is used to display the items in the section "Customers who viewed this also viewed."

Another implicit rating is what the customer actually buys. Amazon also keeps track of this information and uses it for their recommendations "Frequently Bought Together" and "Customers Who Viewed This Item Also Bought":



You would think that "Frequently Bought Together" would lead to some unusual recommendations but this works surprisingly well.

Imagine what information a program can acquire by monitoring your behavior in iTunes.

Name	Time	Artist	Plays
Anchor	3:24	Zee Avi	52
My Companjera	3:22	Gogol Bordello	27
Wake Up Everybody	4:25	John Legend & the...	17
Milestone Moon	3:40	Zee Avi	17
...			

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 image above, I've listened to Zee Avi's “Anchor” 52 times. That suggests that I like that song (and in fact I do). 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.

brain calisthenics

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 example, the play count) is a more accurate judge of what an individual likes?

Jim

Explicit Rating:
match.com bio:

I am a vegan. I enjoy a fine Cabernet Sauvignon, long walks in the woods, reading Chekov by the fire, French Films, Saturdays at the art museum, and Schumann piano works.

Implicit Ratings:

what we found in Jim's pocket

Receipts for:
12 pack of Pabst Blue Ribbon beer, Whataburger, Ben and Jerry's ice cream, pizza & donuts
DVD rental receipts: Marvel's The Avengers, Resident Evil: Retribution, Ong Bak 3

Problems with explicit ratings

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

First, users will typically not bother to rate items. I 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 *Ready Player One*) 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.

my slightly bent REI pole

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 a few pages back. 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.

brain calisthenics

What do you think are the problems with implicit ratings?

(hint: think about the purchases you made on Amazon)

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 book *Ready Player One*. Amazon's purchase history can't distinguish between purchases for myself and purchases I make as gifts. Stephen Baker describes a related example:

Baker 2008.60-61.

> 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 saw 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.

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.

brain calisthenics

What can we use as implicit data when we are observing a person's behavior at a computer? Before turning the page come up with a list of possibilities

Implicit Data:

Web pages:	clicking on the link to a page
	time spent looking at a page
	repeated visits
	referring a page to others
	what a person watches on Hulu
Music players:	what the person plays
	skipping tunes
	number of times a tune is played

This just scratches the surface!

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. Every time you want to make a recommendation for someone you need to calculate one million distances (comparing that person to the 999,999 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.

Lots of iron:
a server farm

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. **Sparsity**. 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	Tamera Young		5				
2	Jasmine Abbey				4		
3	Arturo Alvarez		1		2		
...	...						
u	Cecilia De La Cueva		5		5		
...	...						
m-1	Jessica Nguyen		4		5		
m	Jordyn Zamora		4				

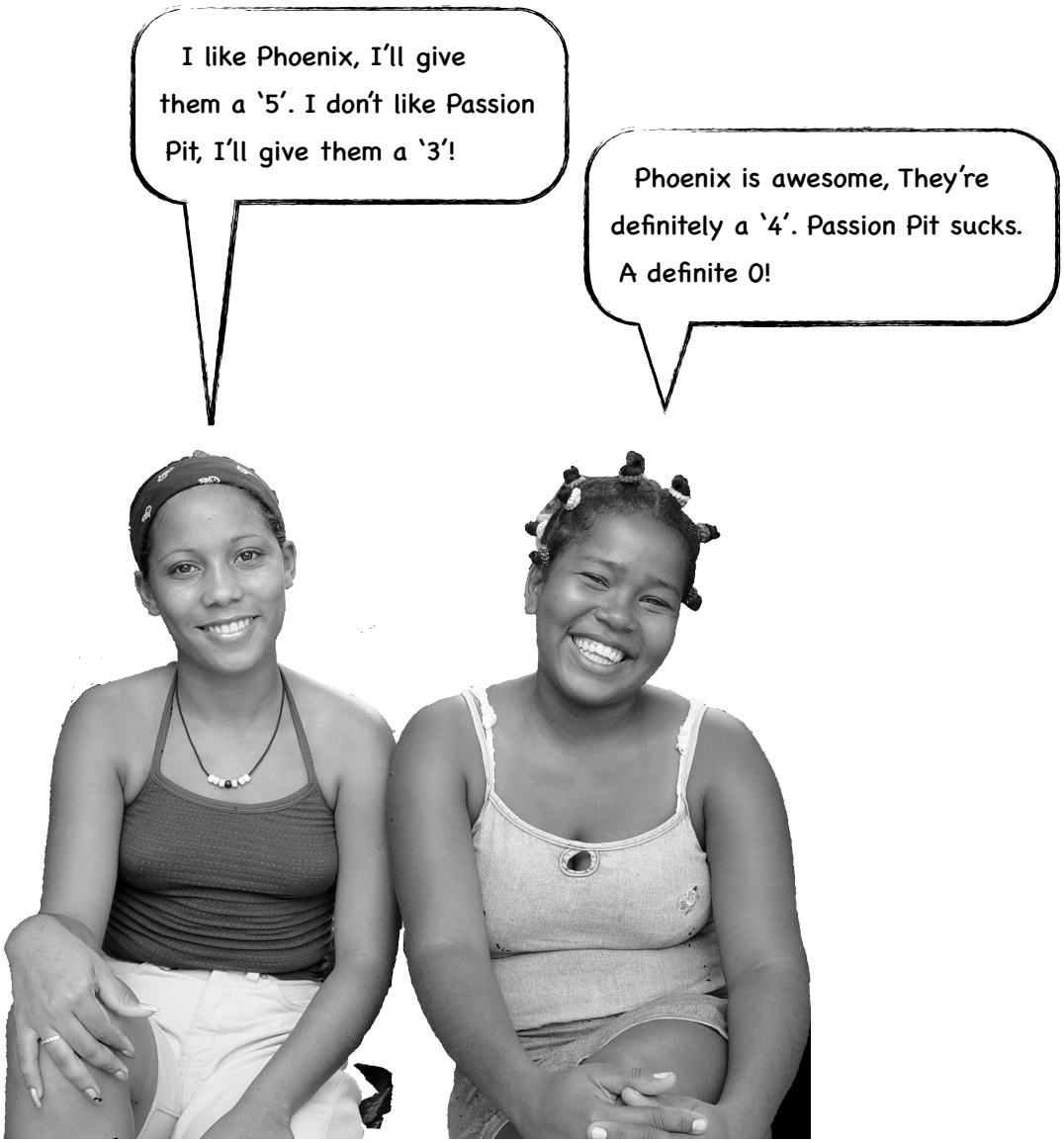
We would like to compute the similarity of Phoenix to Passion Pit. To do this we only use users who rated both bands as indicated by the blue squares. If we were doing user-based filtering we would determine the similarity between rows. For item-based filtering we are determining the similarity between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

User-based filtering is also called memory based collaborative filtering. Why? Because we need to store all the ratings in order to make recommendations.

Item-based filtering is also called model-based collaborative filtering. Why? Because we don't need to store all the ratings. We build a model representing how close every item is to every other item!

Adjusted Cosine Similarity.

To compute the similarity between items we will use Cosine Similarity which was introduced in chapter 2. We also 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 on the following page.



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

U is the set of all users who rated both items i and j!

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)

$$\left(R_{u,i} - \overline{R}_u\right)$$

means the rating *R* user *u* gives to item *i* minus the average rating that user gave for all items she rated. This gives us the normalized rating. In the formula above for $s(i,j)$ we are finding the similarity between items *i* and *j*. The numerator says that for every user who rated both items multiply the normalized rating of those two items and sum the results. In the denominator we sum the squares of all the normalized ratings for item *i* and then take the square root of that result. We do the same for item *j*. And then we multiply those two together.

To illustrate adjusted cosine similarity we will use the following data where five students rated five musical artists.

Users	average rating	Kacey Musgraves	Imagine Dragons	Daft Punk	Lorde	Fall Out Boy
David			3	5	4	1
Matt			3	4	4	1
Ben		4	3		3	1
Chris		4	4	4	3	1
Torri		5	4	5		3

The first thing to do is to compute each user's average rating. That is easy! Go ahead and fill that in.

Users	average rating	Kacey Musgraves	Imagine Dragons	Daft Punk	Lorde	Fall Out Boy
David	3.25		3	5	4	1
Matt	3.0		3	4	4	1
Ben	2.75	4	3		3	1
Chris	3.2	4	4	4	3	1
Tori	4.25	5	4	5		3

Now for each pair of musical artists we are going to compute their similarity. Let's start with Kacey Musgraves and Imagine Dragons. In the above table, I have circled the cases where a user rated both bands. So the adjusted cosine similarity formula is

$$s(Musgraves, Dragons) = \frac{\sum_{u \in U}(R_{u,Musgraves} - \overline{R}_u)(R_{u,Dragons} - \overline{R}_u)}{\sqrt{\sum_{u \in U}(R_{u,Musgraves} - \overline{R}_u)^2}\sqrt{\sum_{u \in U}(R_{u,Dragons} - \overline{R}_u)^2}}$$

Ben's ratings

Chris's ratings

Tori's ratings

$$= \frac{(4-2.75)(3-2.75)+(4-3.2)(4-3.2)+(5-4.25)(4-4.25)}{\sqrt{(4-2.75)^2+(4-3.2)^2+(5-4.25)^2}\sqrt{(3-2.75)^2+(4-3.2)^2+(4-4.25)^2}}$$

$$= \frac{0.7650}{\sqrt{2.765}\sqrt{0.765}} = \frac{0.7650}{(1.6628)(0.8746)} = \frac{0.7650}{1.4543} = 0.5260$$

So the similarity between Kacey Musgraves and Imagine Dragons is 0.5260. I have computed some of the other similarities and entered them in this table:

	Fall Out Boy	Lorde	Daft Punk	Imagine Dragons
Kacey Musgraves	-0.9549		1.0000	0.5260
Imagine Dragons	-0.3378		0.0075	
Daft Punk	-0.9570			
Lorde	-0.6934			
Fall Out Boy				

sharpen your pencil

Compute the rest of the values in the table above!

sharpen your pencil - solution

	Fall Out Boy	Lorde	Daft Punk	Imagine Dragons
Kacey Musgraves	-0.9549	0.3210	1.0000	0.5260
Imagine Dragons	-0.3378	-0.2525	0.0075	
Daft Punk	-0.9570	0.7841		
Lorde	-0.6934			

To compute these values I wrote a small Python script:

```
def computeSimilarity(band1, band2, userRatings):
   averages = {}
   for (key, ratings) in userRatings.items():
      averages[key] = (float(sum(ratings.values()))
                       / len(ratings.values()))

   num = 0  # numerator
   dem1 = 0 # first half of denominator
   dem2 = 0
   for (user, ratings) in userRatings.items():
      if band1 in ratings and band2 in ratings:
         avg = averages[user]
         num += (ratings[band1] - avg) * (ratings[band2] - avg)
         dem1 += (ratings[band1] - avg)**2
         dem2 += (ratings[band2] - avg)**2
   return num / (sqrt(dem1) * sqrt(dem2))
```

The format for the userRatings is shown on the following page!

sharpen your pencil - solution cont'd

```
users3 = {"David": {"Imagine Dragons": 3, "Daft Punk": 5,
                    "Lorde": 4, "Fall Out Boy": 1},
          "Matt":  {"Imagine Dragons": 3, "Daft Punk": 4,
                    "Lorde": 4, "Fall Out Boy": 1},
          "Ben":   {"Kacey Musgraves": 4, "Imagine Dragons": 3,
                    "Lorde": 3, "Fall Out Boy": 1},
          "Chris": {"Kacey Musgraves": 4, "Imagine Dragons": 4,
                    "Daft Punk": 4, "Lorde": 3, "Fall Out Boy": 1},
          "Tori":  {"Kacey Musgraves": 5, "Imagine Dragons": 4,
                    "Daft Punk": 5, "Fall Out Boy": 3}}
```

	Fall Out Boy	Lorde	Daft Punk	Imagine Dragons
Kacey Musgraves	-0.9549	0.3210	1.0000	0.5260
Imagine Dragons	-0.3378	-0.253	0.0075	
Daft Punk	-0.9570	0.7841		
Lorde	-0.6934			

Now that we have this nice matrix of similarity values, it would be dreamy if we could use it to make predictions.! (I wonder how well David will like Kacey Musgraves?)

$$p(u,i) = \frac{\sum_{N \in similarTo(i)} (S_{i,N} \times R_{u,N})}{\sum_{N \in similarTo(i)} (|S_{i,N}|)}$$

English, please!

Okay! p(u,i) means we are going to predict the rating user u will give item i.

so, P(David, Kacey Musgraves) means our prediction for the rating David (the u in the equation) will give Kacey Musgraves (the i in the equation)

N is each of the items that person u rated that are similar to item i. By 'similar' I mean that there is a similarity score between N and i in our matrix!

$S_{i,N}$ is the similarity between i and N (from the similarity matrix)

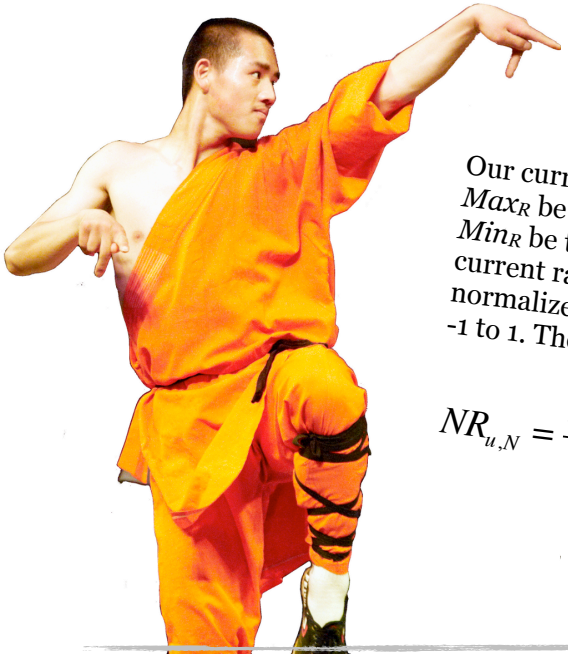
$R_{u,N}$ is the rating user u gave item N

$$p(u,i) = \frac{\sum_{N \in similarTo(i)} (S_{i,N} \times R_{u,N})}{\sum_{N \in similarTo(i)} (|S_{i,N}|)}$$

$R_{u,N}$ is We are trying to predict how well user u will like item i (what rating user u will give item i)

For this to work best, $R_{N,i}$ should be a value in the range -1 to 1.

Our ratings are in the range 1 to 5. So we will need some numeric Kung Fu to convert our ratings to the -1 to 1 scale.



Our current music ratings range from 1 to 5. Let Max_R be the maximum rating (5 in our case) and Min_R be the minimum rating (1). $R_{u,N}$ is the current rating user u gave item N. $NR_{u,N}$ is the normalized rating (the new rating on the scale of -1 to 1. The equation to normalize the rating is

$$NR_{u,N} = \frac{2(R_{u,N} - Min_R) - (Max_R - Min_R)}{(Max_R - Min_R)}$$

The equation to denormalize (go from the normalized rating to one in our original scale of 1-5 is:

$$R_{u,N} = \frac{1}{2}((NR_{u,N} + 1) \times (Max_R - Min_R)) + Min_R$$

Let's say someone rated Fall Out Boy a 2. Our normalized rating would be ...

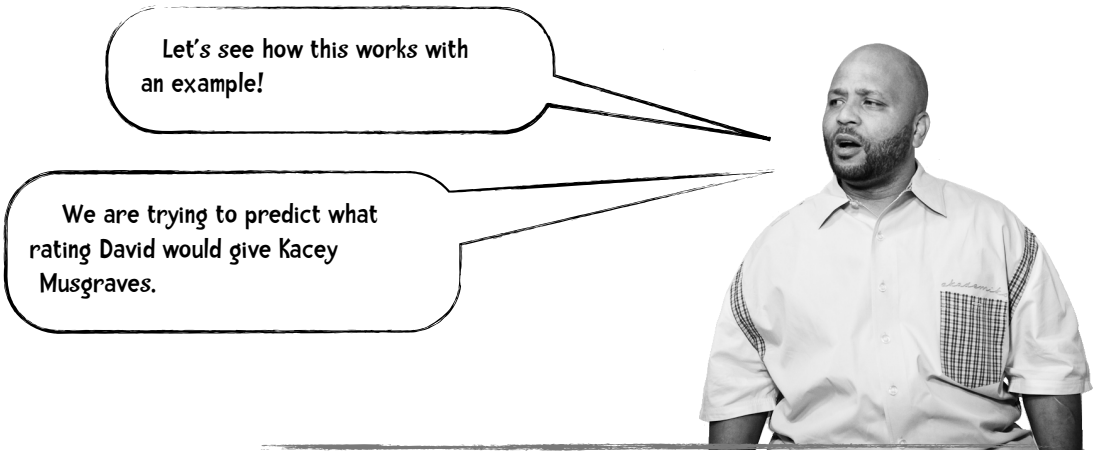
$$NR_{u,N} = \frac{2(R_{u,N} - Min_R) - (Max_R - Min_R)}{(Max_R - Min_R)} = \frac{2(2-1) - (5-1)}{(5-1)} = \frac{-2}{4} = -0.5$$

and to go the other way ...

$$R_{u,N} = \frac{1}{2}((NR_{u,N} + 1) \times (Max_R - Min_R)) + Min_R$$

$$= \frac{1}{2}((-0.5 + 1) \times 4) + 1 = \frac{1}{2}(2) + 1 = 1 + 1 = 2$$

Okay. We now have that numeric Kung Fu under our belt!



The first thing we are going to do is normalize David's ratings:

David's Ratings

Artist	R	NR
Imagine Dragons	3	0
Daft Punk	5	1
Lorde	4	0.5
Fall Out Boy	1	-1

We will learn more about normalization in the next chapter!

Similarity Matrix

	Fall Out Boy	Lorde	Daft Punk	Imagine Dragons
Kacey Musgraves	-0.9549	0.3210	1.0000	0.5260
Imagine Dragons	-0.3378	-0.2525	0.0075	
Daft Punk	-0.9570	0.7841		
Lorde	-0.6934			

David rated Imagine Dragons, Daft Punk, Lorde, and Fall Out Boy so we will use those in our calculations to determine how well he will like Kacey Musgraves.

And we will be using the normalized ratings!

$$p(u,i) = \frac{\sum_{N \in similarTo(i)} (S_{i,N} \times NR_{u,N})}{\sum_{N \in similarTo(i)} (|S_{i,N}|)} =$$

Imagine Dragons Daft Punk Lorde Fall Out Boy

$$\frac{(.5260 \times 0) + (1.00 \times 1) + (.321 \times 0.5) + (-.955 \times -1)}{0.5260 + 1.000 + 0.321 + 0.955}$$

$$= \frac{0+1+0.1605+0.955}{2.802} = \frac{2.1105}{2.802} = 0.753$$

So we predict that David will rate Kacey Musgraves a 0.753 on a scale of -1 to 1. To get back to our scale of 1 to 5 we need to denormalize:

$$R_{u,N} = \frac{1}{2}((NR_{u,N}+1)\times(Max_R - Min_R)) + Min_R$$

$$= \frac{1}{2}((0.753+1)\times 4)+1 = \frac{1}{2}(7.012)+1 = 3.506+1 = 4.506$$

So we predict that David will rate Kacey Musgraves a 4.506!

Adjusted Cosine Similiarity is a Model-Based Colloborative Filtering Method. As mentioned a few pages back, one advantage of these methods compared to memory-based ones is that they scale better. For large data sets, model-based methods tend to be fast and require less memory.

Often people use rating scales differently. I may rate artists I am not keen on a '3' and artists I like a '4'. You may rate artists you dislike a '1' and artists you like a '5'. Adjusted Cosine Similarity handles this problem by subtracting the corresponding user's average rating from each rating.

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 PSY and a rating of 4 to Whitney Houston. Ben gave a rating of 4 to PSY. We'd like to predict how Ben would rate Whitney Houston. In table form the problem might look like this:

	PSY	Whitney Houston
Amy	3	4
Ben	4	?

To guess what Ben might rate Whitney Houston we could reason as follows. Amy rated Whitney one whole point better than PSY. We can predict then than Ben would rate Whitney one point higher so we will predict that Ben will give her 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 Whitney is rated 1 better than PSY. 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 Whitney Houston and we want to predict how he would rate her. Using all the bands he did rate along with our database of deviations we are going to make a prediction.

The Broad Brush Picture

Part 1 (done ahead of time)
Compute deviations between every pair of items

Part 2
Use deviations to make predictions

Part 1: Computing deviation

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

	Taylor Swift	PSY	Whitney Houston
Amy	4	3	4
Ben	5	2	?
Clara	?	3.5	4
Daisy	5	?	3

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

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

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 PSY with respect to Taylor Swift. In this case, $card(S_{j,i}(X))$ is 2—there are 2 people (Amy and Ben) who rated both Taylor Swift and PSY. The $u_j - u_i$ numerator is (that user's rating for Taylor Swift) minus (that user's rating for PSY). So the deviation is:

$$dev_{swift,psy} = \frac{(4-3)}{2} + \frac{(5-2)}{2} = \frac{1}{2} + \frac{3}{2} = 2$$

So the deviation from PSY to Taylor Swift is 2 meaning that on average users rated Taylor Swift 2 better than PSY. What is the deviation from Taylor Swift to PSY?

$$dev_{psy,swift} = \frac{(3-4)}{2} + \frac{(2-5)}{2} = -\frac{1}{2} + -\frac{3}{2} = -2$$

sharpen your pencil

Compute the rest of the values in this table:

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2	
PSY	-2	0	
Whitney Houston			0

sharpen your pencil - solution

Compute the rest of the values in this table:

Taylor Swift with respect to Whitney Houston:

$$dev_{swift,houston} = \frac{(4-4)}{2} + \frac{(5-3)}{2} = \frac{0}{2} + \frac{2}{2} = 1$$

PSY with respect to Whitney Houston:

$$dev_{psy,houston} = \frac{(3-4)}{2} + \frac{(3.5-4)}{2} = \frac{-1}{2} + \frac{-.5}{2} = -.75$$

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2	1
PSY	-2	0	-0.75
Whitney Houston	-1	0.75	0

brain calisthenics

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?

(answer on next page)

brain calisthenics

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 Taylor Swift with respect to PSY of 2 based on 9 people. I have a new person who rated Taylor Swift 5 and PSY 1 the updated deviation would be

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

Part 2: Making predictions with Weighted Slope One

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 P^{wS1} --for Weighted Slope One Prediction. The formula is:

$$P^{wS1}(u)_j = \frac{\sum_{i \in S(u)-\{j\}} (dev_{j,i} + u_i)c_{j,i}}{\sum_{i \in S(u)-\{j\}} c_{j,i}}$$

where

$$c_{j,i} = card(S_{j,i}(\chi))$$

$P^{wS1}(u)_j$ means our prediction using Weighted Slope One of user u's rating for item j. So, for example $P^{wS1}(Ben)_{Whitney\ Houston}$ means our prediction for what *Ben* would rate Whitney Houston.

Let's say I am interested in answering that question: How might Ben rate Whitney Houston?

Let's dissect the numerator.

$$\sum_{i \in S(u)-\{j\}}$$

means for every musician that Ben has rated (except for Whitney Houston that is the {j} bit).

The entire numerator means for every musician i that Ben has rated (except for Whitney Houston) we will look up the deviation of Whitney Houston to that musician and we will add that to Ben's rating for musician i. We multiply that by the cardinality of that pair—the number of people that rated both musicians (Whitney and musician i).

Let's step through this:

First, here are Ben's ratings and our deviations table from before:

	Taylor Swift	PSY	Whitney Houston
Ben	5	2	?

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2	1
PSY	-2	0	-0.75
Whitney Houston	-1	0.75	0

1. Ben has rated Taylor Swift and gave her a 5—that is the u_i.
2. The deviation of Whitney Houston with respect to Taylor Swift is -1 —this is the $dev_{j,i}$.
3. $dev_{j,i} + u_i$ then is 4.
4. Looking at page 3-19 we see that there were two people (Amy and Daisy) that rated both Taylor Swift and Whitney Houston so $c_{j,i} = 2$
5. So $(dev_{j,i} + u_i)\ c_{j,i} = 4 \times 2 = 8$
6. Ben has rated PSY and gave him a 2.
7. The deviation of Whitney Houston with respect to PSY is 0.75
8. $dev_{j,i} + u_i$ then is 2.75
9. Two people rated both Whitney Houston and PSY so $(dev_{j,i} + u_i)\ c_{j,i} = 2.75 \times 2 = 5.5$
10. We sum up steps 5 and 9 to get 13.5 for the numerator

DENOMINATOR

11. Dissecting the denominator we get something like for every musician that Ben has rated, sum the cardinalities of those musicians (how many people rated both that musician and

Whitney Houston). So Ben has rated Taylor Swift and the cardinality of Taylor Swift and Whitney Houston (that is, the total number of people that rated both of them) is 2. Ben has rated PSY and his cardinality is also 2. So the denominator is 4.

12. So our prediction of how well Ben will like Whitney Houston is $\frac{13.5}{4} = 3.375$

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": {"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 4},
          "Ben": {"Taylor Swift": 5, "PSY": 2},
          "Clara": {"PSY": 3.5, "Whitney Houston": 4},
          "Daisy": {"Taylor Swift": 5, "Whitney Houston": 3}}
```

First computing the deviations.

Again, the formula for computing deviations is

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

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:

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2 (2)	1 (2)
PSY	-2 (2)	0	-0.75 (2)
Whitney Houston	-1 (2)	0.75 (2)	0

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

The pseudoCode 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 pseudocode looks pretty nice but as you can see, there is a disconnect between the data format expected by the pseudocode 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 pseudocode 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": {"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 4},
          "Ben": {"Taylor Swift": 5, "PSY": 2},
          "Clara": {"PSY": 3.5, "Whitney Houston": 4},
          "Daisy": {"Taylor Swift": 5, "Whitney Houston": 3}}
```

So the first time through the loop `ratings = {"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 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.

```
    def __init__(self, data, k=1, metric='pearson', n=5):
        ...

        #
        # The following two variables are used for Slope One
        #
        self.frequencies = {}
        self.deviations = {}
```

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.

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:

```
{"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 4}
```

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

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

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

in only 18 lines of code. Incredible!

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

```
users2 = {"Amy": {"Taylor Swift": 4, "PSY": 3, "Whitney Houston": 4},
          "Ben": {"Taylor Swift": 5, "PSY": 2},
          "Clara": {"PSY": 3.5, "Whitney Houston": 4},
          "Daisy": {"Taylor Swift": 5, "Whitney Houston": 3}}
```

I get

```
>>> r = recommender(users2)
>>> r.computeDeviations()
>>> r.deviations
{'PSY': {'Taylor Swift': -2.0, 'Whitney Houston': -0.75}, 'Taylor
Swift': {'PSY': 2.0, 'Whitney Houston': 1.0}, 'Whitney Houston':
{'PSY': 0.75, 'Taylor Swift': -1.0}}
```

which is what we obtained when we computed this example by hand:

	Taylor Swift	PSY	Whitney Houston
Taylor Swift	0	2	1
PSY	-2	0	-0.75
Whitney Houston	-1	0.75	0

Shout out to Bryan O'Sullivan and his blog teideal glic deisbhéalach (serpentine.com/blog) which presented a Python implementation 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:

$$P^{wS1}(u)_j = \frac{\sum_{i \in S(u)-\{j\}} (dev_{j,i} + u_i)c_{j,i}}{\sum_{i \in S(u)-\{j\}} c_{j,i}}$$

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 pseudocode. You try:

sharpen your pencil

The formula in pseudo English:

sharpen your pencil - a solution

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

{"Taylor Swift": 5, "PSY": 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 =  [(self.convertProductID2name(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)
[('Whitney Houston', 3.375)]
```

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.

Again, you can download the code to this chapter at www.guidetodatamining.com!

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

```
>>> r = recommender(0)
>>> r.loadMovieLens('/Users/raz/Downloads/ml-100k/')
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
...
```

User 1 rated all these movies a '5'!

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

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

(this takes about 50 seconds to run on my laptop)

Finally, let's get recommendations for User 1:

```
>>> r.slopeOneRecommendations(r.data['1'])
```

```
[('Entertaining Angels: The Dorothy Day Story (1996)', 6.375), ('Aiqing
wansui (1994)', 5.849056603773585), ('Boys, Les (1997)',
5.644970414201183), ("Someone Else's America (1995)",
5.391304347826087), ('Santa with Muscles (1996)', 5.380952380952381),
('Great Day in Harlem, A (1994)', 5.275862068965517),  ...
```

and user 25:

```
>>> r.slopeOneRecommendations(r.data['25'])
```

```
[('Aiqing wansui (1994)', 5.674418604651163), ('Boys, Les (1997)',
5.523076923076923), ('Star Kid (1997)', 5.25), ('Santa with Muscles
(1996)',
```

Projects

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 suggest 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 x 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.

Congratulations on finishing chapter 3!!

There was some hard work in this chapter--dissecting complex-looking formulas to gain an understanding of them and then implementing them.