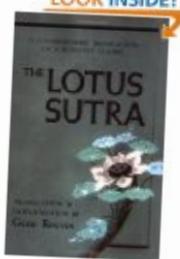
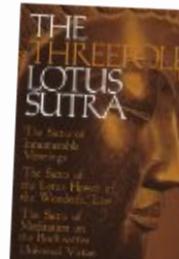
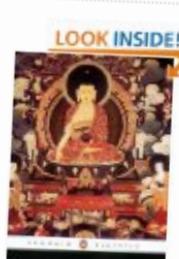


Chapter 2: Collaborative Filtering

I like what you like

We are going to start our exploration of data mining by looking at recommendation systems. Recommendation systems are everywhere—from Amazon:

More Items to Consider

You viewed	Customers who viewed this also viewed
 LOOK INSIDE! The Lotus Sutra: A Contemporary... Gene Reeves Paperback \$19.95 \$13.57	 LOOK INSIDE! The Lotus Sutra Burton Watson Paperback \$24.95 \$16.47
 The Threefold Lotus Sutra Bunno Kato, Yoshiro Tamura Paperback \$16.95 \$11.53	 LOOK INSIDE! Buddhist Scriptures (Penguin Classics) Donald S. Lopez Paperback \$17.00 \$11.56

[View or edit your browsing history](#)

to last.fm recommending music or festivals:

The screenshot shows a Last.fm artist profile for Stanley Clarke. At the top left is a black and white photo of him playing a bass guitar. To the right of the photo, his name "Stanley Clarke" is displayed with a play button icon, followed by "602,599 plays (82,365 listeners)". Below this is a "Add to Your Library" button. Underneath the button are genre tags: "jazz, bass, fusion, funk, jazz fusion". A short bio follows: "Clarke was born in Philadelphia, Pennsylvania. Having graduated from the Philadelphia Musical Academy... [Read more](#)". In the top right corner of the main profile area is a close button (an 'X'). To the right of the main profile is a sidebar titled "Similar Artists from Your Library" which lists four artists: Marcus Miller (40 plays), Herbie Hancock (30 plays), Victor Wooten (2 plays), and Hiromi (60 plays). Below these is a link "Return to Forever (1 play)".

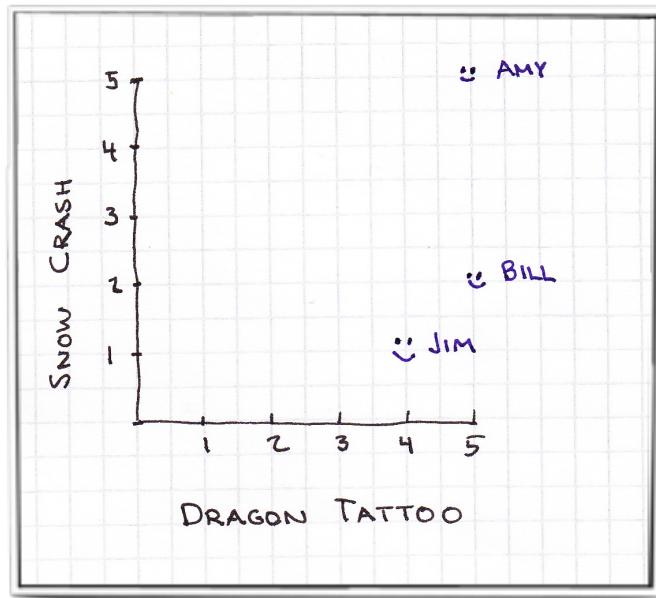
Festival	Date	Compatibility
Amarone in Jazz 2010 Verona e Provincia, Verona e Provincia, Italy Hiromi, Mike Stern, Gianluigi Trovesi Quintet, Nicola Trocchia, Apotech, Simona Severini	Tue 5 Oct Mon 11 Oct	81%

In the Amazon example, above, Amazon combines two bits of information to make a recommendation. The first is that I viewed *The Lotus Sutra* translated by Gene Reeves; the second, that customers who viewed that translation of the *Lotus Sutra* also viewed several other translations.

The recommendation method we are looking at in this chapter is called collaborative filtering. It's called *collaborative* because it makes recommendations based on other people—in effect, people collaborate to come up with recommendations. It works like this. Suppose the task is to recommend a book to you. I search among other users of the site to find one that is similar to you in the books she enjoys. Once I find that similar person I can see what she likes and recommend those books to you—perhaps Paolo Bacigalupi's *The Windup Girl*.

How do I find someone who is similar?

So the first step is to find someone who is similar. Here's the simple 2D (dimensional) explanation. Suppose users rate books on a 5 star system—zero stars means the book is terrible, 5 stars means the book is great. Because I said we are looking at the simple 2D case, we restrict our ratings to two books: Neal Stephenson's *Snow Crash* and the Steig Larsson's *The Girl with the Dragon Tattoo*.



First, here's a table showing 3 users who rated these books

	Snow Crash	Girl with the Dragon Tattoo
Amy	5★	5★
Bill	2★	5★
Jim	1★	4★

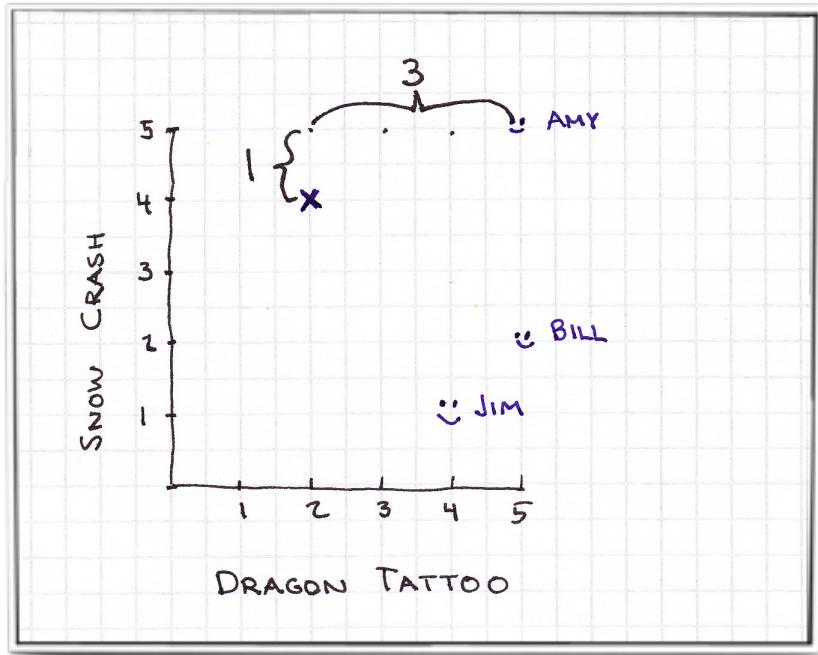
I would like to recommend a book to the mysterious Ms. X who rated Snow Crash 4 stars and The Girl with the Dragon Tattoo 2 stars. The first task is to find the person who is most similar, or closest, to Ms. X. I do this by computing distance.

Manhattan Distance

The easiest distance measure to compute is what is called Manhattan Distance or cab driver distance. In the 2D case, each person is represented by an (x, y) point. I will add a subscript to the x and y to refer to different people. So (x_1, y_1) might be Amy and (x_2, y_2) might be the elusive Ms. X. Manhattan Distance is then calculated by

$$| x_1 - x_2 | + | y_1 - y_2 |$$

(so the absolute value of the difference between the x values plus the absolute value of the difference between the y values). So the Manhattan Distance for Amy and Ms. X is 4:



Computing the distance between Ms. X and all three people gives us:

	Distance From Ms. X
Amy	4
Bill	5
Jim	5

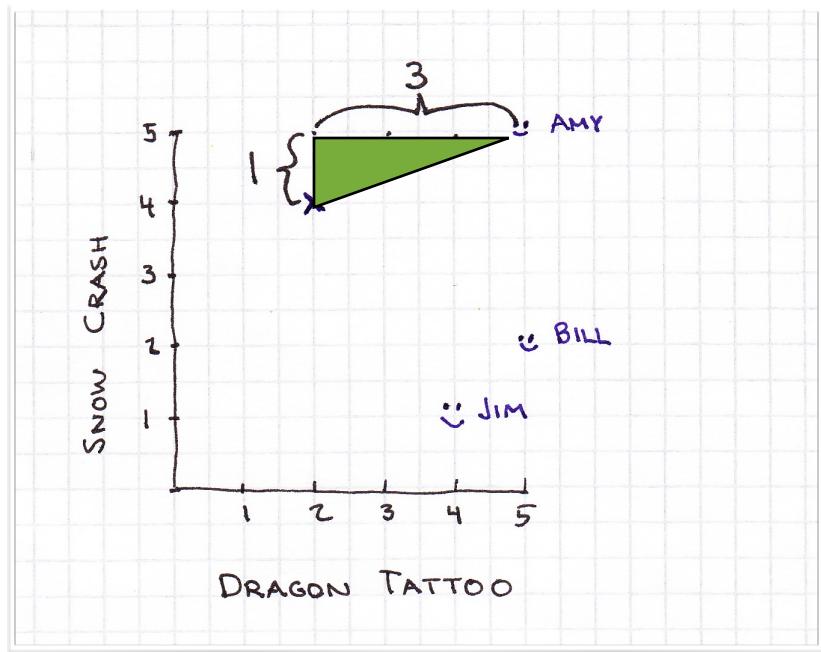
Amy is the closest match. We can look in her history and see, for example, that she gave five stars to Paolo Bacigalupi's *The Windup Girl* and we would recommend that book to Ms. X.

Euclidean Distance

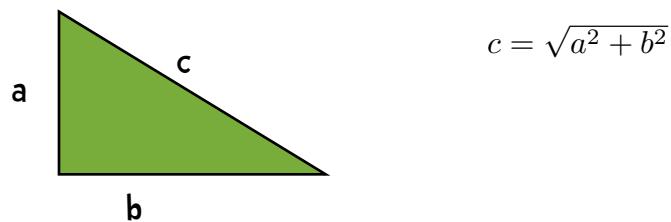
One benefit of Manhattan Distance is that it is fast to compute. If we are Facebook and are trying to find who among one million users is most similar to little Danny from Kalamazoo, fast is good.

Pythagorean Theorem

You may recall the Pythagorean Theorem from your distant educational past. Here, instead of finding the Manhattan Distance between Amy and Ms. X (which was 4) we are going to figure out the straight line, as-the-crow-flies, distance



The Pythagorean Theorem tells us how to compute that distance.



This straight-line, as-the-crow-flies distance we are calling Euclidean Distance. The formula is

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Recall that x_1 is how well person 1 liked *Dragon Tattoo* and x_2 is how well person 2 liked it; y_1 is how well person 1 liked *Snow Crash* and y_2 is how well person 2 liked it.

Amy rated both *Snow Crash* and *Dragon Tattoo* a 5; The elusive Ms. X rated *Dragon Tattoo* a 2 and *Snow Crash* a 4. So the Euclidean distance between

$$\sqrt{(5 - 2)^2 + (5 - 4)^2} = \sqrt{3^2 + 1^2} = \sqrt{10} = 3.16$$

Computing the rest of the distances we get

	Distance From Ms. X
Amy	3.16
Bill	3.61
Jim	3.61

N-dimensional thinking

Let's branch out slightly from just looking at rating two books (and hence 2D) to looking at something slightly more complex. Suppose we work for an online streaming music service and we want to make the experience more compelling by recommending bands. Let's say users can rate bands on a star system 0-5 stars and they can give half star ratings (for example, you can give a band 2.5 stars). The following chart shows 8 users and their ratings of eight bands.

	Angelica	Bill	Chan	Dan	Hailey	Jordyn	Sam	Veronica
Blues Traveler	3.5	2	5	3	-	-	5	3
Broken Bells	2	3.5	1	4	4	4.5	2	-
Deadmau5	-	4	1	4.5	1	4	-	-
Norah Jones	4.5	-	3	-	4	5	3	5
Phoenix	5	2	5	3	-	5	5	4
Slightly Stoopid	1.5	3.5	1	4.5	-	4.5	4	2.5
The Strokes	2.5	-	-	4	4	4	5	3
Vampire Weekend	2	3	-	2	1	4	-	-

The hyphens in the table indicate that a user didn't rate that particular band. For now we are going to compute the distance based on the number of bands they both reviewed. So, for example, when computing the distance between Angelica and Bill, we will use the ratings for *Blues Traveler*, *Broken Bells*, *Phoenix*, *Slightly Stoopid*, and *Vampire Weekend*. So the Manhattan Distance would be:

	Angelica	Bill	Difference
Blues Traveler	3.5	2	1.5
Broken Bells	2	3.5	1.5
Deadmau5	-	4	
Norah Jones	4.5	-	
Phoenix	5	2	3
Slightly Stoopid	1.5	3.5	2
The Strokes	2.5	-	-
Vampire Weekend	2	3	1
Manhattan Distance:			9

Computing the Euclidean Distance is similar. We only use the bands the both reviewed:

	Angelica	Bill	Difference	Difference ²
Blues Traveler	3.5	2	1.5	2.25
Broken Bells	2	3.5	1.5	2.25
Deadmau5	-	4		
Norah Jones	4.5	-		
Phoenix	5	2	3	9
Slightly Stoopid	1.5	3.5	2	4
The Strokes	2.5	-	-	
Vampire Weekend	2	3	1	1
Sum of squares				18.5
Euclidean Distance				4.3

To parse that out a bit more:

$$Euclidean = \sqrt{(3.5 - 2)^2 + (2 - 3.5)^2 + (5 - 2)^2 + (1.5 - 3.5)^2 + (2 - 3)^2}$$

$$= \sqrt{1.5^2 + -1.5^2 + 3^2 + -2^2 + 1^2}$$

$$= \sqrt{18.5} = 4.3$$

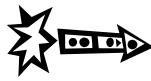
	Angelica	Bill	Chan	Dan	Hailey	Jordyn	Sam	Veronica
Blues Traveler	3.5	2	5	3	-	-	5	3
Broken Bells	2	3.5	1	4	4	4.5	2	-
Deadmau5	-	4	1	4.5	1	4	-	-
Norah Jones	4.5	-	3	-	4	5	3	5
Phoenix	5	2	5	3	-	5	5	4
Slightly Stoopid	1.5	3.5	1	4.5	-	4.5	4	2.5
The Strokes	2.5	-	-	4	4	4	5	3
Vampire Weekend	2	3	-	2	1	4	-	-



sharpen your pencil

Compute the Euclidean Distance between Hailey and Veronica.

Compute the Euclidean Distance between Hailey and Jordyn



sharpen your pencil - solution

Compute the Euclidean Distance between Hailey and Veronica.

$$= \sqrt{(4-5)^2 + (4-3)^2} = \sqrt{1+1} = \sqrt{2} = 1.414$$

Compute the Euclidean Distance between Hailey and Jordyn

$$= \sqrt{(4-4.5)^2 + (1-4)^2 + (4-5)^2 + (4-4)^2 + (1-4)^2}$$

$$= \sqrt{(-0.5)^2 + (-3)^2 + (-1)^2 + (0)^2 + (-3)^2}$$

$$= \sqrt{.25 + 9 + 1 + 0 + 9} = \sqrt{19.25} = 4.387$$

A Flaw

It looks like we discovered a flaw with using these distance measures. When we computed the distance between Hailey and Veronica, we noticed they only rated two bands in common (Norah Jones and The Strokes), whereas when we computed the distance between Hailey and Jordyn, we noticed they rated five bands in common. This seems to skew our distance measurement, since the Hailey-Veronica distance is in 2 dimensions while the Hailey-Jordyn distance is in 5 dimensions. Manhattan Distance and Euclidean Distance work best when there are no missing values. Dealing with missing values is an

active area of scholarly research. Later in the book we will talk about how to deal with this problem. For now just be aware of the flaw as we continue our first exploration into building a recommendation system.

A generalization

We can generalize Manhattan Distance and Euclidean Distance to what is called the Minkowski Distance Metric:

$$d(x, y) = \left(\sum_{k=1}^n |x_k - y_k|^r \right)^{\frac{1}{r}}$$

When

- $r = 1$: The formula is Manhattan Distance.
- $r = 2$: The formula is Euclidean Distance
- $r = \infty$: Supremum Distance



Arghhhh Math!

When you see formulas like this in a book you have several options. One option is to see the formula--brain neurons fire that say *math formula*--and then you quickly skip over it to the next English bit. I have to admit that I was once a skipper. The other option is to see the formula, pause, and dissect it. Many times you'll find the formula quite understandable. Let's dissect it now. When $r = 1$ the formula reduces to Manhattan Distance:

$$d(x, y) = \sum_{k=1}^n |x_k - y_k|$$

So for the music example we have been using throughout the chapter, x and y represent two people and $d(x, y)$ represents the distance between them. n is the number of bands they both rated (both x and y rated that band). We've done that calculation a few pages back:

	Angelica	Bill	Difference
Blues Traveler	3.5	2	1.5
Broken Bells	2	3.5	1.5
Deadmau5	-	4	
Norah Jones	4.5	-	
Phoenix	5	2	3
Slightly Stoopid	1.5	3.5	2
The Strokes	2.5	-	-
Vampire Weekend	2	3	1
Manhattan Distance:			9

That difference column represents the absolute value of the difference and we sum those up to get 9.
When $r = 2$, we get the Euclidean distance:

$$ed(x, y) = \sqrt{\sum_{k=1}^n (x_k - y_k)^2}$$

Here is the scoop. The greater the r , the more a large difference in one dimension will influence the total difference.

Representing the data in Python (finally some coding)

There are several ways of representing the data in the table above using Python. I am going to use Python's dictionary (also called an associative array or hash table):

```
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}}
```

Remember,

All the code for the book is available at
www.guidetodatamining.com.

While you are there, please leave a comment
regarding errors in this book, confusing bits, or
(gasp) even positive feedback!

We can get the ratings of a particular user as follows:

```
>>> users["Veronica"]
{'The Strokes': 3.0, 'Slightly Stoopid': 2.5, 'Norah Jones': 5.0, 'Phoenix': 4.0,
'Blues Traveler': 3.0}
>>>
```

The code to compute Manhattan distance

I'd like to write a function that computes the Manhattan distance as follows:

```
def manhattan(rating1, rating2):
    """Computes the Manhattan distance. Both rating1 and rating2 are dictionaries
    of the form {'The Strokes': 3.0, 'Slightly Stoopid': 2.5}"""
    distance = 0
    commonRatings = False
    for key in rating1:
        if key in rating2:
            distance += abs(rating1[key] - rating2[key])
            commonRatings = True
    if commonRatings:
        return distance
    else:
        return -1 #Indicates no ratings in common
```

To test the function:

```
>>> manhattan(users['Hailey'], users['Veronica'])
2.0
>>> manhattan(users['Hailey'], users['Jordyn'])
1.5
>>>
```

Now a function to find the closest person (actually this returns a sorted list with the closest person first):

```

def computeNearestNeighbor(username, users):
    """creates a sorted list of users based on their distance to username"""
    distances = []
    for user in users:
        if user != username:
            distance = manhattan(users[user], users[username])
            distances.append((distance, user))
    # sort based on distance -- closest first
    distances.sort()
    return distances

```

And just a quick test of that function:

```

>>> computeNearestNeighbor('Hailey', users)
[(2.0, 'Veronica'), (4.0, 'Chan'), (4.0, 'Sam'), (4.5, 'Dan'), (5.0, 'Angelica'),
(5.5, 'Bill'), (7.5, 'Jordyn')]
>>>

```

Finally, we are going to put this all together to make recommendations. Let's say I want to make recommendations for Hailey. I find her nearest neighbor—Veronica in this case. I will then find bands that Veronica has rated but Hailey has not. Also, I will assume that Hailey would have rated the bands the same as (or at least very similar to) Veronica. For example, Hailey has not rated the great band Phoenix. Veronica has rated Phoenix a '4' so we will assume Hailey is likely to enjoy the band as well. Here is my function to make recommendations.

```

def recommend(username, users):
    """Give list of recommendations"""
    # first find nearest neighbor
    nearest = computeNearestNeighbor(username, users)[0][1]
    recommendations = []
    # now find bands neighbor rated that user didn't
    neighborRatings = users[nearest]
    userRatings = users[username]
    for artist in neighborRatings:
        if not artist in userRatings:
            recommendations.append((artist, neighborRatings[artist]))
    recommendations.sort(key=lambda artistTuple: artistTuple[1], reverse = True)
    return recommendations

```

And now to make recommendations for Hailey:

```

>>> recommend('Hailey', users)
[('Phoenix', 4.0), ('Blues Traveler', 3.0), ('Slightly Stoopid', 2.5)]

```

That fits with our expectations. As we saw above, Hailey's nearest neighbor was Veronica and Veronica gave Phoenix a '4'. Let's try a few more:

```
>>> recommend('Chan', users)
[('The Strokes', 4.0), ('Vampire Weekend', 1.0)]
>>> recommend('Sam', users)
[('Deadmau5', 1.0)]
```

We think Chan will like The Strokes and also predict that Sam will not like Deadmau5.

```
>>> recommend('Angelica', users)
[]
```

Hmm. For Angelica we got back an empty set meaning we have no recommendations for her. Let us see what went wrong:

```
>>> computeNearestNeighbor('Angelica', users)
[(3.5, 'Veronica'), (4.5, 'Chan'), (5.0, 'Hailey'), (8.0, 'Sam'), (9.0, 'Bill'),
(9.0, 'Dan'), (9.5, 'Jordyn')]
```

Angelica's nearest neighbor is Veronica. When we look at their ratings:

	Angelica	Bill	Chan	Dan	Hailey	Jordyn	Sam	Veronica
Blues Traveler	3.5	2	5	3	-	-	5	3
Broken Bells	2	3.5	1	4	4	4.5	2	-
Deadmau5	-	4	1	4.5	1	4	-	-
Norah Jones	4.5	-	3	-	4	5	3	5
Phoenix	5	2	5	3	-	5	5	4
Slightly Stoopid	1.5	3.5	1	4.5	-	4.5	4	2.5
The Strokes	2.5	-	-	4	4	4	5	3
Vampire Weekend	2	3	-	2	1	4	-	-

We see that Angelica rated every band that Veronica did. We have no new ratings, so no recommendations.

Shortly, we will see how to improve the system to avoid these cases.



exercise

- 1) Implement the Minkowski Distance Function.
 - 2) Alter the computeNearestNeighbor Function to use Minkowski Distance.



exercise - solution

1) Implement the Minkowski Distance function.

```
def minkowski(rating1, rating2, r):
    """Computes the Minkowski distance.
    Both rating1 and rating2 are dictionaries of the form
    {'The Strokes': 3.0, 'Slightly Stoopid': 2.5}"""
    distance = 0
    commonRatings = False
    for key in rating1:
        if key in rating2:
            distance +=
                pow(abs(rating1[key] - rating2[key]), r)
            commonRatings = True
    if commonRatings:
        return pow(distance, 1/r)
    else:
        return -1 #Indicates no ratings in common
```

2) Alter the computeNearestNeighbor Function to use Minkowski Distance.

just need to alter the distance = line to

```
distance = minkowski(users[user], users[username], 2)
```

(the 2 as the r argument implements Euclidean)

Blame the users

Let's take a look at the user ratings in a bit more detail. We see that users have very different behaviors when it comes to rating bands

Bill seems to avoid the extremes. His ratings range from 2 to 4

Jordyn seems to like everything. Her ratings range from 4 to 5.

	Angelica	Bill	Chan	Dan	Hailey	Jordyn	Sam	Veronica
Blues Traveler	3.5	2	5	3	-	-	5	3
Broken Bells	2	3.5	1	4	4	4.5	2	-
Deadmau5	-	4	1	4.5	1	4	-	-
Norah Jones	4.5	-	3	-	4	5	3	5
Phoenix	5	2	5	3	-	5	5	4
Slightly Stoopid	1.5	3.5	1	4.5	-	4.5	4	2.5
The Strokes	2.5	-	-	4	4	4	5	3
Vampire Weekend	2	3	-	2	1	4	-	-

Hailey is a binary person giving either 1s or 4s to bands.

So how do we compare, for example, Hailey to Jordan? Does Hailey's '4' mean the same as Jordyn's '4' or Jordyn's '5'? I would guess it is more like Jordyn's '5'. This variability can create problems with a recommendation system.

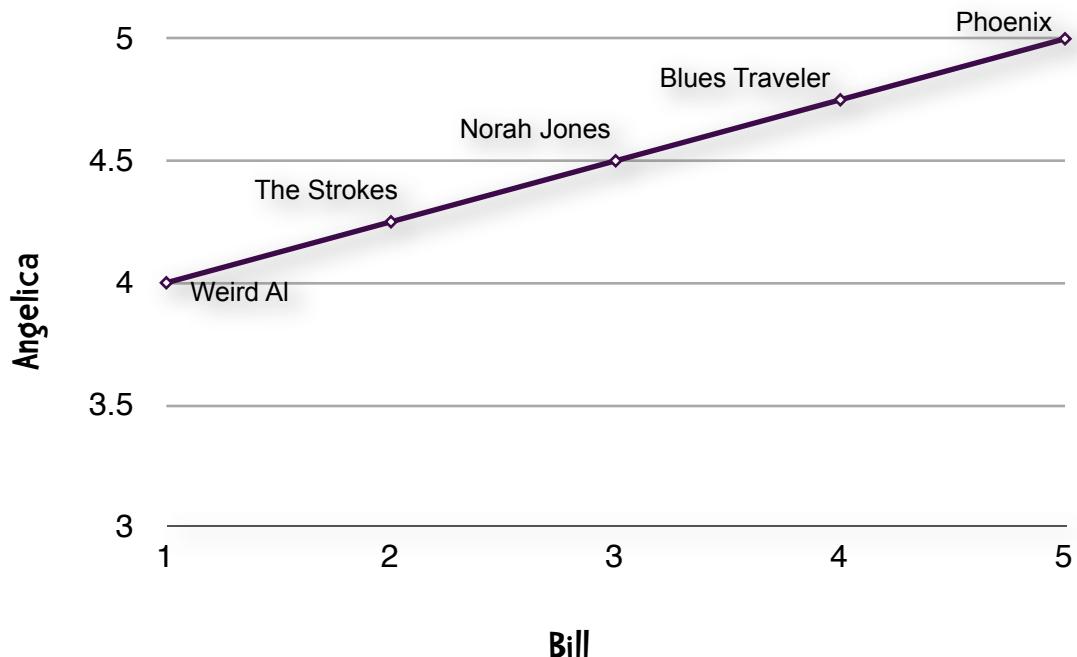


Pearson Correlation Coefficient

One way to fix this problem is to use the Pearson Correlation Coefficient. First, the general idea. Consider the following data:

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

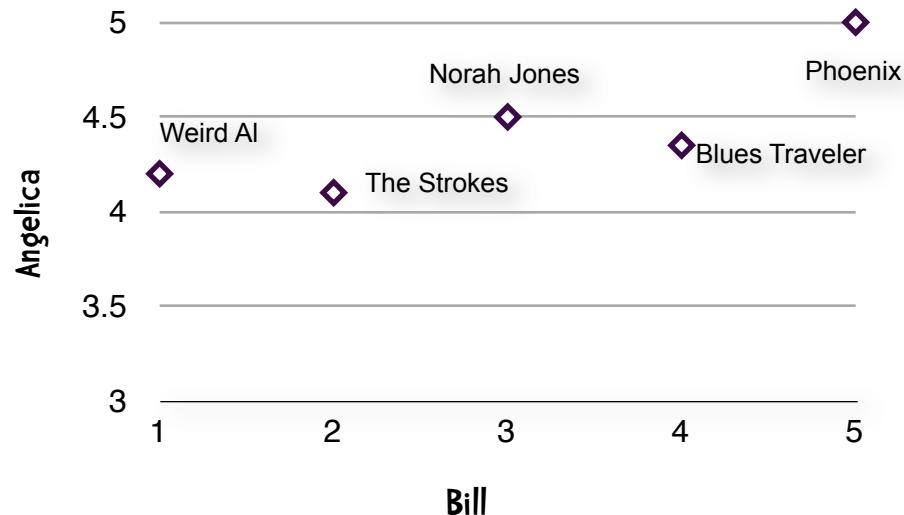
This is an example of what is called 'grade inflation' in the data mining community. Angelica's lowest rating is 4—all her ratings are between 4 and 5. If we are to graph this chart it would look like



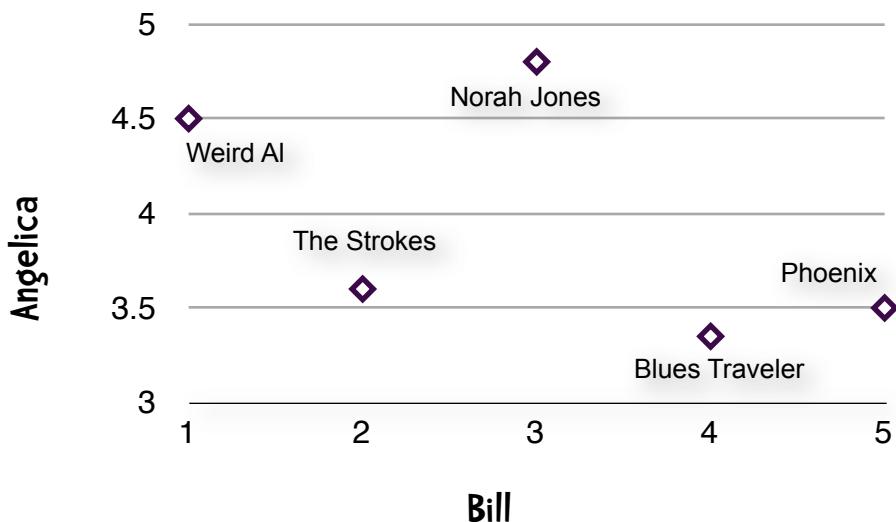
Perfect Agreement

The fact that this is a **straight line** indicates a perfect agreement between Angelica and Bill. Both Angelica and Bill rated Phoenix as the best band, Blues Traveler next, Norah Jones after that, and so on. As Angelica and Bill agree less, the less the data points reside on a straight line:

Pretty Good Agreement:



Not So Good Agreement:



So chart-wise perfect agreement is indicated by a straight line. The Pearson Correlation Coefficient is a measure of correlation between two variables (in this specific case the correlation between Angelica and Bill). It ranges between -1 and 1 inclusive. 1 indicates perfect agreement. -1 indicates perfect disagreement. To give you a general feel for this, the chart above with the straight line has a Pearson of 1, the chart above that I labelled ‘pretty good agreement’ has a Pearson of 0.91, and the ‘not so good agreement’ chart has a Pearson of 0.81 So we can use this to find the individual who is most similar to the person we are interested in.

The formula for the Pearson Correlation Coefficient is

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$



Arghhhh Math Again!

Here's a personal confession. I have a Bachelor of Fine Arts degree in music. While I have taken courses in ballet, modern dance, and costume design, I did not have a single math course as an undergrad. Either due to this background or some innate wiring in my brain, when I read a book that has formulas like the one above, I tend to skip over the formulas and continue with the text below them. If you are like me I would urge you to fight that urge and actually look at the formula. Many formulas that on a quick glimpse look complex are actually understandable by mere mortals.

Other than perhaps looking complex, the problem with the formula above is that the algorithm to implement it would require multiple passes through the data. Fortunately for us algorithmic people, there is an alternative formula, which is an approximation of Pearson:

$$r = \frac{\sum_{i=1}^n x_i y_i - \frac{\sum_{i=1}^n x_i \sum_{i=1}^n y_i}{n}}{\sqrt{\sum_{i=1}^n x_i^2 - \frac{(\sum_{i=1}^n x_i)^2}{n}} \sqrt{\sum_{i=1}^n y_i^2 - \frac{(\sum_{i=1}^n y_i)^2}{n}}}$$

(Remember what I said two paragraphs above about not skipping over formulas) This formula, in addition to looking initially horribly complex is, more importantly, numerically unstable meaning that what might be a small error is amplified by this reformulation. The big plus is that we can implement it using a single-pass algorithm, which we will get to shortly. First, let's dissect this formula and work through the example we saw a few pages back:

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

To start with, let us compute

$$\sum_{i=1}^n x_i y_i$$

Here the x and y represent Angelica and Bill. For each band we are going to multiple Angelica's and Bill's rating together and sum the results:

$$(4.75 * 4) + (4.5 * 3) + (5 * 5) + (4.25 * 2) + (4 * 1)$$

$$19 + 13.5 + 25 + 8.5 + 4 = 70$$

Sweet! Now let's compute

$$\frac{\sum_{i=1}^n x_i \sum_{i=1}^n y_i}{n}$$

So the sum of Angelica's ratings is 22.5 and the sum of Bill's is 15. And they rated 5 bands

$$\frac{22.5*15}{5} = 67.5$$

So the numerator in the formula on page 23 is $70 - 67.5 = 2.5$

Now let's dissect the denominator.

$$\sqrt{\sum_{i=1}^n x_i^2 - \frac{(\sum_{i=1}^n x_i)^2}{n}}$$

First,

$$\sum_{i=1}^n x_i^2 = (4.75)^2 + (4.5)^2 + (5)^2 + (4.25)^2 + (4)^2 = 101.875$$

We've already computed the sum of Angelica's ratings, which is 22.5. Square that and we get 506.25. We divide that by the number of co-rated bands (5) and we get 101.25.

$$\sqrt{101.875 - 101.25} = \sqrt{.625} = .79057$$

Next we do the same computation for Ben.

$$\sqrt{\sum_{i=1}^n y_i^2 - \frac{(\sum_{i=1}^n y_i)^2}{n}} = \sqrt{55 - 45} = 3.162277$$

Putting this all together we get:

$$r = \frac{2.5}{.79057(3.162277)} = \frac{2.5}{2.5} = 1.00$$



exercise

Before going to the next page, implement the algorithm in Python. You should get the following results.

```
>>> pearson(users['Angelica'], users['Bill'])
-0.90405349906826993
>>> pearson(users['Angelica'], users['Hailey'])
0.42008402520840293
>>> pearson(users['Angelica'], users['Jordyn'])
0.76397486054754316
>>>
```

For this implementation you will need 2 Python functions `sqrt` (square root) and `pow` (power):

```
>>> from math import sqrt
>>> sqrt(9)
3.0
>>> pow(3,2)
9
```



exercise - solution

Here is my implementation of Pearson

```
def pearson(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)
    # 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
```

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

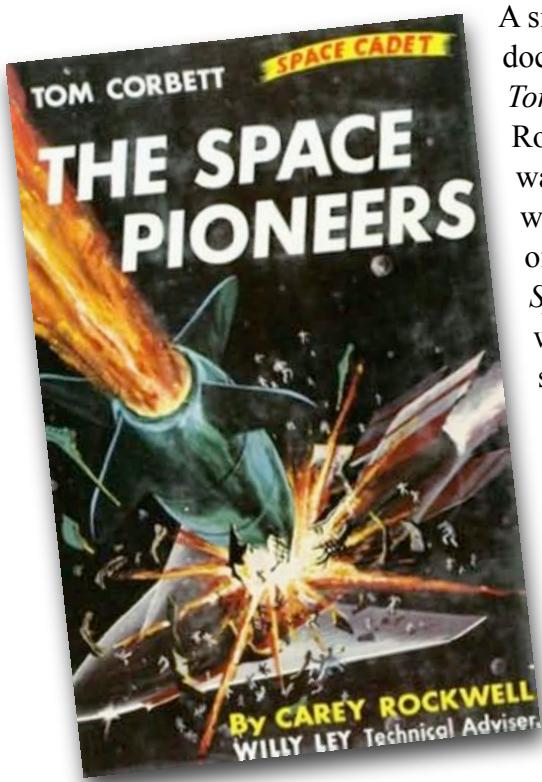
Just by eye-balling the above chart (and by using any of the distance formulas mentioned above) we can see that Sally is more similar in listening habits to Ann than Ben is.

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
Λ TOMS 2008	4:18	Sara Bareilles	Little Voice	Folk	18
Λ Immigraniada (We Comin' Rougher)	3:48	Gogol Bordello	Trans-Continental...	Alternative...	18
Λ Rebellious Love	3:23	Gogol Bordello	Trans-Continental...	Alternative...	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 *Tom 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

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

where \cdot indicates the dot product and $\|x\|$ indicates the length of the vector x . The length of a vector is

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

For example, suppose my two vectors are

$$\begin{aligned}x &= (4, 1, 3, 5, 2) \\y &= (5, 2, 4, 5, 1)\end{aligned}$$

then

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

$$\|y\| = \sqrt{5^2 + 2^2 + 4^2 + 5^2 + 1^2} = \sqrt{71} = 8.426$$

The dot product is

$$x \cdot y = 4 * 5 + 1 * 2 + 3 * 4 + 5 * 5 + 2 * 1 = 61$$

And the cosine

similarity is

$$\cos(x, y) = \frac{61}{7.416 * 8.426} = \frac{61}{62.4899} = 0.976$$

The cosine similarity rating ranges from 1 indicated perfect similarity to -1 indicate perfect negative similarity.

Which similarity measure to use?

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

- If your data is dense (almost all attributes have non-zero values) and the magnitude of the attribute values is important, use distance measures such as Euclidean.
- If the data is subject to grade-inflation (different users may be using different scales) use Pearson.
- If the data is sparse consider using 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.

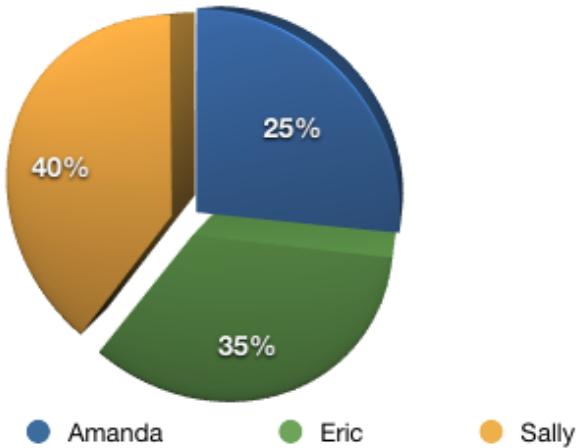
K-nearest neighbor

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

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 $0.7 / 2$ 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

To determine how well we think Ann will like The Grey Wardens we multiply each person's rating by their influence and sum those values up:

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

$$\begin{aligned}\text{Projected rating} &= (4.5 \times 0.25) + (5 \times 0.35) + (3.5 \times 0.4) \\ &= 4.275\end{aligned}$$



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

$$= (3.5 \times (0.8 / 1.5)) + (5 \times (0.7 / 1.5))$$

$$= (3.5 \times .5333) + (5 \times 1.867)$$

$$= 1.867 + 2.333$$

$$= 4.2$$

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.guiddetodatamining.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, prod_id):
    """Given product id number return product name"""
    if prod_id in self.productid2name:
        return self.productid2name[prod_id]
    else:
        return prod_id

def userRatings(self, uid, n):
    """Return n top ratings for user with id"""
    print ("Ratings for " + self.userid2name[uid])
    ratings = self.data[uid]
    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]
        #
        # 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]

```

Examples of this Python 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 CVS 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-SQL-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), ('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 ROMANOV'S RUSSIAN / ENGLISH DICTIONARY by A.C.
Pomahoba    8
Narcissus in Chains (Anita Blake Vampire Hunter (Paperback)) by Laurell K. Hamilton   6
China Men (Vintage International (Paperback)) by MAXINE HONG KINGSTON    0
Wuthering Heights (Penguin Classics) by Emily Bronte 0
The Faber Storybook by Kathleen Lines  0
```

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.