One last formula – Cosine Similarity

I would like to present one last formula, which is very popular in text mining but also used in collaborative filtering—cosine similarity. To see when we might use this formula, let's say I change my example slightly. We will keep track of the number of times a person played a particular song track and use that information to base our recommendations on.

	number of plays		
	The Decemberists The King is Dead	Radiohead The King of Limbs	Katy Perry E.T.
Ann	10	5	32
Ben	15	25	1
Sally	12	6	27

Just by eye-balling the above chart (and by using any of the distance formulas mentioned above) we can see that Sally is more similar in listening habits to Ann than Ben is.

So what is the problem?

I have around four thousand tracks in iTunes. Here is a snapshot of the top few ordered by number of plays:

✓	Name	Time	Artist	Album	Genre	Plays ▾
✓	Moonlight Sonata	7:38	Marcus Miller	Silver Rain	Jazz+Funk	25
✓	Blast!	5:43	Marcus Miller	Marcus	Jazz	20
✓	Art Isn't Real (City of Sin)	2:48	Deer Tick	War Elephant	Alt-Country	19
✓	Between the Lines	4:35	Sara Bareilles	Little Voice	Folk	19
✓	Stay Around A Little Longer (Feat. B.B. King)	5:00	BUDDY GUY	Living Proof	Blues	18
✓	My Companjera	3:22	Gogol Bordello	Trans-Continental...	Alternative...	18
✓	Rebellious Love	3:57	Gogol Bordello	Trans-Continental...	Alternative...	18
✓	Immigraniada (We Comin' Rougher)	3:46	Gogol Bordello	Trans-Continental...	Alternative...	18
✓	Love Song	4:19	Sara Bareilles	Little Voice	Folk	18

So my top track is Moonlight Sonata by Marcus Miller with 25 plays. Chances are that you have played that track zero times. In fact, chances are good that you have not played any of my top tracks. In addition, there are over 15 million tracks in iTunes and I have only four thousand. So the data for a single person is sparse since it has relatively few non-zero attributes (plays of a track). When we compare two people by using the number of plays of the 15 million tracks, mostly they will have shared zeros in common. However, we do not want to use these shared zeros when we are computing similarity.

A similar case can be made when we are comparing text documents using words. Suppose we liked a certain book, say T*om Corbett Space Cadet: The Space Pioneers* by Carey Rockwell and we want to find a similar book. One possible way is to use word frequency. The attributes will be individual words and the values of those attributes will be the frequency of those words in the book. So 6.13% of the words in *The Space Pioneers* are occurrences of the word *the*, 0.89% are the word *Tom*, 0.25% of the words are *space*. I can compute the similarity of this book to others by using these word frequencies. However, the same problem related to sparseness of data occurs here. There are 6,629 unique words in *The Space Pioneers* and there are a bit over one million unique words in English. So if our attributes are English words, there will be relatively few non-zero attributes for *The Space Pioneers* or any other book. Again, any measure of similarity should not depend on the shared-zero values.

Cosine similarity ignores 0-0 matches. It is defined as

$$\cos(x,y) = \frac{x \cdot y}{||x|| \times ||y||}$$

where · indicates the dot product and ||x|| indicates the length of the vector x. The length of a vector is

$$\sqrt{\sum_{i=1}^{n} x_i^2}$$

Let's give this a try with the perfect agreement example used above:

	Blues Traveler	Norah Jones	Phoenix	The Strokes	Weird Al
Clara	4.75	4.5	5	4.25	4
Robert	4	3	5	2	1

The two vectors are:

$$x = (4.75, 4.5, 5, 4.25, 4)$$
$$y = (4, 3, 5, 2, 1)$$

then

$$||x|| = \sqrt{4.75^2 + 4.5^2 + 5^2 + 4.25^2 + 4^2} = \sqrt{101.875} = 10.09$$

$$||y|| = \sqrt{4^2 + 3^2 + 5^2 + 2^2 + 1^2} = \sqrt{55} = 7.416$$

The dot product is

$$x \cdot y = (4.75 \times 4) + (4.5 \times 3) + (5 \times 5) + (4.25 \times 2) + (4 \times 1) = 70$$

And the cosine similarity is

$$\cos(x, y) = \frac{70}{10.093 \times 7.416} = \frac{70}{74.85} = 0.935$$

The cosine similarity rating ranges from 1 indicated perfect similarity to -1 indicate perfect negative similarity. So 0.935 represents very good agreement.

sharpen your pencil

Compute the Cosine Similarity between Angelica and Veronica (from our dataset). (Consider dashes equal to zero)

	Blues Traveler	Broken Bells	Deadmau 5	Norah Jones	Phoenix	Slightly Stoopid	The Strokes	Vampire Weekend
Angelica	3.5	2	-	4.5	5	1.5	2.5	2
Veronica	3	-	-	5	4	2.5	3	-

sharpen your pencil - solution

Compute the Cosine Similarity between Angelica and Veronica (from our dataset).

	Blues Traveler	Broken Bells	Deadmau 5	Norah Jones	Phoenix	Slightly Stoopid	The Strokes	Vampire Weekend
Angelica	3.5	2	-	4.5	5	1.5	2.5	2
Veronica	3	-	-	5	4	2.5	3	-

$x = (3.5, 2, 0, 4.5, 5, 1.5, 2.5, 2)$

$y = (3, 0, 0, 5, 4, 2.5, 3, 0)$

$$||x|| = \sqrt{3.5^2 + 2^2 + 0^2 + 4.5^2 + 5^2 + 1.5^2 + 2.5^2 + 2^2} = \sqrt{74} = 8.602$$

$$||y|| = \sqrt{3^2 + 0^2 + 0^2 + 5^2 + 4^2 + 2.5^2 + 3^2 + 0^2} = \sqrt{65.25} = 8.078$$

The dot product is

$$x \cdot y =$$
$$(3.5 \times 3) + (2 \times 0) + (0 \times 0) + (4.5 \times 5) + (5 \times 4) + (1.5 \times 2.5) + (2.5 \times 3) + (2 \times 0) = 64.25$$

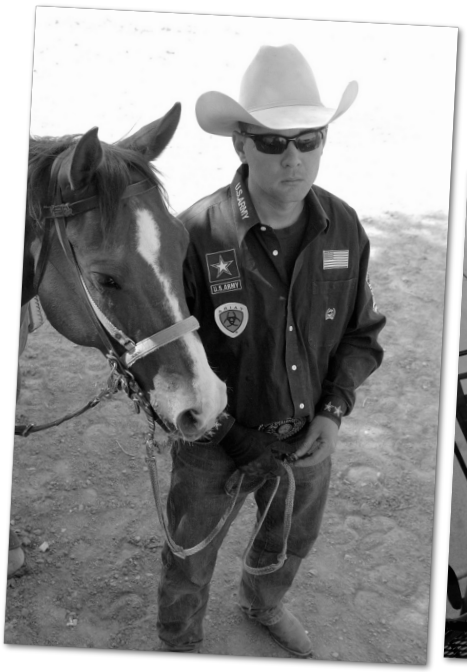
Cosine Similarity is

$$\cos(x, y) = \frac{64.25}{8.602 \times 8.078} = \frac{64.25}{69.487} = 0.9246$$

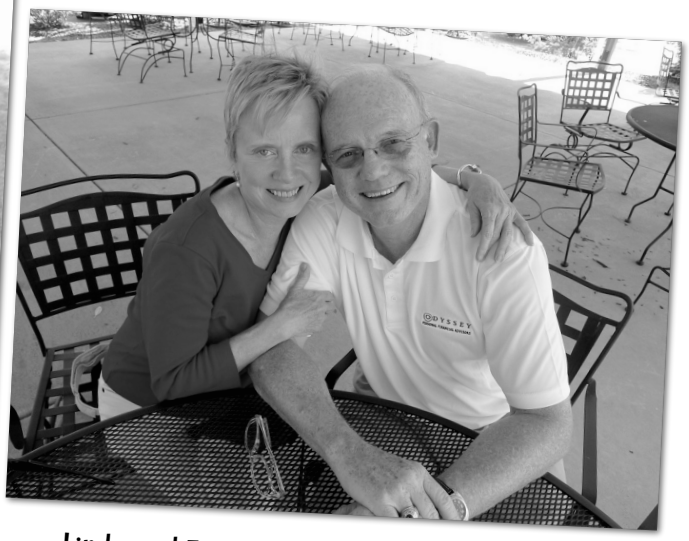
Which similarity measure to use?

We will be exploring this question throughout the book. For now, here are a few helpful hints:

So, if the data is dense (nearly all attributes have non-zero values) then Manhattan and Euclidean are reasonable to use. What happens if the data is not dense? Consider an expanded music rating system and three people, all of which have rated 100 songs on our site:



Jake: hardcore fan of Country



Linda and Eric: love, love, love 60s rock!

Linda and Eric enjoy the same kind of music. In fact, among their ratings, they have 20 songs in common and the difference in their ratings of those 20 songs (on a scale of 1 to 5) averages only 0.5!! The Manhattan Distance between them would be 20 x .5 = 10. The Euclidean Distance would be:

$$d = \sqrt{(.5)^2 \times 20} = \sqrt{.25 \times 20} = \sqrt{5} = 2.236$$

Linda and Jake have rated only one song in common: Chris Cagle's *What a Beautiful Day*. Linda thought it was okay and rated it a 3, Jake thought it was awesome and gave it a 5. So the Manhattan Distance between Jake and Linda is 2 and the Euclidean Distance is

$$d = \sqrt{(3-5)^2} = \sqrt{4} = 2$$

So both the Manhattan and Euclidean Distances show that Jake is a closer ~~match~~ to Linda than Eric is. So in this case both distance measures produce poor results.

Hey, I have an idea that might fix this problem.

Right now, people rate tunes on a scale of 1 to 5. How about for the tunes people don't rate I will assume the rating is 0. That way we solve the problem of sparse data as every object has a value for every attribute!

Good idea, but that doesn't work either. To see why we need to bring in a few more characters into our little drama: Cooper and Kelsey. Jake, Cooper and Kelsey have amazingly similar musical tastes. Jake has rated 25 songs on our site.

Cooper

Kelsey

Cooper has rated 26 songs, and 25 of them are the same songs Jake rated. They love the same kind of music and the average distance in their ratings is only 0.25!!

Kelsey loves both music and our site and has rated 150 songs. 25 of those songs are the same as the ones Cooper and Jake rated. Like Cooper, the average distance in her ratings and Jake's is only 0.25!!

Our gut feeling is that Cooper and Kelsey are equally close matches to Jake.

Now consider our modified Manhattan and Euclidean distance formulas where we assign a 0 for every song the person didn't rate.

With this scheme, Cooper is a much closer match to Jake than Kelsey is.

Why?

To answer why, let us look at a the following simplified example (again, a 0 means that person did not rate that song):

Song:	1	2	3	4	5	6	7	8	9	10
Jake	0	0	0	4.5	5	4.5	0	0	0	0
Cooper	0	0	4	5	5	5	0	0	0	0
Kelsey	5	4	4	5	5	5	5	5	4	4

Again, looking at the songs they mutually rated (songs 4, 5, and 6), Cooper and Kelsey seem like equally close matches for Jake. However, Manhattan Distance using those zero values tells a different story:

$$d_{Cooper,Jake} = (4-0)+(5-4.5)+(5-5)+5-4.5) = 4+0.5+0+0.5 = 5$$

$$d_{Kelsey,Jake} = (5-0)+(4-0)+(4-0)+(5-4.5)+(5-5)+(5-4.5)+(5-0)$$

$$+(5-0)+(4-0)+(4-0)$$

$$= 5+4+4+0.5+0+.5+5+5+4+4 = 32$$

The problem is that these zero values tend to dominate any measure of distance. So the solution of adding zeros is no better than the original distance formulas. One workaround people have used is to compute—in some sense—an 'average' distance where one computes the distance by using songs they rated in common divided that by the number of songs they rated in common.

Again, Manhattan and Euclidean work spectacularly well on dense data, but if the data is sparse it may be better to use Cosine Similarity.

Weirdnesses

Suppose we are trying to make recommendations for Amy who loves Phoenix, Passion Pit and Vampire Weekend. Our closest match is Bob who also loves Phoenix, Passion Pit, and Vampire Weekend. His father happens to play accordion for the Walter Ostanek Band, this year's Grammy winner in the polka category. Because of familial obligations, Bob gives 5 stars to the Walter Ostanek Band. Based on our current recommendation system, we think Amy will absolutely love the band. But common sense tells us she probably won't.

Or think of Professor Billy Bob Olivera who loves to read data mining books and science fiction. His closest match happens to be me, who also likes data mining books and science fiction. However, I like standard poodles and have rated The Secret Lives of Standard Poodles highly. Our current recommendation system would likely recommend that book to the professor.

The problem is that we are relying on a single “most similar” person. Any quirk that person has is passed on as a recommendation. One way of evening out those quirks is to base our recommendations on more than one person who is similar to our user. For this we can use the k-nearest neighbor approach.

K-nearest neighbor

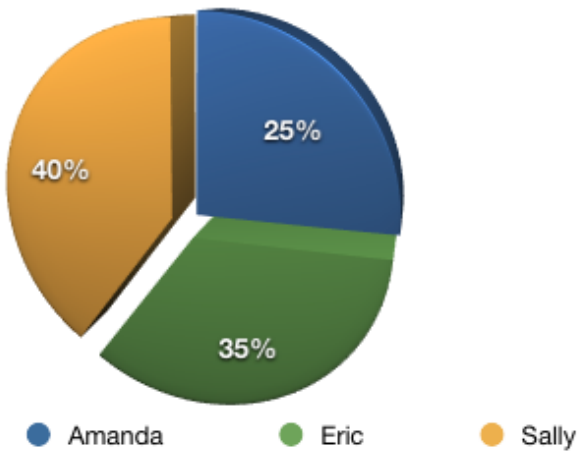
In the k-nearest neighbor approach to collaborative filtering we use k most similar people to determine recommendations. The best value for k is application specific—you will need to do some experimentation. Here's an example to give you the basic idea.

Suppose I would like to make recommendations for Ann and am using k-nearest neighbor with k=3. The three nearest neighbors and their Pearson scores are shown in the following table:

Person	Pearson
Sally	0.8
Eric	0.7
Amanda	0.5

0.8 + 0.7 + 0.5 = 2.0

Each of these three people are going to influence the recommendations. The question is how can I determine how much influence each person should have. If there is a Pie of Influence™, how big a slice should I give each person? If I add up the Pearson scores I get 2. Sally's share is 0.8/2 or 40%. Eric's share is 35% (0.7 / 2) and Amanda's share is 25%.



Suppose Amanda, Eric, and Sally, rated the band, The Grey Wardens as follows

Person	Grey Wardens Rating
Amanda	4.5
Eric	5
Sally	3.5

Person	Grey Wardens Rating	Influence
Amanda	4.5	25.00%
Eric	5	35.00%
Sally	3.5	40.00%

Projected rating = (4.5 x 0.25) + (5 x 0.35) + (3.5 x 0.4)

= 4.275

sharpen your pencil

Suppose I use the same data as above but use a k-nearest neighbor approach with k=2. What is my projected rating for Grey Wardens?

Person	Pearson
Sally	0.8
Eric	0.7
Amanda	0.5

Person	Grey Wardens Rating
Amanda	4.5
Eric	5
Sally	3.5

solution

Person	Pearson
Sally	0.8
Eric	0.7
Amanda	0.5

Person	Grey Wardens Rating
Amanda	4.5
Eric	5
Sally	3.5

Projected rating = Sally's portion + Eric's portion

= (3.5 x (0.8 / 1.5)) + (5 x (0.7 / 1.5))

= (3.5 x .5333) + (5 x 0.4667)

= 1.867 + 2.333

= 4.2

A Python Recommendation Class

I combined some of what we covered in this chapter in a Python Class. Even though it is slightly long I have included the code here (don't forget you can download the code at http://www.guidetodatamining.com).

```
import codecs
from math import sqrt

users = {"Angelica": {"Blues Traveler": 3.5, "Broken Bells": 2.0,
                      "Norah Jones": 4.5, "Phoenix": 5.0,
                      "Slightly Stoopid": 1.5,
                      "The Strokes": 2.5, "Vampire Weekend": 2.0},

         "Bill":{"Blues Traveler": 2.0, "Broken Bells": 3.5,
                 "Deadmau5": 4.0, "Phoenix": 2.0,
                 "Slightly Stoopid": 3.5, "Vampire Weekend": 3.0},

         "Chan": {"Blues Traveler": 5.0, "Broken Bells": 1.0,
                  "Deadmau5": 1.0, "Norah Jones": 3.0, "Phoenix": 5,
                  "Slightly Stoopid": 1.0},

         "Dan": {"Blues Traveler": 3.0, "Broken Bells": 4.0,
                 "Deadmau5": 4.5, "Phoenix": 3.0,
                 "Slightly Stoopid": 4.5, "The Strokes": 4.0,
                 "Vampire Weekend": 2.0},

         "Hailey": {"Broken Bells": 4.0, "Deadmau5": 1.0,
                    "Norah Jones": 4.0, "The Strokes": 4.0,
                    "Vampire Weekend": 1.0},

         "Jordyn":  {"Broken Bells": 4.5, "Deadmau5": 4.0,
                     "Norah Jones": 5.0, "Phoenix": 5.0,
                     "Slightly Stoopid": 4.5, "The Strokes": 4.0,
                     "Vampire Weekend": 4.0},
```

```
         "Sam": {"Blues Traveler": 5.0, "Broken Bells": 2.0,
                 "Norah Jones": 3.0, "Phoenix": 5.0,
                 "Slightly Stoopid": 4.0, "The Strokes": 5.0},

         "Veronica": {"Blues Traveler": 3.0, "Norah Jones": 5.0,
                      "Phoenix": 4.0, "Slightly Stoopid": 2.5,
                      "The Strokes": 3.0}
        }



class recommender:

    def __init__(self, data, k=1, metric='pearson', n=5):
        """ initialize recommender
        currently, if data is dictionary the recommender is initialized
        to it.
        For all other data types of data, no initialization occurs
        k is the k value for k nearest neighbor
        metric is which distance formula to use
        n is the maximum number of recommendations to make"""
        self.k = k
        self.n = n
        self.username2id = {}
        self.userid2name = {}
        self.productid2name = {}
        # for some reason I want to save the name of the metric
        self.metric = metric
        if self.metric == 'pearson':
            self.fn = self.pearson
        #
        # if data is dictionary set recommender data to it
        #
        if type(data).__name__ == 'dict':
            self.data = data
```

```
def convertProductID2name(self, id):
    """Given product id number return product name"""
    if id in self.productid2name:
        return self.productid2name[id]
    else:
        return id


def userRatings(self, id, n):
    """Return n top ratings for user with id"""
    print ("Ratings for " + self.userid2name[id])
    ratings = self.data[id]
    print(len(ratings))
    ratings = list(ratings.items())
    ratings = [(self.convertProductID2name(k), v)
               for (k, v) in ratings]
    # finally sort and return
    ratings.sort(key=lambda artistTuple: artistTuple[1],
                 reverse = True)
    ratings = ratings[:n]
    for rating in ratings:
        print("%s\t%i" % (rating[0], rating[1]))



def loadBookDB(self, path=''):
    """loads the BX book dataset. Path is where the BX files are
    located"""
    self.data = {}
    i = 0
    #
    # First load book ratings into self.data
    #
    f = codecs.open(path + "BX-Book-Ratings.csv", 'r', 'utf8')
    for line in f:
        i += 1
```

```
        # separate line into fields
        fields = line.split(';')
        user = fields[0].strip('"')
        book = fields[1].strip('"')
        rating = int(fields[2].strip().strip('"'))
        if user in self.data:
            currentRatings = self.data[user]
        else:
            currentRatings = {}
        currentRatings[book] = rating
        self.data[user] = currentRatings
    f.close()
    #
    # Now load books into self.productid2name
    # Books contains isbn, title, and author among other fields
    #
    f = codecs.open(path + "BX-Books.csv", 'r', 'utf8')
    for line in f:
        i += 1
        # separate line into fields
        fields = line.split(';')
        isbn = fields[0].strip('"')
        title = fields[1].strip('"')
        author = fields[2].strip().strip('"')
        title = title + ' by ' + author
        self.productid2name[isbn] = title
    f.close()
    #
    #  Now load user info into both self.userid2name and
    #  self.username2id
    #
    f = codecs.open(path + "BX-Users.csv", 'r', 'utf8')
    for line in f:
        i += 1
        # separate line into fields
        fields = line.split(';')
        userid = fields[0].strip('"')
```

```
            location = fields[1].strip('"')
            if len(fields) > 3:
                age = fields[2].strip().strip('"')
            else:
                age = 'NULL'
            if age != 'NULL':
                value = location + '  (age: ' + age + ')'
            else:
                value = location
            self.userid2name[userid] = value
            self.username2id[location] = userid
        f.close()
        print(i)


    def pearson(self, rating1, rating2):
        sum_xy = 0
        sum_x = 0
        sum_y = 0
        sum_x2 = 0
        sum_y2 = 0
        n = 0
        for key in rating1:
            if key in rating2:
                n += 1
                x = rating1[key]
                y = rating2[key]
                sum_xy += x * y
                sum_x += x
                sum_y += y
                sum_x2 += pow(x, 2)
                sum_y2 += pow(y, 2)
        if n == 0:
            return 0
        # now compute denominator
        denominator = (sqrt(sum_x2 - pow(sum_x, 2) / n)
                       * sqrt(sum_y2 - pow(sum_y, 2) / n))
```

```
        if denominator == 0:
            return 0
        else:
            return (sum_xy - (sum_x * sum_y) / n) / denominator


    def computeNearestNeighbor(self, username):
        """creates a sorted list of users based on their distance to
        username"""
        distances = []
        for instance in self.data:
            if instance != username:
                distance = self.fn(self.data[username],
                                   self.data[instance])
                distances.append((instance, distance))
        # sort based on distance -- closest first
        distances.sort(key=lambda artistTuple: artistTuple[1],
                       reverse=True)
        return distances

    def recommend(self, user):
       """Give list of recommendations"""
       recommendations = {}
       # first get list of users  ordered by nearness
       nearest = self.computeNearestNeighbor(user)
       #
       # now get the ratings for the user
       #
       userRatings = self.data[user]
       #
       # determine the total distance
       totalDistance = 0.0
       for i in range(self.k):
          totalDistance += nearest[i][1]
       # now iterate through the k nearest neighbors
       # accumulating their ratings
       for i in range(self.k):
```

```
            # compute slice of pie
            weight = nearest[i][1] / totalDistance
            # get the name of the person
            name = nearest[i][0]
            # get the ratings for this person
            neighborRatings = self.data[name]
            # get the name of the person
            # now find bands neighbor rated that user didn't
            for artist in neighborRatings:
                if not artist in userRatings:
                    if artist not in recommendations:
                        recommendations[artist] = (neighborRatings[artist]
                                                   * weight)
                    else:
                        recommendations[artist] = (recommendations[artist]
                                                   + neighborRatings[artist]
                                                   * weight)
        # now make list from dictionary
        recommendations = list(recommendations.items())
        recommendations = [(self.convertProductID2name(k), v)
                           for (k, v) in recommendations]
        # finally sort and return
        recommendations.sort(key=lambda artistTuple: artistTuple[1],
                             reverse = True)
        # Return the first n items
        return recommendations[:self.n]
```

Example of this program executing

First, I will construct an instance of the recommender class with the data we previously used:

```
>>> r = recommender(users)
```

Some simple examples using these band ratings:

```
>>> r.recommend('Jordyn')
[('Blues Traveler', 5.0)]
>>> r.recommend('Hailey')
[('Phoenix', 5.0), ('Slightly Stoopid', 4.5)]
```

A New Dataset

Ok, it is time to look at a more realistic dataset. Cai-Nicolas Zeigler collected over one million ratings of books from the Book Crossing website. This ratings are of 278,858 users rating 271,379 books. This anonymized data is available at http://www.informatik.uni-freiburg.de/~cziegler/BX/ both as an SQL dump and a text file of comma-separated-values (CSV). I had some problems loading this data into Python due to apparent character encoding problems. My fixed version of the CSV files are available on this book's website.

The CSV files represent three tables:

- BX-Users, which, as the name suggests, contains information about the users. There is an integer user-id field, as well as the location (i.e., Albuquerque, NM) and age. The names have been removed to anonymize the data.
- BX-Books. Books are identified by the ISBN, book title, author, year of publication, and publisher.
- BX-Book-Ratings, which includes a user-id, book ISBN, and a rating from 0-10.

The function loadBookDB in the recommender class loads the data from these files.

Now I am going to load the book dataset. The argument to the loadBookDB function is the path to the BX book files.

```
>>> r.loadBookDB('/Users/raz/Downloads/BX-Dump/')
1700018
```

> Note:
> This is a large dataset and may take a bit of time to load on your computer. On my Hackintosh (2.8 GHz i7 860 with 8GB RAM) it takes 24 seconds to load the dataset and 30 seconds to run a query.

Now I can get recommendations for user 17118, a person from Toronto:

```
>>> r.recommend('171118')
[("The Godmother's Web by Elizabeth Ann Scarborough", 10.0), ("The Irrational
Season (The Crosswicks Journal, Book 3) by Madeleine L'Engle", 10.0), ("The
Godmother's Apprentice by Elizabeth Ann Scarborough", 10.0), ("A Swiftly
Tilting Planet by Madeleine L'Engle", 10.0), ('The Girl Who Loved Tom Gordon by
Stephen King', 9.0), ('The Godmother by Elizabeth Ann Scarborough', 8.0)]
```

```
>>> r.userRatings('171118', 5)
Ratings for toronto, ontario, canada
2421
The Careful Writer by Theodore M. Bernstein   10
Wonderful Life: The Burgess Shale and the Nature of History by Stephen Jay
Gould  10
Pride and Prejudice (World's Classics) by Jane Austen      10
The Wandering Fire (The Fionavar Tapestry, Book 2) by Guy Gavriel Kay   10
Flowering trees and shrubs: The botanical paintings of Esther Heins by Judith
Leet   10
```

Projects

You won't really learn this material unless you play around with the code. Here are some suggestions of what you might try.

1. Implement Manhattan distance and Euclidean distance and compare the results of these three methods.

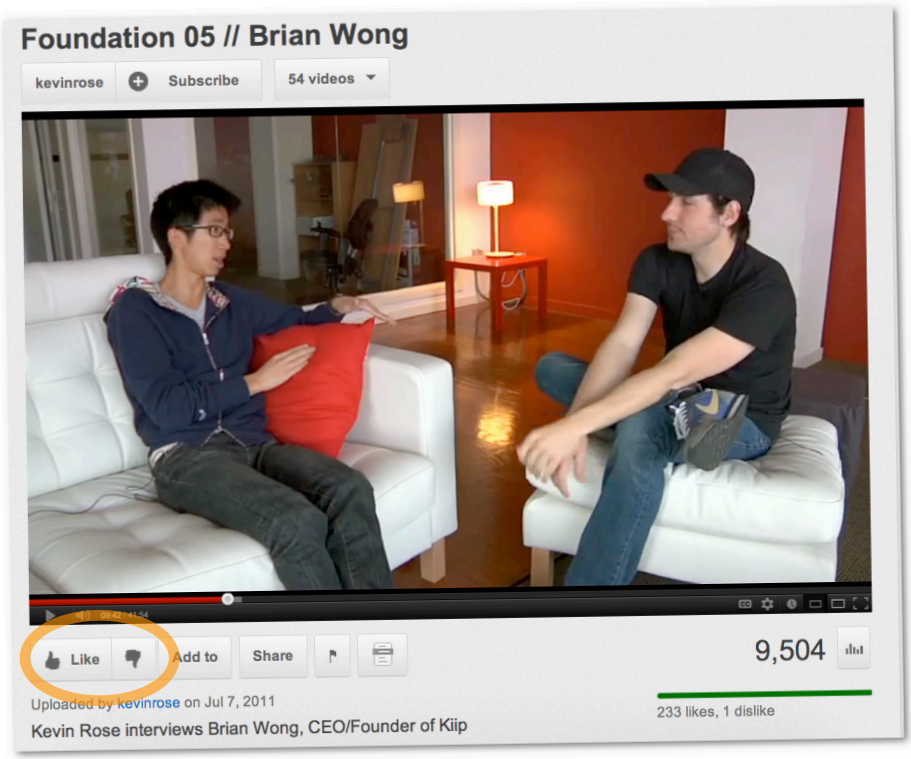
2. The book website has a file containing movie ratings for 25 movies. Create a function that loads the data into your classifier. The recommend method described above should recommend movies for a specific person.

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

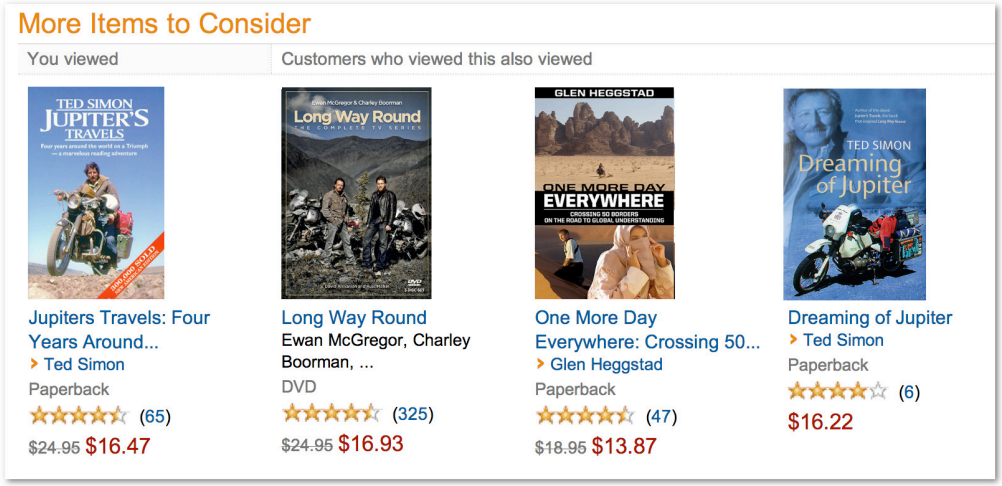
Published 8 months ago by Eric Maroney

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.

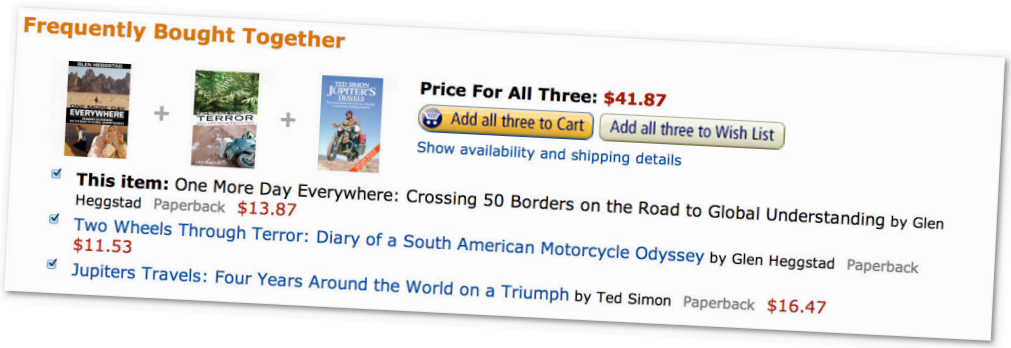
After observing what a user clicks on for a few weeks you can imagine that we could develop a reasonable profile of that user—she doesn't like sports but seems to like technology news. If the user clicks on the article "Fastest Way to Lose Weight Discovered by Professional Trainers" and the article "Slow 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				

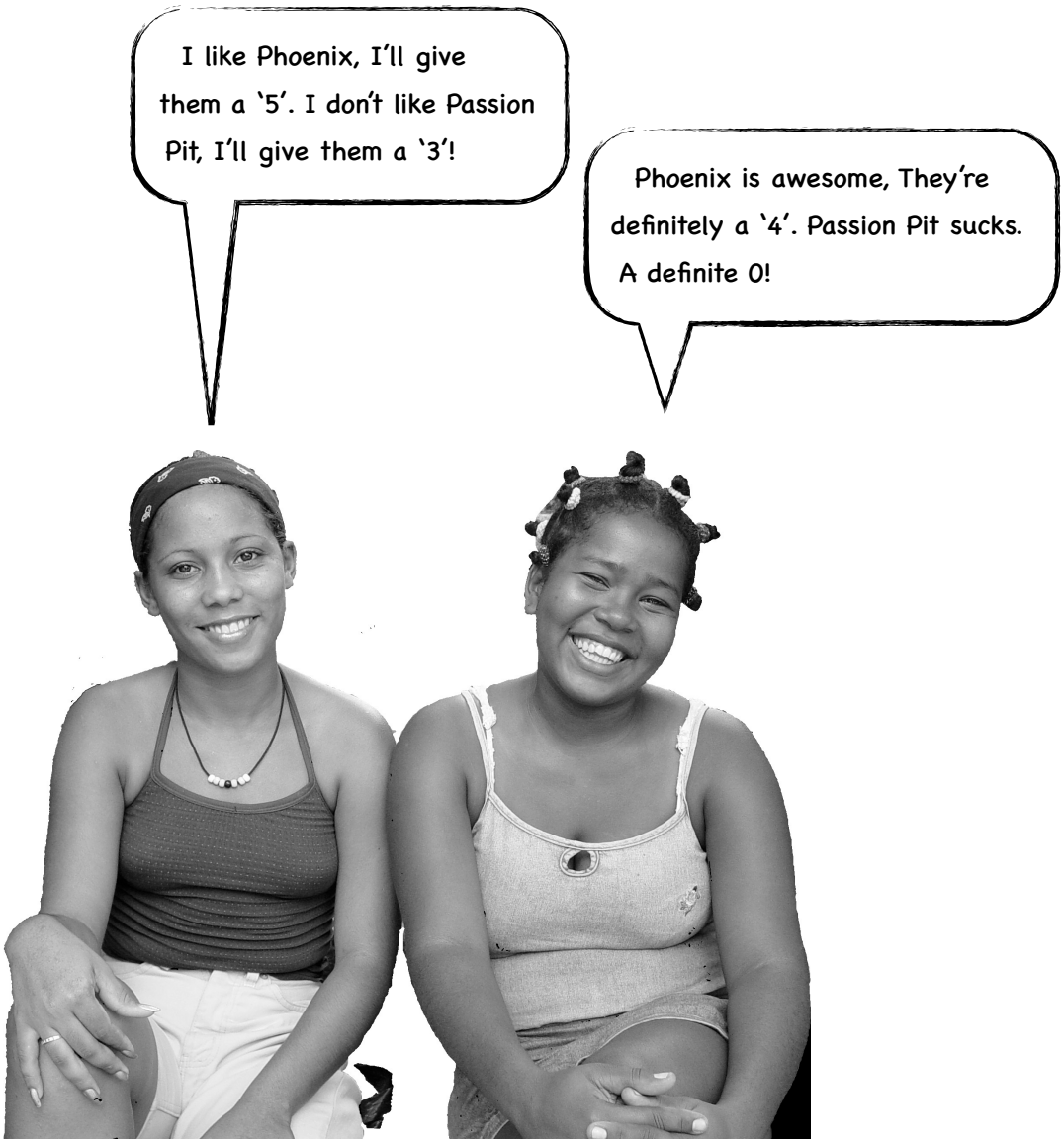
We would like to compute the similarity of Phoenix to Passion Pit. To do this we only use users who rated both bands as indicated by the blue squares. If we were doing user-based filtering we would determine the similarity between rows. For item-based filtering we are determining the similarity between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

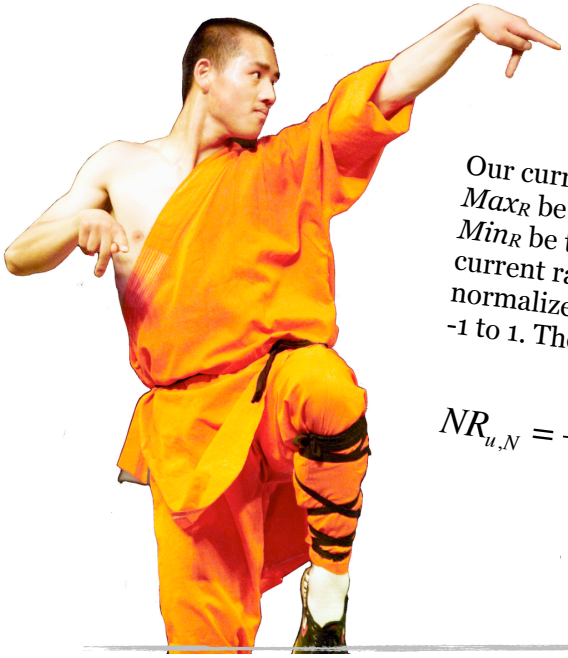
$R_{u,N}$ is the rating user u gave item N

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

$R_{u,N}$ 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 ...

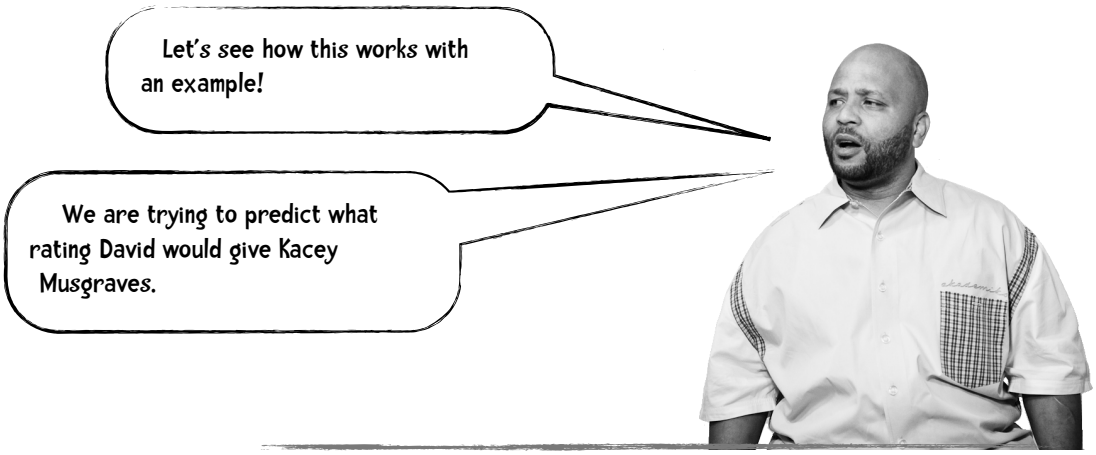
$$NR_{u,N} = \frac{2(R_{u,N} - Min_R) - (Max_R - Min_R)}{(Max_R - Min_R)} = \frac{2(2-1) - (5-1)}{(5-1)} = \frac{-2}{4} = -0.5$$

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!