Similarity Matrix

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

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

And we will be using the normalized ratings!

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

Imagine Dragons Daft Punk Lorde Fall Out Boy

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

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

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

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

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

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

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

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

Slope One

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

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

	PSY	Whitney Houston
Amy	3	4
Ben	4	?

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

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

The Broad Brush Picture

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

Part 2
Use deviations to make predictions

Part 1: Computing deviation

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

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

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

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

where $card(S)$ is how many elements are in S and X is the entire set of all ratings. So

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

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

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

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

sharpen your pencil

Compute the rest of the values in this table:

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

sharpen your pencil - solution

Compute the rest of the values in this table:

Taylor Swift with respect to Whitney Houston:

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

PSY with respect to Whitney Houston:

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

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

brain calisthenics

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

(answer on next page)

brain calisthenics

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

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

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

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

Part 2: Making predictions with Weighted Slope One

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

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

where

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

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

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

Let's dissect the numerator.

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

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

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

Let's step through this:

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

	Taylor Swift	PSY	Whitney Houston
Ben	5	2	?

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

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

DENOMINATOR

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

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

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

Putting this into Python

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

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

First computing the deviations.

Again, the formula for computing deviations is

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

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

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

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

The pseudoCode for our function could be

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

That pseudocode looks pretty nice but as you can see, there is a disconnect between the data format expected by the pseudocode and the format the data is really in (see users2 above as an example). As code warriors we have two possibilities, either alter the format of the data, or revise the psuedocode. I am going to opt for the second approach. This revised pseudocode looks like

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

Let's construct the method step-by-step

Step 1:

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

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

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

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

Step 2

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

In the recommender class init method I initialized self.frequencies and self.deviations to be dictionaries.

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

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

The Python dictionary method setdefault takes 2 arguments: a key and an initialValue. This method does the following. If the key does not exist in the dictionary it is added to the dictionary with the value initialValue. Otherwise it returns the current value of the key.

Step 3

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

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

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

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

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

Step 4:

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

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

   for (item, ratings) in self.deviations.items():
      for item2 in ratings:
         ratings[item2] /= self.frequencies[item][item2]
```

That's it! Even with comments we implemented

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

in only 18 lines of code. Incredible!

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

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

I get

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

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

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

Shout out to Bryan O'Sullivan and his blog teideal glic deisbhéalach (serpentine.com/blog) which presented a Python implementation of Slope One! The code presented here is based on his work.

Weighted Slope 1: The recommendation component

Now it is time to code the recommendation component:

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

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

sharpen your pencil

The formula in pseudo English:

sharpen your pencil - a solution

Here's my version of the formula:

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

{"Taylor Swift": 5, "PSY": 2}

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

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

And here is my conversion of that to Python:

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

      recommendations =  [(self.convertProductID2name(k),
                           v / frequencies[k])
                          for (k, v) in recommendations.items()]

      # finally sort and return
      recommendations.sort(key=lambda artistTuple: artistTuple[1],
                           reverse = True)
      return recommendations
```

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

```
>>> r = recommender(users2)
>>> r.computeDeviations()
>>> g = users2['Ben']
>>> r.slopeOneRecommendations(g)
[('Whitney Houston', 3.375)]
```

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

MovieLens data set

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

Let's give it a try.

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

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

```
>>> r = recommender(0)
>>> r.loadMovieLens('/Users/raz/Downloads/ml-100k/')
102625
```

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

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

```
Maya Lin: A Strong Clear Vision (1994)   5
Mighty Aphrodite (1995) 5
Bound (1996)      5
Full Monty, The (1997)  5
Chasing Amy (1997)      5
Ridicule (1996)   5
Nightmare Before Christmas, The (1993)   5
Three Colors: Red (1994)      5
Professional, The (1994)      5
Priest (1994)     5
...
```

User 1 rated all these movies a '5'!

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

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

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

Finally, let's get recommendations for User 1:

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

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

and user 25:

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

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

Projects

1. See how well the Slope One recommender recommends movies for you. Rate 10 movies or so (ones that are in the MovieLens dataset). Does the recommender suggest movies you might like?

2. Implement Adjusted Cosine Similarity. Compare its performance to Slope One.

3. (harder) I run out of memory (I have 8GB on my desktop) when I try to run this on the Book Crossing Dataset. Recall that there are 270,000 books that are rated. So we would need a 270,000 x 270,000 dictionary to store the deviations. That's roughly 73 billion dictionary entries. How sparse is this dictionary for the MovieLens dataset? Alter the code so we can handle larger datasets.

Congratulations on finishing chapter 3!!

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

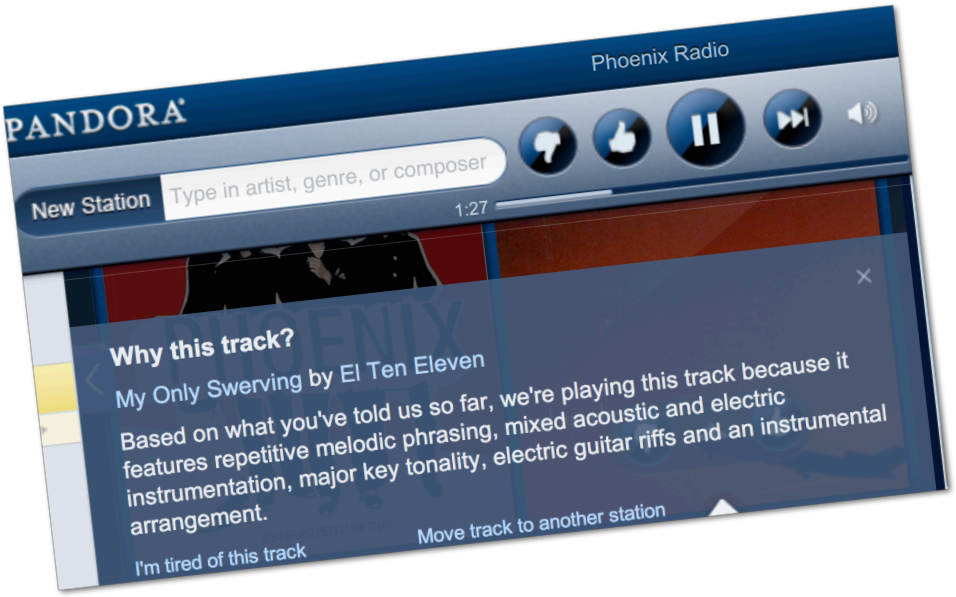
Classification based on item attributes

In the previous chapters we talked about making recommendations by collaborative filtering (also called *social filtering*). In collaborative filtering we harness the power of a community of people to help us make recommendations. You buy Wolfgang Amadeus Phoenix. We know that many of our customers who bought that album also bought Contra by Vampire Weekend. So we recommend that album to you. I watch an episode of Doctor Who and Netflix recommends Quantum Leap because many people who watched Doctor Who also watched Quantum Leap. In previous chapters we talked about some of the difficulties of collaborative filtering including problems with data sparsity and scalability. Another problem is that recommendation systems based on collaborative filtering tend to recommend already popular items—there is a bias toward popularity. As an extreme case, consider a debut album by a brand new band. Since that band and album have never been rated by anyone (or purchased by anyone since it is brand new), it will never be recommended.

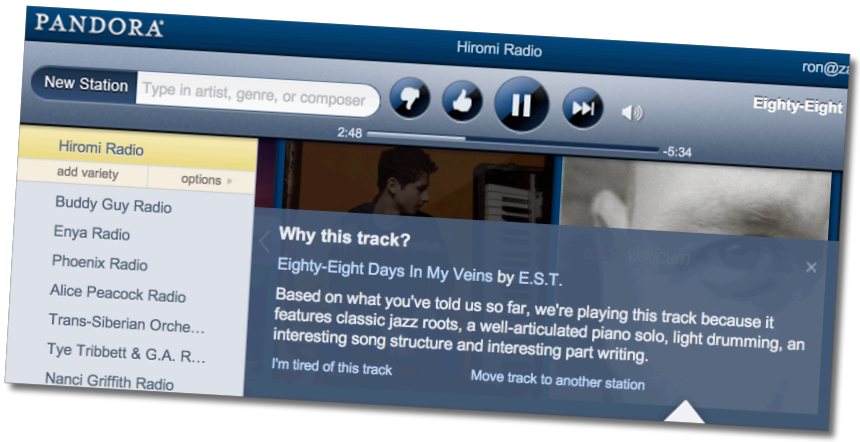
"These recommenders can create a rich-get-richer effect for popular products and vice-versa for unpopular ones"

Daniel Fleder & Kartik Hosanagar. 2009. "Blockbusters Culture's Next Rise or Fall: The Impact of Recommender Systems on Sales Diversity" Management Science vol 55

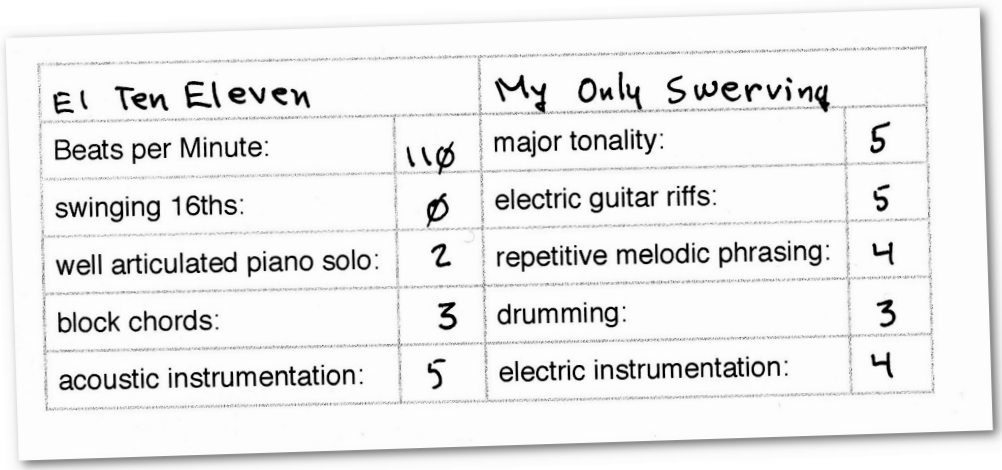
In this chapter we look at a different approach. Consider the streaming music site, Pandora. In Pandora, as many of you know, you can set up different streaming radio stations. You seed each station with an artist and Pandora will play music that is similar to that artist. I can create a station seeded with the band Phoenix. It then plays bands it thinks are similar to Phoenix—for example, it plays a tune by El Ten Eleven. It doesn't do this with collaborative filtering—because people who listened to Phoenix also listened to the El Ten Eleven. It plays El Ten Eleven because the algorithm believes that El Ten Eleven is musically similar to Phoenix. In fact, we can ask Pandora why it played a tune by the group:



It plays El Ten Eleven's tune *My Only Swerving* on the Phoenix station because "Based on what you told us so far, we're playing this track because it features repetitive melodic phrasing, mixed acoustic and electric instrumentation, major key tonality, electric guitar riffs and an instrumental arrangement." On my Hiromi station it plays a tune by E.S.T. because "it features classic jazz roots, a well-articulated piano solo, light drumming, an interesting song structure and interesting part writing."



Pandora bases its recommendation on what it calls The Music Genome Project. They hire professional musicians with a solid background in music theory as analysts who determine the features (they call them 'genes') of a song. These analysts are given over 150 hours of training. Once trained they spend an average of 20-30 minutes analyzing a song to determine its genes/features. Many of these genes are technical



The analyst provides values for over 400 genes. Its a very labor intensive process and approximately 15,000 new songs are added per month.

> NOTE: The Pandora algorithms are proprietary and I have no knowledge as to how they work. What follows is not a description of how Pandora works but rather an explanation of how to construct a similar system.

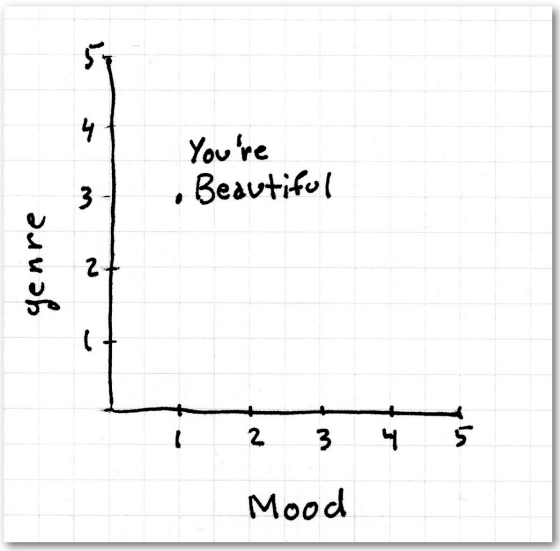
The importance of selecting appropriate values

Consider two genes that Pandora may have used: genre and mood. The values of these might look like this:

genre	
Country	1
Jazz	2
Rock	3
Soul	4
Rap	5

Mood	
Melancholy	1
joyful	2
passion	3
angry	4
unknown	5

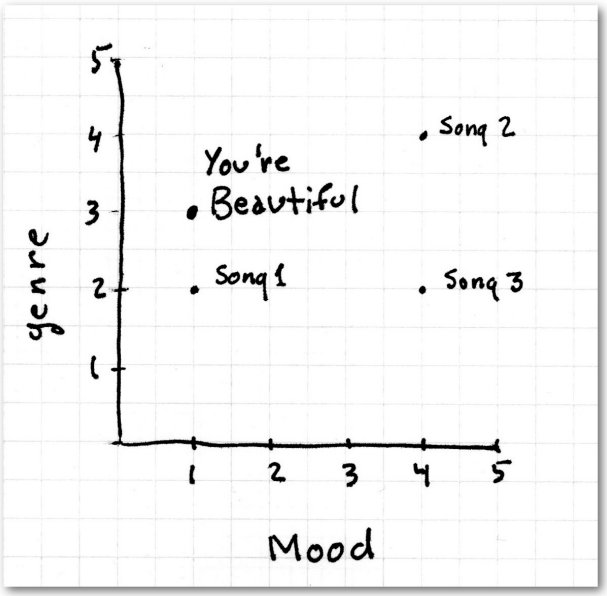
So a genre value of 4 means 'Soul' and a mood value of 3 means 'passion'. Suppose I have a rock song that is melancholy—for example the gag-inducing *You're Beautiful* by James Blunt. In 2D space, inked quickly on paper, that would look as follows:



FACT:
In a Rolling Stone poll on the Most Annoying Songs ever, You're Beautiful placed #7!

Let's say Tex just absolutely loves You're Beautiful and we would like to recommend a song to him.

Let me populate our dataset with more songs. Song 1 is a jazz song that is melancholy; Song 2 is a soul song that is angry and Song 3 is a jazz song that is angry. Which would you recommend to Tex?



Song 1 looks closest!

I hope you see that we have a fatal flaw in our scheme. Let's take a look at the possible values for our variables again:

Mood	
melancholy	1
joyful	2
passion	3
angry	4
unknown	5

genre	
Country	1
Jazz	2
Rock	3
Soul	4
Rap	5

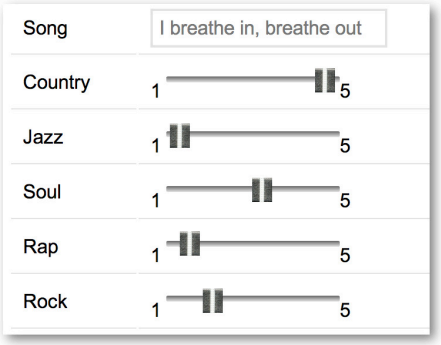
If we are trying to use any distance metrics with this scheme we are saying that jazz is closer to rock than it is to soul (the distance between jazz and rock is 1 and the distance between

jazz and soul is 2). Or melancholy is closer to joyful than it is to angry. Even when we rearrange values the problem remains.

Mood	
melancholy	1
angry	2
passion	3
joyful	4
unknown	5

genre	
Country	1
Jazz	2
Soul	3
Rap	4
Rock	5

Re-ordering does not solve the problem. No matter how we rearrange the values this won't work. This shows us that we have chosen our features poorly. We want features where the values fall along a meaningful scale. We can easily fix our genre feature by dividing it into 5 separate features—one for country, another for jazz, etc.



They all can be on a 1-5 scale—how 'country' is the sound of this track—'1' means no hint of country to '5' means this is a solid country sound. Now the scale does mean something. If we are trying to find a song similar to one that rated a country value of '5', a song that rated a country of '4' would be closer than one of a '1'.

This is exactly how Pandora constructs its gene set. The values of most genes are on a scale of 1-5 with ½ integer increments. Genes are arranged into categories. For example, there is a musical qualities category which contains genes for Blues Rock Qualities, Folk Rock Qualities, and Pop Rock Qualities among others. Another category is instruments with genes such as Accordion, Dirty Electric Guitar Riffs and Use of Dirty Sounding Organs. Using these genes, each of which has a well-defined set of values from 1 to 5, Pandora represents each song as a vector of 400 numeric values (each song is a point in a 400 dimensional space). Now Pandora can make recommendations (that is, decide to play a song on a user-defined radio station) based on standard distance functions like those we already have seen.

A simple example

Let us create a simple dataset so we can explore this approach. Suppose we have seven features each one ranging from 1-5 in ½ integer increments (I admit this isn't a very rational nor complete selection):

Amount of piano	1 indicates lack of piano; 5 indicates piano throughout and featured prominently
Amount of vocals	1 indicates lack of vocals; 5 indicates prominent vocals throughout song.
Driving beat	Combination of constant tempo, and how the drums & bass drive the beat.
Blues Influence	
Presence of dirty electric guitar	
Presence of backup vocals	
Rap Influence	

Now, using those features I rate ten tunes:

	Piano	Vocals	Driving beat	Blues infl.	Dirty elec. Guitar	Backup vocals	Rap infl.
Dr. Dog/ Fate	2.5	4	3.5	3	5	4	1
Phoenix/ Lisztomania	2	5	5	3	2	1	1
Heartless Bastards / Out at Sea	1	5	4	2	4	1	1
Todd Snider/ Don't Tempt Me	4	5	4	4	1	5	1
The Black Keys/ Magic Potion	1	4	5	3.5	5	1	1
Glee Cast/ Jessie's Girl	1	5	3.5	3	4	5	1
Black Eyed Peas/ Rock that Body	2	5	5	1	2	2	4
La Roux/ Bulletproof	5	5	4	2	1	1	1
Mike Posner/ Cooler than me	2.5	4	4	1	1	1	1
Lady Gaga/ Alejandro	1	5	3	2	1	2	1

Thus, each tune is represented as a list of numbers and we can use any distance function to compute the distance between tunes. For example, The Manhattan Distance between Dr. Dog's *Fate* and Phoenix's *Lisztomania* is:

Dr. Dog/ Fate	2.5	4	3.5	3	5	4	1
Phoenix/ Lisztomania	2	5	5	3	2	1	1
Distance	**0.5**	**1**	**1.5**	**0**	**3**	**3**	**0**

summing those distances gives us a Manhattan Distance of 9.

sharpen your pencil

I am trying to find out what tune is closest to Glee's rendition of Jessie's Girl using **Euclidean Distance**. Can you finish the following table and determine what group is closest?

	distance to Glee's Jessie's Girl
Dr. Dog/ Fate	??
Phoenix/ Lisztomania	4.822
Heartless Bastards / Out at Sea	4.153
Todd Snider/ Don't Tempt Me	4.387
The Black Keys/ Magic Potion	4.528
Glee Cast/ Jessie's Girl	0
Black Eyed Peas/ Rock that Body	5.408
La Roux/ Bulletproof	6.500
Mike Posner/ Cooler than me	5.701
Lady Gaga/ Alejandro	??

sharpen your pencil - solution

	distance to Glee's Jessie's Girl
Dr. Dog/ Fate	2.291
Lady Gaga/ Alejandro	4.387

Recall that the Euclidean Distance between any two objects, x and y, which have n attributes is:

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

So the Euclidean Distance between Glee and Lady Gaga

	piano	vocals	beat	blues	guitar	backup	rap	SUM	SQRT
Glee	1	5	3.5	3	4	5	1		
Lady G	1	5	3	2	1	2	1		
(x-y)	0	0	0.5	1	3	3	0		
$(x-y)^2$	0	0	0.25	1	9	9	0	19.25	4.387

Doing it Python Style!

Recall that our data for social filtering was of the format:

```
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}}
```

We can represent this current data in a similar way:

```
music = {"Dr Dog/Fate": {"piano": 2.5, "vocals": 4, "beat": 3.5,
                         "blues": 3, "guitar": 5, "backup vocals": 4,
                         "rap": 1},
         "Phoenix/Lisztomania": {"piano": 2, "vocals": 5, "beat": 5,
                                 "blues": 3, "guitar": 2,
                                 "backup vocals": 1, "rap": 1},
         "Heartless Bastards/Out at Sea": {"piano": 1, "vocals": 5,
                                           "beat": 4, "blues": 2,
                                           "guitar": 4,
                                           "backup vocals": 1,
                                           "rap": 1},
         "Todd Snider/Don't Tempt Me": {"piano": 4, "vocals": 5,
                                        "beat": 4, "blues": 4,
                                        "guitar": 1,
                                        "backup vocals": 5, "rap": 1},
         "The Black Keys/Magic Potion":{"piano": 1, "vocals": 4,
                                           "beat": 5, "blues": 3.5,
                                           "guitar": 5,
                                           "backup vocals": 1,
                                           "rap": 1},
         "Glee Cast/Jessie's Girl": {"piano": 1, "vocals": 5,
                                     "beat": 3.5, "blues": 3,
                                     "guitar":4, "backup vocals": 5,
                                     "rap": 1},
         "La Roux/Bulletproof": {"piano": 5, "vocals": 5, "beat": 4,
```

```
                          "blues": 2, "guitar": 1,
                          "backup vocals": 1, "rap": 1},
        "Mike Posner": {"piano": 2.5, "vocals": 4, "beat": 4,
                        "blues": 1, "guitar": 1, "backup vocals": 1,
                        "rap": 1},
        "Black Eyed Peas/Rock That Body": {"piano": 2, "vocals": 5,
                                           "beat": 5, "blues": 1,
                                           "guitar": 2,
                                           "backup vocals": 2,
                                           "rap": 4},
        "Lady Gaga/Alejandro": {"piano": 1, "vocals": 5, "beat": 3,
                          "blues": 2, "guitar": 1,
                          "backup vocals": 2, "rap": 1}}
```

Now suppose I have a friend who says he likes the Black Keys Magic Potion. I can plug that into my handy Manhattan distance function:

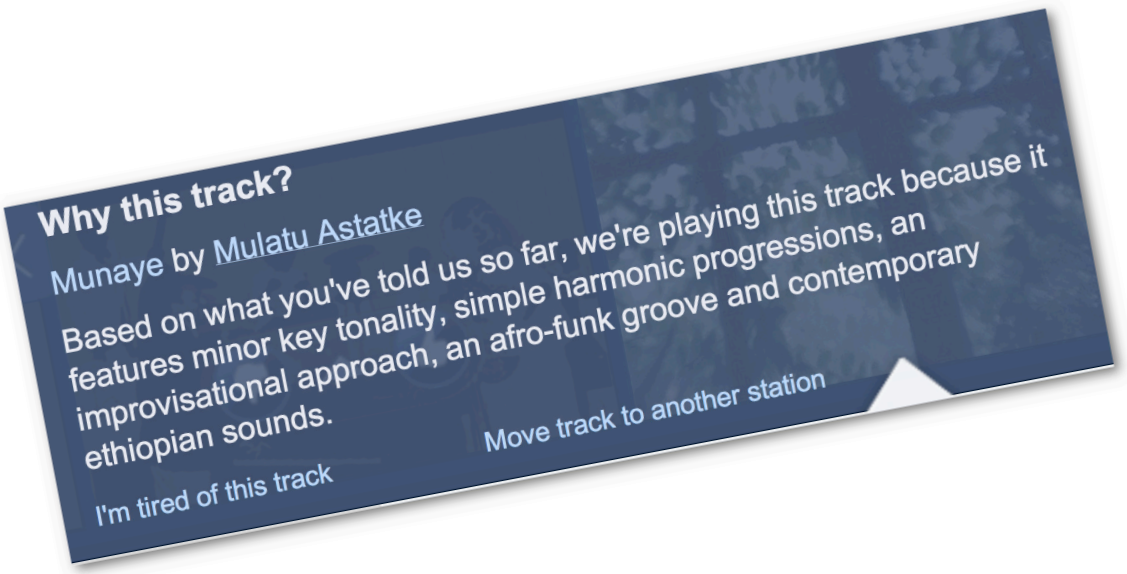
```
>>> computeNearestNeighbor('The Black Keys/Magic Potion', music)

[(4.5, 'Heartless Bastards/Out at Sea'), (5.5, 'Phoenix/Lisztomania'),
(6.5, 'Dr Dog/Fate'), (8.0, "Glee Cast/Jessie's Girl"), (9.0, 'Mike
Posner'), (9.5, 'Lady Gaga/Alejandro'), (11.5, 'Black Eyed Peas/Rock
That Body'), (11.5, 'La Roux/Bulletproof'), (13.5, "Todd Snider/Don't
Tempt Me")]
```

and I can recommend to him Heartless Bastard's Out at Sea. This is actually a pretty good recommendation.

> NOTE:
> The code for this example, as well as all examples in this book, is available on the book website
> http://www.guidetodatamining.com

Answering the question "Why?"

When Pandora recommends something it explains why you might like it:



We can do the same. Remember our friend who liked The Black Keys Magic Potion and we recommended Heartless Bastards Out at Sea. What features influenced that recommendation? We can compare the two feature vectors:

	Piano	Vocals	Driving beat	Blues infl.	Dirty elec. Guitar	Backup vocals	Rap infl.
Black Keys Magic Potion	1	5	4	2	4	1	1
Heartless Bastards Out at Sea	1	4	5	3.5	5	1	1
difference	0	1	1	1.5	1	0	0

The features that are closest between the two tunes are piano, presence of backup vocals, and rap influence—they all have a distance of zero. However, all are on the low end of the scale: no piano, no presence of backup vocals, and no rap influence and it probably would not be helpful to say "We think you would like this tune because it lacks backup vocals." Instead, we will focus on what the tunes have in common on the high end of the scale.

We think you might like Heartless Bastards Out at Sea because it has a driving beat and features vocals and dirty electric guitar.

Because our data set has few features, and is not well-balanced, the other recommendations are not as compelling:

```
>>> computeNearestNeighbor("Phoenix/Lisztomania", music)

[(5, 'Heartless Bastards/Out at Sea'), (5.5, 'Mike Posner'), (5.5, 'The
Black Keys/Magic Potion'), (6, 'Black Eyed Peas/Rock That Body'), (6,
'La Roux/Bulletproof'), (6, 'Lady Gaga/Alejandro'), (8.5, "Glee Cast/
Jessie's Girl"), (9.0, 'Dr Dog/Fate'), (9, "Todd Snider/Don't Tempt
Me")]

>>> computeNearestNeighbor("Lady Gaga/Alejandro", music)

[(5, 'Heartless Bastards/Out at Sea'), (5.5, 'Mike Posner'), (6, 'La
Roux/Bulletproof'), (6, 'Phoenix/Lisztomania'), (7.5, "Glee Cast/
Jessie's Girl"), (8, 'Black Eyed Peas/Rock That Body'), (9, "Todd
Snider/Don't Tempt Me"), (9.5, 'The Black Keys/Magic Potion'), (10.0,
'Dr Dog/Fate')]
```

That Lady Gaga recommendation is particularly bad.

A problem of *scale*

Suppose I want to add another feature to my set. This time I will add beats per minute (or bpm). This makes some sense—I might like fast beat songs or slow ballads. Now my data would look like this:

	Piano	Vocals	Driving beat	Blues infl.	Dirty elec. Guitar	Backup vocals	Rap infl.	bpm
Dr. Dog/ Fate	2.5	4	3.5	3	5	4	1	140
Phoenix/ Lisztomania	2	5	5	3	2	1	1	110
Heartless Bastards / Out at Sea	1	5	4	2	4	1	1	130
The Black Keys/ Magic Potion	1	4	5	3.5	5	1	1	88
Glee Cast/ Jessie's Girl	1	5	3.5	3	4	5	1	120
Bad Plus/ Smells like Teen Spirit	5	1	2	1	1	1	1	90

Without using beats per minute, the closest match to The Black Keys' Magic Potion is Heartless Bastards' Out to Sea and the tune furthest away is Bad Plus's version of Smells Like Teen Spirit. However, once we add beats per minute, it wrecks havoc with our distance function—bpm dominates the calculation. Now Bad Plus is closest to The Black Keys simply because the bpm of the two tunes are close.

Consider another example. Suppose I have a dating site and I have the weird idea that the best attributes to match people up by are salary and age.

gals		
name	age	salary
Yun L	35	75,000
Allie C	52	55,000
Daniela C	27	45,000
Rita A	37	115,000

guys		
name	age	salary
Brian A	53	70,000
Abdullah K	25	105,000
David A	35	69,000
Michael W	48	43,000

Here the scale for age ranges from 25 to 53 for a difference of 28 and the salary scale ranges from 43,000 to 115,000 for a difference of 72,000. Because these scales are so different, salary dominates any distance calculation. If we just tried to eyeball matches we might recommend David to Yun since they are the same age and their salaries are fairly close. However, if we went by any of the distance formulas we have covered, 53-year old Brian would be the person recommended to Yun. This does not look good for my fledgling dating site.

In fact, this difference in scale among attributes is a **big problem** for any recommendation system.

Arghhhh.

Normalization

No need to panic.

Relax.

The solution is normalization!

To remove this bias we need to standardize or normalize the data. One common method of normalization involves having the values of each feature range from 0 to 1.

For example, consider the salary attribute in our dating example. The minimum salary was 43,000 and the max was 115,000. That makes the range from minimum to maximum 72,000. To convert each value to a value in the range 0 to 1 we subtract the minimum from the value and divide by the range.

gals		
name	salary	normalized salary
Yun L	75,000	0.444
Allie C	55,000	0.167
Daniela C	45,000	0.028
Rita A	115,000	1.0

So the normalized value for Yun is

(75,000 - 43,000) / 72,000 = 0.444

Depending on the dataset this rough method of normalization may work well.

If you have taken a statistics course you will be familiar with more accurate methods for standardizing data. For example, we can use what is called The Standard Score which can be computed as follows

We can standardize a value using the Standard Score (aka z-score) which tells us how many deviations the value is from the mean!

$$\frac{\text{(each value) - (mean)}}{\text{(standard deviation)}} = \text{Standard Score}$$

Standard Deviation is

$$sd = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{card(x)}}$$

card(x) is the cardinality of x—that is, how many values there are.

By the way, if you are rusty with statistics and like manga be sure to check out the awesome book "The Manga Guide to Statistics" by Shin Takahashi.

Consider the data from the dating site example a few pages back.

name	salary
Yun L	75,000
Allie C	55,000
Daniela C	45,000
Rita A	115,000
Brian A	70,000
Abdullah K	105,000
David A	69,000
Michael W	43,000

The sum of all the salaries is 577,000. Since there are 8 people, the mean is 72,125.

Now let us compute the standard deviation:

$$sd = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{card(x)}}$$

so that would be

Yun's salary Allie's salary Daniela's salary etc.

$$\sqrt{\frac{(75{,}000 - 72{,}125)^2 + (55{,}000 - 72{,}125)^2 + (45{,}000 - 72{,}125)^2 + \ldots}{8}}$$

$$= \sqrt{\frac{8{,}265{,}625 + 293{,}265{,}625 + 735{,}765{,}625 + \ldots}{8}} = \sqrt{602{,}395{,}375}$$

$$= 24{,}543.01$$

Again, the standard score is

$$\frac{\text{(each value) - (mean)}}{\text{(standard deviation)}}$$

So the Standard Score for Yun's salary is

$$\frac{75000 - 72125}{24543.01} = \frac{2875}{24543.01} = 0.117$$

sharpen your pencil

Can you compute the Standard Scores for the following people?

name	salary	Standard Score
Yun L	75,000	0.117
Allie C	55,000	
Daniela C	45,000	
Rita A	115,000	

sharpen your pencil – solution

Can you compute the Standard Scores for the following people?

name	salary	Standard Score
Yun L	75,000	0.117
Allie C	55,000	-0.698
Daniela C	45,000	-1.105
Rita A	115,000	1.747

Allie:
(55,000 - 72,125) / 24,543.01
= -0.698

Daniela:
(45,000 - 72,125) / 24,543.01
= -1.105

Rita:
(115,000 - 72,125) / 24,543.01
= 1.747

The problem with using Standard Score

The problem with the standard score is that it is greatly influenced by outliers. For example, if all the 100 employees of LargeMart make $10/hr but the CEO makes six million a year the mean hourly wage is

(100 * $10 + 6,000,000 / (40 * 52)) / 101

= (1000 + 2885) / 101 = $38/hr.

Not a bad average wage at LargeMart. As you can see, the mean is greatly influenced by outliers.

Because of this problem with the mean, the standard score formula is often modified.

Modified Standard Score

To compute the Modified Standard Score you replace the mean in the above formula by the median (the middle value) and replace the standard deviation by what is called the absolute standard deviation:

$$asd = \frac{1}{card(x)} \sum_i |x_i - \mu|$$

where μ is the median.

Modified Standard Score:

$$\frac{\text{(each value) - (median)}}{\text{(absolute standard deviation)}}$$

To compute the median you arrange the values from lowest to highest and pick the middle value. If there are an even number of values the median is the average of the two middle values.

Okay, let's give this a try. In the table on the right I've arranged our salaries from lowest to highest. Since there are an equal number of values, the median is the average of the two middle values:

Name	Salary
Michael W	43,000
Daniela C	45,000
Allie C	55,000
David A	69,000
Brian A	70,000
Yun L	75,000
Abdullah K	105,000
Rita A	115,000

$$median = \frac{(69{,}000 + 70{,}000)}{2} = 69{,}500$$

The absolute standard deviation is

$$asd = \frac{1}{card(x)} \sum_i |x_i - \mu|$$

$$asd = \frac{1}{8}(|43{,}000 - 69{,}500| + |45{,}000 - 69{,}500| + |55{,}000 - 69{,}500)| + ...)$$

$$= \frac{1}{8}(26{,}500 + 24{,}500 + 14{,}500 + 500 + ...)$$

$$= \frac{1}{8}(153{,}000) = 19{,}125$$

Now let us compute the Modified Standard Score for Yun.

Modified Standard Score:

$$\frac{\text{(each value) - (median)}}{\text{(absolute standard deviation)}}$$

$$mss = \frac{(75{,}000 - 69{,}500)}{19{,}125} = \frac{5{,}500}{19{,}125} = 0.2876$$

sharpen your pencil

The following table shows the play count of various tracks I played. Can you standardize the values using the Modified Standard Score?

track	play count	modified standard score
Power/Marcus Miller	21	
I Breathe In, I Breathe Out/ Chris Cagle	15	
Blessed / Jill Scott	12	
Europa/Santana	3	
Santa Fe/ Beirut	7	

sharpen your pencil – solution

The following table shows the play count of various tracks I played. Can you standardize the values using the Modified Standard Score?

Step 1. Computing the median.

I put the values in order (3, 7, 12, 15, 21) and select the middle value, 12.
The median μ is 12.

Step 2. Computing the Absolute Standard Deviation.

$$asd = \frac{1}{5}(|3-12|+|7-12|+|12-12|+|15-12|+|21-12|)$$

$$= \frac{1}{5}(9+5+0+3+9) = \frac{1}{5}(26) = 5.2$$

Step 3. Computing the Modified Standard Scores.

Power / Marcus Miller: (21 - 12) / 5.2 = 9/5.2 = 1.7307692

I Breathe In, I Breathe Out / Chris Cagle: (15 - 12) / 5.2 = 3/5.2 = 0.5769231

Blessed / Jill Scott: (12 - 12) / 5.2 = 0

Europa / Santana: (3 - 12) / 5.2 = -9 / 5.2 = -1.7307692

Santa Fe / Beirut: (7 - 12) / 5.2 = - 5 / 5.2 = -0.961538

To normalize or not.

Normalization makes sense when the scale of the features—the scales of the different dimensions—significantly varies. In the music example earlier in the chapter there were a number of features that ranged from one to five and then beats-per-minute that could potentially range from 60 to 180. In the dating example, there was also a mismatch of scale between the features of age and salary.

Suppose I am dreaming of being rich and looking at homes in the Santa Fe, New Mexico area.

asking price	bedrooms	bathrooms	sq. ft.
$1,045,000	2	2.0	1,860
$1,895,000	3	4.0	2,907
$3,300,000	6	7.0	10,180
$6,800,000	5	6.0	8,653
$2,250,000	3	2.0	1,030

The table on the left shows a few recent homes on the market.

Here we see the problem again. Because the scale of one feature (in this case asking price) is so much larger than others it will dominate any distance calculation. Having two bedrooms or twenty will not have much of an effect on the total distance between two homes.

We should normalize when

1. **our data mining method calculates the distance between two entries based on the values of their features.**

2. **the scale of the different features is different (especially when it is drastically different—for ex., the scale of asking price compared to the scale of the number of bedrooms).**

Consider a person giving thumbs up and thumbs down ratings to news articles on a news site. Here a list representing a user's ratings consists of binary values (1 = thumbs up; 0 = thumbs down):