Chapter 4 Content Based Filtering & Classification

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”

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.

<table>
<thead>
<tr>
<th>El Ten Eleven</th>
<th>My Only Swerving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beats per Minute:</td>
<td>110</td>
</tr>
<tr>
<td>swinging 16ths:</td>
<td>Ø</td>
</tr>
<tr>
<td>well articulated piano solo:</td>
<td>2</td>
</tr>
<tr>
<td>block chords:</td>
<td>3</td>
</tr>
<tr>
<td>acoustic instrumentation:</td>
<td>5</td>
</tr>
</tbody>
</table>
The analyst provides values for over 400 genes. It’s 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:

<table>
<thead>
<tr>
<th></th>
<th>genre</th>
<th>Mood</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Country</td>
<td>Melancholy</td>
</tr>
<tr>
<td></td>
<td>Jazz</td>
<td>joyful</td>
</tr>
<tr>
<td></td>
<td>Rock</td>
<td>passion</td>
</tr>
<tr>
<td></td>
<td>Soul</td>
<td>angry</td>
</tr>
<tr>
<td></td>
<td>Rap</td>
<td>unknown</td>
</tr>
</tbody>
</table>

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

That “You’re Beautiful” is so sad and beautiful. I love it!
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?

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:

<table>
<thead>
<tr>
<th>Mood</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>melancholy</td>
<td>Country</td>
</tr>
<tr>
<td>joyful</td>
<td>Jazz</td>
</tr>
<tr>
<td>passion</td>
<td>Rock</td>
</tr>
<tr>
<td>angry</td>
<td>Soul</td>
</tr>
<tr>
<td>unknown</td>
<td>Rap</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Mood</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>melancholy</td>
<td>Country</td>
</tr>
<tr>
<td>angry</td>
<td>Jazz</td>
</tr>
<tr>
<td>passion</td>
<td>Soul</td>
</tr>
<tr>
<td>joyful</td>
<td>Rap</td>
</tr>
<tr>
<td>unknown</td>
<td>Rock</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Song</th>
<th>I breathe in, breathe out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country</td>
<td>1</td>
</tr>
<tr>
<td>Jazz</td>
<td>1</td>
</tr>
<tr>
<td>Soul</td>
<td>1</td>
</tr>
<tr>
<td>Rap</td>
<td>1</td>
</tr>
<tr>
<td>Rock</td>
<td>1</td>
</tr>
</tbody>
</table>

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 $\frac{1}{2}$ 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 $\frac{1}{2}$ integer increments (I admit this isn't a very rational nor complete selection):

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amount of piano</td>
<td>1 indicates lack of piano; 5 indicates piano throughout and featured prominently</td>
</tr>
<tr>
<td>Amount of vocals</td>
<td>1 indicates lack of vocals; 5 indicates prominent vocals throughout song.</td>
</tr>
<tr>
<td>Driving beat</td>
<td>Combination of constant tempo, and how the drums &amp; bass drive the beat.</td>
</tr>
<tr>
<td>Blues Influence</td>
<td></td>
</tr>
<tr>
<td>Presence of dirty electric guitar</td>
<td></td>
</tr>
<tr>
<td>Presence of backup vocals</td>
<td></td>
</tr>
<tr>
<td>Rap Influence</td>
<td></td>
</tr>
</tbody>
</table>

Now, using those features I rate ten tunes:
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:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Dog/ Fate</td>
<td>2.5</td>
<td>4</td>
<td>3.5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Phoenix/ Lisztomania</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Heartless Bastards / Out at Sea</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Todd Snider/ Don't Tempt Me</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>The Black Keys/ Magic Potion</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>3.5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Glee Cast/ Jessie's Girl</td>
<td>1</td>
<td>5</td>
<td>3.5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Black Eyed Peas/ Rock that Body</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>La Roux/ Bulletproof</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mike Posner/ Cooler than me</td>
<td>2.5</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Lady Gaga/ Alejandro</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Distance**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Dog/ Fate</td>
<td>2.5</td>
<td>4</td>
<td>3.5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Phoenix/ Lisztomania</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Distance</strong></td>
<td><strong>0.5</strong></td>
<td><strong>1</strong></td>
<td><strong>1.5</strong></td>
<td><strong>0</strong></td>
<td><strong>3</strong></td>
<td><strong>3</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

summing those distances gives us a Manhattan Distance of 9.
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?

<table>
<thead>
<tr>
<th>Song 1</th>
<th>Distance to Glee’s Jessie’s Girl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Dog/ Fate</td>
<td>??</td>
</tr>
<tr>
<td>Phoenix/ Lisztomania</td>
<td>4.822</td>
</tr>
<tr>
<td>Heartless Bastards / Out at Sea</td>
<td>4.153</td>
</tr>
<tr>
<td>Todd Snider/ Don’t Tempt Me</td>
<td>4.387</td>
</tr>
<tr>
<td>The Black Keys/ Magic Potion</td>
<td>4.528</td>
</tr>
<tr>
<td>Glee Cast/ Jessie’s Girl</td>
<td>0</td>
</tr>
<tr>
<td>Black Eyed Peas/ Rock that Body</td>
<td>5.408</td>
</tr>
<tr>
<td>La Roux/ Bulletproof</td>
<td>6.500</td>
</tr>
<tr>
<td>Mike Posner/ Cooler than me</td>
<td>5.701</td>
</tr>
<tr>
<td>Lady Gaga/ Alejandro</td>
<td>??</td>
</tr>
</tbody>
</table>
**sharpen your pencil - solution**

<table>
<thead>
<tr>
<th></th>
<th>distance to Glee’s Jessie’s Girl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Dog/ Fate</td>
<td>2.291</td>
</tr>
<tr>
<td>Lady Gaga/ Alejandro</td>
<td>4.387</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>piano</th>
<th>vocals</th>
<th>beat</th>
<th>blues</th>
<th>guitar</th>
<th>backup</th>
<th>rap</th>
<th>SUM</th>
<th>SQRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Glee</td>
<td>1</td>
<td>5</td>
<td>3.5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lady G</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x-y)</td>
<td>0</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x-y)^2</td>
<td>0</td>
<td>0</td>
<td>0.25</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>19.25</td>
<td>4.387</td>
</tr>
</tbody>
</table>
Doing it Python Style!

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

```python
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:

```python
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,
```

4-12
"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:

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

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](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:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Black Keys Magic Potion</strong></td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Heartless Bastards Out at Sea</strong></td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>3.5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>difference</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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:

```python
>>> 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")]
```

```python
>>> 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:

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

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.

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.

<table>
<thead>
<tr>
<th>gals</th>
<th>salary</th>
<th>normalized salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yun L</td>
<td>75,000</td>
<td>0.444</td>
</tr>
<tr>
<td>Allie C</td>
<td>55,000</td>
<td>0.167</td>
</tr>
<tr>
<td>Daniela C</td>
<td>45,000</td>
<td>0.028</td>
</tr>
<tr>
<td>Rita A</td>
<td>115,000</td>
<td>1.0</td>
</tr>
</tbody>
</table>

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!

\[
\text{(each value)} - \text{(mean)} = \text{Standard Score} \\
(\text{standard deviation})
\]

Standard Deviation is

\[
sd = \sqrt{\frac{\sum (x_i - \bar{x})^2}{\text{card}(x)}}
\]

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.

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:

\[ sd = \sqrt{\frac{\sum (x_i - \bar{x})^2}{\text{card}(x)}} \]

so that would be

Yun's salary \[ 75,000 - 72,125 \]
Allie's salary \[ 55,000 - 72,125 \]
Daniela's salary \[ 45,000 - 72,125 \]
... etc.

\[ \sqrt{(75,000 - 72,125)^2 + (55,000 - 72,125)^2 + (45,000 - 72,125)^2 + ...} \]
\[ = \sqrt{8,265,625 + 293,265,625 + 735,765,625 + ...} \]
\[ = \sqrt{8} \]
\[ = \sqrt{602,395,375} \]
\[ = 24,543.01 \]
Again, the standard score is

\[
\frac{(each \ value) - (mean)}{(standard \ deviation)}
\]

So the Standard Score for Yun’s salary is

\[
\frac{75000 - 72125}{24543.01} = \frac{2875}{24543.01} = 0.117
\]

\[\star\] sharpen your pencil

Can you compute the Standard Scores for the following people?

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
<th>Standard Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yun L</td>
<td>75,000</td>
<td>0.117</td>
</tr>
<tr>
<td>Allie C</td>
<td>55,000</td>
<td></td>
</tr>
<tr>
<td>Daniela C</td>
<td>45,000</td>
<td></td>
</tr>
<tr>
<td>Rita A</td>
<td>115,000</td>
<td></td>
</tr>
</tbody>
</table>
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

\[
(100 \times 10 + 6,000,000 / (40 \times 52)) / 101
\]

\[
= (1000 + 2885) / 101 = \$38/hr.
\]
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:

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

where $\mu$ is the median.

**Modified Standard Score:**

$$(\text{each value}) - (\text{median})$$

___________________________

$$(\text{absolute standard deviation})$$

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:

\[
\text{median} = \frac{(69,000 + 70,000)}{2} = 69,500
\]

The absolute standard deviation is

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

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

\[
= \frac{1}{8} (26,500 + 24,500 + 14,500 + 500 + \ldots)
\]

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

Now let us compute the Modified Standard Score for Yun.

\[
\text{Modified Standard Score:}
\]

\[
mss = \frac{(75,000 - 69,500)}{19,125} = \frac{5,500}{19,125} = 0.2876
\]
The following table shows the play count of various tracks I played. Can you standardize the values using the Modified Standard Score?

<table>
<thead>
<tr>
<th>track</th>
<th>play count</th>
<th>modified standard score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power/Marcus Miller</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>I Breathe In, I Breathe Out/Chris Cagle</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Blessed / Jill Scott</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Europa/Santana</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Santa Fe/ Beirut</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
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 $\mu$ 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.

<table>
<thead>
<tr>
<th>asking price</th>
<th>bedrooms</th>
<th>bathrooms</th>
<th>sq. ft.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1,045,000</td>
<td>2</td>
<td>2.0</td>
<td>1,860</td>
</tr>
<tr>
<td>$1,895,000</td>
<td>3</td>
<td>4.0</td>
<td>2,907</td>
</tr>
<tr>
<td>$3,300,000</td>
<td>6</td>
<td>7.0</td>
<td>10,180</td>
</tr>
<tr>
<td>$6,800,000</td>
<td>5</td>
<td>6.0</td>
<td>8,653</td>
</tr>
<tr>
<td>$2,250,000</td>
<td>3</td>
<td>2.0</td>
<td>1,030</td>
</tr>
</tbody>
</table>

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):
Bill = \{0, 0, 0, 1, 1, 1, 1, 0, 1, 0 \ldots \}

Obviously there is no need to normalize this data. What about the Pandora case: all variables lie on a scale from 1 to 5 inclusive. Should we normalize or not? It probably wouldn't hurt the accuracy of the algorithm if we normalized, but keep in mind that there is a computational cost involved with normalizing. In this case, we might empirically compare results between using the regular and normalized data and select the best performing approach. Later in this chapter we will see a case where normalization reduces accuracy.

**Back to Pandora**

In the Pandora inspired example, we had each song represented by a number of attributes. If a user creates a radio station for Green Day we decide what to play based on a nearest neighbor approach. Pandora allows a user to give a particular tune a thumbs up or thumbs down rating. How do we use the information that a user gives a thumbs up for a particular song?

Suppose I use 2 attributes for songs: the amount of dirty guitar and the presence of a driving beat both rated on a 1-5 scale. A user has given the thumbs up to 5 songs indicating he liked the song (and indicated on the following chart with a 'L'); and a thumbs down to 5 songs indicating he disliked the song (indicated by a 'D').

Do you think the user will like or dislike the song indicated by the ‘?’ in this chart?
I am guessing you said he would like the song. We base this on the fact that the ‘?’ is closer to the Ls in the chart than the Ds. We will spend the rest of this chapter and the next describing computational approaches to this idea. The most obvious approach is to find the nearest neighbor of the “?” and predict that it will share the class of the nearest neighbor. The question mark’s nearest neighbor is an L so we would predict that the ‘? tune’ is something the user would like.

**The Python nearest neighbor classifier code**

Let's use the example dataset I used earlier—ten tunes rated on 7 attributes (amount of piano, vocals, driving beat, blues influence, dirty electric guitar, backup vocals, rap influence).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Dog/Fate</td>
<td>2.5</td>
<td>4</td>
<td>3.5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Phoenix/Lisztomania</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Heartless Bastards/Out at Sea</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Todd Snider/Don’t Tempt Me</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>The Black Keys/Magic Potion</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>3.5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Glee Cast/Jessie’s Girl</td>
<td>1</td>
<td>5</td>
<td>3.5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Black Eyed Peas/Rock that Body</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>La Roux/Bulletproof</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mike Posner/Cooler than me</td>
<td>2.5</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Lady Gaga/Alejandro</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Earlier in this chapter we developed a Python representation of this data:

```python
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, "blues": 2,
        "guitar": 4,
        "backup vocals": 4, "rap": 1
    },
    "La Roux/Bulletproof": {
        "piano": 5, "vocals": 4,
        "beat": 1, "blues": 2,
        "guitar": 1,
        "backup vocals": 1, "rap": 1
    },
    "Mike Posner": {
        "piano": 2.5, "vocals": 4,
        "beat": 1, "blues": 1,
        "guitar": 1, "backup vocals": 1, "rap": 1
    }
}
```

Here the strings `piano`, `vocals`, `beat`, `blues`, `guitar`, `backup vocals`, and `rap` occur multiple times; if I have a 100,000 tunes those strings are repeated 100,000 times. I'm going to remove those strings from the representation of our data and simply use vectors:

```python
# the item vector represents the attributes: piano, vocals, beat, blues, guitar, backup vocals, rap
#
items = {
    "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],
    "La Roux/Bulletproof": [5, 5, 4, 2, 1, 1, 1],
    "Mike Posner": [2.5, 4, 4, 1, 1, 1, 1],
    "Black Eyed Peas/Rock That Body": [2, 5, 5, 1, 2, 2, 4],
    "Lady Gaga/Alejandro": [1, 5, 3, 2, 1, 2, 1]
}
```
In linear algebra, a vector is a quantity that has magnitude and direction. Various well defined operators can be performed on vectors including adding and subtracting vectors and scalar multiplication.

In data mining, a vector is simply a list of numbers that represent the attributes of an object. The example on the previous page represented attributes of a song as a list of numbers. Another example, would be representing a text document as a vector—each position of the vector would represent a particular word and the number at that position would represent how many times that word occurred in the text.

Plus, using the word "vector" instead of "list of attributes" is cool!

Once we define attributes this way, we can perform vector operations (from linear algebra) on them.
In addition to representing the attributes of a song as a vector, I need to represent the thumbs up/thumbs down ratings that users give to songs. Because each user doesn't rate all songs (sparse data) I will go with the dictionary of dictionaries approach:

```
```

My way of representing ‘thumbs up’ as L for like and ‘thumbs down’ as D is arbitrary. You could use 0 and 1, like and dislike.

In order to use the new vector format for songs I need to revise the Manhattan Distance and the computeNearestNeighbor functions.

```python
def manhattan(vector1, vector2):
    """Computes the Manhattan distance."""
    distance = 0
    total = 0
    n = len(vector1)
    for i in range(n):
        distance += abs(vector1[i] - vector2[i])
    return distance
```
def computeNearestNeighbor(itemName, itemVector, items):
    """creates a sorted list of items based on their distance to item""
    distances = []
    for otherItem in items:
        if otherItem != itemName:
            distance = manhattan(itemVector, items[otherItem])
            distances.append((distance, otherItem))
    # sort based on distance -- closest first
    distances.sort()
    return distances

Finally, I need to create a classify function. I want to predict how a particular user would rate an item represented by itemName and itemVector. For example:

"Chris Cagle/ I Breathe In. I Breathe Out"  [1, 5, 2.5, 1, 1, 5, 1]

(NOTE: To better format the Python example below, I will use the string Cagle to represent that singer and song pair.)

The first thing the function needs to do is find the nearest neighbor of this Chris Cagle tune. Then it needs to see how the user rated that nearest neighbor and predict that the user will rate Chris Cagle the same. Here's my rudimentary classify function:

def classify(user, itemName, itemVector):
    """Classify the itemName based on user ratings
    Should really have items and users as parameters""
    # first find nearest neighbor
    nearest = computeNearestNeighbor(itemName, itemVector, items)[0][1]
    rating = users[user][nearest]
    return rating

Ok. Let's give this a try. I wonder if Angelica will like Chris Cagle's I Breathe In, I Breathe Out?

classify('Angelica', 'Cagle', [1, 5, 2.5, 1, 1, 5, 1])
"L"

We are predicting she will like it! Why are we predicting that?
```python
computeNearestNeighbor('Angelica', 'Cagle', [1, 5, 2.5, 1, 1, 5, 1])
```

```
```

We are predicting that Angelica will like Chris Cagle's *I Breathe In, I Breathe Out* because that tune's nearest neighbor is Lady Gaga’s *Alejandro* and Angelica liked that tune.

What we have done here is build a classifier—in this case, our task was to classify tunes as belonging to one of two groups—the like group and the dislike group.

**Attention, Attention.**

**We just built a classifier!!**
A classifier is a program that uses an object’s attributes to determine what group or class it belongs to!

A classifier uses a set of objects that are already labeled with the class they belong to. It uses that set to classify new, unlabeled objects. So in our example, we knew about songs that Angelica liked (labeled ‘liked’) and songs she did not like. We wanted to predict whether Angelica would like a Chris Cagle tune.

First we found a song Angelica rated that was most similar to the Chris Cagle tune. It was Lady Gaga’s Alejandro.

Next, we checked whether Angelica liked or disliked the Alejandro—she liked it. So we predict that Angelica will also like the Chris Cagle tune, I Breathe In, I Breathe Out.

Classifiers can be used in a wide range of applications. The following page lists just a few.
Twitter Sentiment Classification
A number of people are working on classifying the sentiment (a positive or negative opinion) in tweets. This can be used in a variety of ways. For example, if Axe releases a new underarm deoderant, they can check whether people like it or not. The attributes are the words in the tweet.

Automatic identification of people in photos.
There are apps now that can identify and tag your friends in photographs. (And the same techniques apply to identifying people walking down the street using public video cams.) Techniques vary but some of them use attributes like the relative position and size of a person’s eyes, nose, jaw, etc.

Classification For Targeted Political Ads
This is called microtargeting. People are classified into such groups as “Barn Raisers”, “Inner Compass”, and “Hearth Keepers.” Hearth Keepers, for example, focus on their family and keep to themselves.

Targeted Marketing
Similar to political microtargeting. Instead of a broad advertising campaign to sell my expensive Vegas time share luxury condos, can I identify likely buyers and market just to them? Even better if I can identify subgroups of likely buyers and I can really tailor my ads to specific groups.

Health and the Quantified Self
It’s the start of the quantified self explosion. We can now buy simple devices like the Fitbit, and the Nike Fuelband. Intel and other companies are working on intelligent homes that have floors that can weigh us, keep track of our movements and alert someone if we deviate from normal. Experts are predicting that in a few years we will be wearing tiny compu-patches that can monitor dozens of factors in real time and make instant classifications.

The list is endless
- classifying people as terrorist or nonterrorist
- automatic classification of email (hey, this email looks pretty important; this is regular email; this looks like spam)
- predicting medical clinical outcomes
- identifying financial fraud (for ex., credit card fraud)
What sport?

To give you a preview of what we will be working on in the next few chapters let us work with an easier example than those given on the previous page—classifying what sport various world-class women athletes play based solely on their height and weight. In the following table I have a small sample dataset drawn from a variety of web sources.

<table>
<thead>
<tr>
<th>Name</th>
<th>Sport</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asuka Teramoto</td>
<td>Gymnastics</td>
<td>16</td>
<td>54</td>
<td>66</td>
</tr>
<tr>
<td>Brittaneey Raven</td>
<td>Basketball</td>
<td>22</td>
<td>72</td>
<td>162</td>
</tr>
<tr>
<td>Chen Nan</td>
<td>Basketball</td>
<td>30</td>
<td>78</td>
<td>204</td>
</tr>
<tr>
<td>Gabby Douglas</td>
<td>Gymnastics</td>
<td>16</td>
<td>49</td>
<td>90</td>
</tr>
<tr>
<td>Helalia Johannes</td>
<td>Track</td>
<td>32</td>
<td>65</td>
<td>99</td>
</tr>
<tr>
<td>Irina Miketenko</td>
<td>Track</td>
<td>40</td>
<td>63</td>
<td>106</td>
</tr>
<tr>
<td>Jennifer Lacy</td>
<td>Basketball</td>
<td>27</td>
<td>75</td>
<td>175</td>
</tr>
<tr>
<td>Kara Goucher</td>
<td>Track</td>
<td>34</td>
<td>67</td>
<td>123</td>
</tr>
<tr>
<td>Linlin Deng</td>
<td>Gymnastics</td>
<td>16</td>
<td>54</td>
<td>68</td>
</tr>
<tr>
<td>Nakia Sanford</td>
<td>Basketball</td>
<td>34</td>
<td>76</td>
<td>200</td>
</tr>
<tr>
<td>Nikki Blue</td>
<td>Basketball</td>
<td>26</td>
<td>68</td>
<td>163</td>
</tr>
<tr>
<td>Qiushuang Huang</td>
<td>Gymnastics</td>
<td>20</td>
<td>61</td>
<td>95</td>
</tr>
<tr>
<td>Rebecca Tunney</td>
<td>Gymnastics</td>
<td>16</td>
<td>58</td>
<td>77</td>
</tr>
<tr>
<td>Rene Kalmer</td>
<td>Track</td>
<td>32</td>
<td>70</td>
<td>108</td>
</tr>
<tr>
<td>Shanna Crossley</td>
<td>Basketball</td>
<td>26</td>
<td>70</td>
<td>155</td>
</tr>
<tr>
<td>Shavonte Zellous</td>
<td>Basketball</td>
<td>24</td>
<td>70</td>
<td>155</td>
</tr>
<tr>
<td>Tatyana Petrova</td>
<td>Track</td>
<td>29</td>
<td>63</td>
<td>108</td>
</tr>
<tr>
<td>Tiki Gelana</td>
<td>Track</td>
<td>25</td>
<td>65</td>
<td>106</td>
</tr>
<tr>
<td>Valeria Straneo</td>
<td>Track</td>
<td>36</td>
<td>66</td>
<td>97</td>
</tr>
<tr>
<td>Viktoria Komova</td>
<td>Gymnastics</td>
<td>17</td>
<td>61</td>
<td>76</td>
</tr>
</tbody>
</table>
The gymnastic data lists some of the top participants in the 2012 and 2008 Olympics. The basketball players play for teams in the WNBA. The women track stars were finishers in the 2012 Olympic marathon. Granted this is a trivial example but it will allow us to apply some of the techniques we have learned.

As you can see, I’ve included age in the table. Just scanning the data you can see that age alone is a moderately good predictor. Try to guess the sports of these athletes.

Candace Parker; Age 26

McKayla Maroney; Age 16

Lisa Jane Weightman; Age 34

Olivera Jevtić; Age 35
The answers

Candace Parker plays basketball for the WNBA’s Los Angeles Sparks and Russia’s UMMC Ekaterinburg. McKayla Maroney was a member of the U.S. Women’s Gymnastic Team and won a Gold and a Silver. Olivera Jevtić is a Serbian long-distance runner who competed in the 2008 and 2012 Olympics. Lisa Jane Weightman is an Australian long-distance runner who also competed in the 2008 and 2012 Olympics.

You just performed classification—you predicted the class of objects based on their attributes. (In this case, predicting the sport of athletes based on a single attribute, age.)

brain calisthenics

Suppose I want to guess what sport a person plays based on their height and weight. My database is small—only two people. Nakia Sanford, the center for the Women’s National Basketball Association team Phoenix Mercury, is 6’4” and weighs 200 pounds. Sarah Beale, a forward on England’s National Rugby Team, is 5’10” and weighs 190. Based on that database, I want to classify Catherine Spencer as either a basketball player or rugby player. She is 5’10” and weighs 200 pounds. What sport do you think she plays?
If you said rugby, you would be correct. Catherine Spencer is a Forward on England’s national team. However, if we based our guess on a distance formula like Manhattan Distance we would be wrong. The Manhattan Distance between Catherine and Basketball player Nakia is 6 (they weigh the same and have a six inch difference in height). The distance between Catherine and Rugby player Sarah is 10 (their height is the same and they differ in weight by 10 pounds). So we would pick the closest person, Nakia, and predict Catherine plays the same sport.

Is there anything we learned that could help us make more accurate classifications?

Hmmm. This rings a bell. I think there was something related to this earlier in the chapter...
Test Data.

Let us remove age from the picture. Here is a group of individuals I would like to classify:

<table>
<thead>
<tr>
<th>Name</th>
<th>Sport</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crystal Langhorne</td>
<td></td>
<td>74</td>
<td>190</td>
</tr>
<tr>
<td>Li Shanshan</td>
<td></td>
<td>64</td>
<td>101</td>
</tr>
<tr>
<td>Kerri Strug</td>
<td></td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>Jaycie Phelps</td>
<td>`</td>
<td>60</td>
<td>97</td>
</tr>
<tr>
<td>Kelly Miller</td>
<td></td>
<td>70</td>
<td>140</td>
</tr>
<tr>
<td>Zhu Xiaolin</td>
<td></td>
<td>67</td>
<td>123</td>
</tr>
<tr>
<td>Lindsay Whalen</td>
<td></td>
<td>69</td>
<td>169</td>
</tr>
<tr>
<td>Koko Tsurumi</td>
<td></td>
<td>55</td>
<td>75</td>
</tr>
<tr>
<td>Paula Radcliffe</td>
<td></td>
<td>68</td>
<td>120</td>
</tr>
<tr>
<td>Erin Thorn</td>
<td></td>
<td>69</td>
<td>144</td>
</tr>
</tbody>
</table>

Let’s build a classifier!
Instead of hard-coding the data in the Python code, I decided to put the data for this example into two files: athletesTrainingSet.txt and athletesTestSet.txt.

I am going to use the data in the athletesTrainingSet.txt file to build the classifier. The data in the athletesTestSet.txt file will be used to evaluate this classifier. In other words, each entry in the test set will be classified by using all the entries in the training set.

The format of these files looks like this:

Asuka Teramoto  Gymnastics  54  66  
Brittainey Raven  Basketball  72  162  
Chen Nan  Basketball  78  204  
Gabby Douglas  Gymnastics  49  90  

Each line of the text represents an object described as a tab-separated list of values. I want my classifier to use a person’s height and weight to predict what sport that person plays. So the last two columns are the numerical attributes I will use in the classifier and the second column represents the class that object is in. The athlete’s name is not used by the classifier. I don’t try to predict what sport a person plays based on their name and I am not trying to predict the name from some attributes.
However, keeping the name might be useful as a means of explaining the classifier’s decision to users: “We think Amelia Pond is a gymnast because she is closest in height and weight to Gabby Douglas who is a gymnast.”

As I said, I am going to write my Python code to not be so hard coded to a particular example (for example, to only work for the athlete example). To help meet this goal I am going to add an initial header line to the athlete training set file that will indicate the function of each column. Here are the first few lines of that file:

```
comment   class   num   num
Asuka Teramoto Gymnastics  54   66
Brittainey Raven Basketball  72   162
```

Any column labeled `comment` will be ignored by the classifier; a column labeled `class` represents the class of the object, and columns labeled `num` indicate numerical attributes of that object.

---

**brain calisthenics -**

How do you think we should represent this data in Python? Here are some possibilities (or come up with your own representation).

- A dictionary of the form:
  ```
  {'Asuka Termoto': ('Gymnastics', [54, 66]),
   'Brittainey Raven': ('Basketball', [72, 162]), ...
  ```

- A list of lists of the form:
  ```
  [['Asuka Termoto', 'Gymnastics', 54, 66],
   ['Brittainey Raven', 'Basketball', 72, 162], ...
  ```

- A list of tuples of the form:
  ```
  [('Gymnastics', [54, 66], ['Asuka Termoto']),
   ('Basketball', [72, 162], ['Brittainey Raven']), ...
  ```
a dictionary of the form:

```python
{'Asuka Termoto': ('Gymnastics', [54, 66]),
 'Brittainey Raven': ('Basketball', [72, 162]), ...}
```

This is not a very good representation of our data. The key for the dictionary is the athlete's name, which we do not even use in the calculations.

a list of lists of the form:

```python
[['Asuka Termoto', 'Gymnastics', 54, 66],
 ['Brittainey Raven', 'Basketball', 72, 162], ...]
```

This is not a bad representation. It mirrors the input file and since the nearest neighbor algorithm requires us to iterate through the list of objects, a list makes sense.

a list of tuples of the form:

```python
(('Gymnastics', [54, 66], ['Asuka Termoto']),
 ('Basketball', [72, 162], ['Brittainey Raven']), ...)
```

I like this representation better than the above since it separates the attributes into their own list and makes the division between class, attributes, and comments precise. I made the comment (the name in this case) a list since there could be multiple columns that are comments.
My python code that reads in a file and converts it to the format

[(('Gymnastics', [54, 66], ['Asuka Termoto']),
('Basketball', [72, 162], ['Brittainey Raven']),...)

looks like this:

class Classifier:

    def __init__(self, filename):
        self.medianAndDeviation = []
        
        # reading the data in from the file
        f = open(filename)
        lines = f.readlines()
        f.close()
        self.format = lines[0].strip().split('	')
        self.data = []
        for line in lines[1:]:
            fields = line.strip().split('	')
            ignore = []
            vector = []
            for i in range(len(fields)):
                if self.format[i] == 'num':
                    vector.append(int(fields[i]))
                elif self.format[i] == 'comment':
                    ignore.append(fields[i])
                elif self.format[i] == 'class':
                    classification = fields[i]
            self.data.append((classification, vector, ignore))
Before we can standardize the data using the Modified Standard Score we need methods that will compute the median and absolute standard deviation of numbers in a list:

```python
>>> heights = [54, 72, 78, 49, 65, 63, 75, 67, 54]
>>> median = classifier.getMedian(heights)
>>> median
65
>>> asd = classifier.getAbsoluteStandardDeviation(heights, median)
>>> asd
8.0
```

Can you write these methods?

Download the template testMedianAndASD.py to write and test these methods at guidetodatamining.com
Assertion Errors and the Assert statement.

It is important that each component of a solution to a problem be turned into a piece of code that implements it and a piece of code that tests it. In fact, it is good practice to write the test code before you write the implementation. The code template I have provided contains a test function called `unitTest`. A simplified version of that function, showing only one test, is shown here:

```python
def unitTest():
    list1 = [54, 72, 78, 49, 65, 63, 75, 67, 54]
    classifier = Classifier('athletesTrainingSet.txt')
    m1 = classifier.getMedian(list1)
    assert(round(m1, 3) == 65)
    print("getMedian and getAbsoluteStandardDeviation work correctly")
```

The `getMedian` function you are to complete initially looks like this:

```python
def getMedian(self, alist):
    """return median of alist""
    """TO BE DONE""
    return 0
```

So initially, `getMedian` returns 0 as the median for any list. You are to complete `getMedian` so it returns the correct value. In the `unitTest` procedure, I call `getMedian` with the list `[54, 72, 78, 49, 65, 63, 75, 67, 54]`

The `assert` statement in `unitTest` says the value returned by `getMedian` should equal 65. If it does, execution continues to the next line and

`getMedian and getAbsoluteStandardDeviation work correctly`

is printed. If they are not equal the program terminates with an error:
File "testMedianAndASD.py", line 78, in unitTest

    assert(round(m1, 3) == 65)

AssertionError

If you download the code from the book’s website and run it without making any changes, you will get this error. Once you have correctly implemented getMedian and getAbsoluteStandardDeviation this error will disappear.

This use of assert as a means of testing software components is a common technique among software developers.

"it is important that each part of the specification be turned into a piece of code that implements it and a test that tests it. If you don’t have tests like these then you don’t know when you are done, you don’t know if you got it right, and you don’t know that any future changes might be breaking something." - Peter Norvig
Solution

Here is one way of writing these algorithms:

```python
def getMedian(self, alist):
    # return median of alist
    if alist == []:
        return []
    blist = sorted(alist)
    length = len(alist)
    if length % 2 == 1:
        # length of list is odd so return middle element
        return blist[int(((length + 1) / 2) - 1)]
    else:
        # length of list is even so compute midpoint
        v1 = blist[int(length / 2)]
        v2 = blist[(int(length / 2) - 1)]
        return (v1 + v2) / 2.0

def getAbsoluteStandardDeviation(self, alist, median):
    # given alist and median return absolute standard deviation
    sum = 0
    for item in alist:
        sum += abs(item - median)
    return sum / len(alist)
```

As you can see my getMedian method first sorts the list before finding the median. Because I am not working with huge data sets I think this is a fine solution. If I wanted to optimize my code, I might replace this with a selection algorithm.

Right now, the data is read from the file athletesTrainingSet.txt and stored in the list data in the classifier with the following format:

```plaintext
[('Gymnastics', [54, 66], ['Asuka Teramoto']),
 ('Basketball', [72, 162], ['Brittainey Raven']),
 ('Basketball', [78, 204], ['Chen Nan']),
 ('Gymnastics', [49, 90], ['Gabby Douglas']), ...
```
Now I would like to normalize the vector so the list data in the classifier contains normalized values. For example,

```
[('Gymnastics', [-1.93277, -1.21842], ['Asuka Teramoto']),
 ('Basketball', [1.09243, 1.63447], ['Brittainey Raven']),
 ('Basketball', [2.10084, 2.88261], ['Chen Nan']),
 ('Gymnastics', [-2.77311, -0.50520], ['Gabby Douglas']),
 ('Track', [-0.08403, -0.23774], ['Helalia Johannes']),
 ('Track', [-0.42017, -0.02972], ['Irina Miketenko'])]
```

To do this I am going to add the following lines to my init method:

```python
# get length of instance vector
self.vlen = len(self.data[0][1])
# now normalize the data
for i in range(self.vlen):
    self.normalizeColumn(i)
```

In the for loop we want to normalize the data, column by column. So the first time through the loop we will normalize the height column, and the next time through, the weight column.

Can you write the normalizeColumn method?

Download the template normalizeColumnTemplate.py to write and test this method at guidetodatamining.com
Solution

Here is an implementation of the normalizeColumn method:

```python
def normalizeColumn(self, columnNumber):
    """given a column number, normalize that column in self.data""
    # first extract values to list
    col = [v[1][columnNumber] for v in self.data]
    median = self.getMedian(col)
    asd = self.getAbsoluteStandardDeviation(col, median)
    # print("Median: %f   ASD = %f" % (median, asd))
    self.medianAndDeviation.append((median, asd))
    for v in self.data:
        v[1][columnNumber] = (v[1][columnNumber] - median) / asd
```

You can see I also store the median and absolute standard deviation of each column in the list medianAndDeviation. I use this information when I want to use the classifier to predict the class of a new instance. For example, suppose I want to predict what sport is played by Kelly Miller, who is 5 feet 10 inches and weighs 170. The first step is to convert her height and weight to Modified Standard Scores. That is, her original attribute vector is [70, 140].

After processing the training data, the value of meanAndDeviation is

\[(65.5, 5.95), (107.0, 33.65)\]

meaning the data in the first column of the vector has a median of 65.5 and an absolute standard deviation of 5.95; the second column has a median of 107 and a deviation of 33.65.

I use this info to convert the original vector [70,140] to one containing Modified Standard Scores. This computation for the first attribute is

\[mss = \frac{x_i - \bar{x}}{asd} = \frac{70 - 65.5}{5.95} = \frac{4.5}{5.95} = 0.7563\]
\[ mss = \frac{x_i - \bar{x}}{asd} = \frac{140 - 107}{33.65} = \frac{33}{33.65} = 0.98068 \]

The Python method that does this is:

```python
def normalizeVector(self, v):
    """We have stored the median and asd for each column.
    We now use them to normalize vector v""
    vector = list(v)
    for i in range(len(vector)):
        (median, asd) = self.medianAndDeviation[i]
        vector[i] = (vector[i] - median) / asd
    return vector
```

The final bit of code to write is the part that predicts the class of a new instance—in our current example, the sport a person plays. To determine the sport played by Kelly Miller, who is 5 feet 10 inches (70 inches) and weighs 170 we would call

```python
classifier.classify([70, 170])
```

In my code, classify is just a wrapper method for nearestNeighbor:

```python
def classify(self, itemVector):
    """Return class we think item Vector is in""
    return (self.nearestNeighbor(self.normalizeVector(itemVector))[1])[0]
```

Can you write the nearestNeighbor method? (For my solution, I wrote an additional method, manhattanDistance.)

Yet again, download the template classifyTemplate.py to write and test this method at guidetodatamining.com.
Solution

The implementation of the nearestNeighbor methods turns out to be very short.

```python
def manhattan(self, vector1, vector2):
    """Computes the Manhattan distance."""
    return sum(map(lambda v1, v2: abs(v1 - v2), vector1, vector2))

def nearestNeighbor(self, itemVector):
    """return nearest neighbor to itemVector""
    return min(((self.manhattan(itemVector, item[1]), item)
                   for item in self.data))
```

That’s it!!!

We have written a nearest neighbor classifier in roughly 200 lines of Python.
In the complete code which you can download from our website, I have included a function, `test`, which takes as arguments a training set file and a test set file and prints out how well the classifier performed. Here is how well the classifier did on our athlete data:

```python
>>> test("athletesTrainingSet.txt", "athletesTestSet.txt")
-   Track  Aly Raisman   Gymnastics  62  115
+   Basketball  Crystal Langhorne  Basketball  74  190
+   Basketball  Diana Taurasi   Basketball  72  163
<snip>
-   Track  Hannah Whelan   Gymnastics  63  117
+   Gymnastics  Jaycie Phelps   Gymnastics  60  97
```

80.00% correct

As you can see, the classifier was 80% accurate. It performed perfectly on predicting basketball players but made four errors between track and gymnastics.

**Irises Data Set**

I also tested our simple classifier on the Iris Data Set, arguably the most famous data set used in data mining. It was used by Sir Ronald Fisher back in the 1930s. The Iris Data Set consists of 50 samples for each of three species of Irises (Iris Setosa, Iris Virginica, and Iris Versicolor). The data set includes measurements for two parts of the Iris’s flower: the sepal (the green covering of the flower bud) and the petals.
All the data sets described in the book are available on the book’s website: guidetodatamining.com. This allows you to download the data and experiment with the algorithm. Does normalizing the data improve or worsen the accuracy? Does having more data in the training set improve results? What effect does switching to Euclidean Distance have?

REMEMBER: Any learning that takes place happens in your brain, not mine. The more you interact with the material in the book, the more you will learn.

The Iris data set looks like this (species is what the classifier is trying to predict):

<table>
<thead>
<tr>
<th>Sepal length</th>
<th>Sepal width</th>
<th>Petal Length</th>
<th>Petal Width</th>
<th>Species</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>l.setosa</td>
</tr>
<tr>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>l.setosa</td>
</tr>
</tbody>
</table>
There were 120 instances in the training set and 30 in the test set (none of the test set instances were in the training set).

How well did our classifier do on the Iris Data Set?

```python
>>> test('irisTrainingSet.data', 'irisTestSet.data')

93.33% correct
```

Again, a fairly impressive result considering how simple our classifier is. Interestingly, without normalizing the data the classifier is 100% accurate. We will explore this normalization problem in more detail in a later chapter.

**miles per gallon.**

Finally, I tested our classifier on a modified version of another widely used data set, the Auto Miles Per Gallon data set from Carnegie Mellon University. It was initially used in the 1983 American Statistical Association Exposition. The format of the data looks like this:

<table>
<thead>
<tr>
<th>mpg</th>
<th>cylinders</th>
<th>c.i.</th>
<th>HP</th>
<th>weight</th>
<th>secs. 0-60</th>
<th>make/model</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>4</td>
<td>68</td>
<td>49</td>
<td>1867</td>
<td>19.5</td>
<td>fiat 128</td>
</tr>
<tr>
<td>45</td>
<td>4</td>
<td>90</td>
<td>48</td>
<td>2085</td>
<td>21.7</td>
<td>vw rabbit (diesel)</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>307</td>
<td>130</td>
<td>3504</td>
<td>12</td>
<td>chevrolet chevelle malibu</td>
</tr>
</tbody>
</table>

In the modified version of the data, we are trying to predict mpg, which is a discrete category (with values 10, 15, 20, 25, 30, 35, 40, and 45) using the attributes cylinders, displacement, horsepower, weight, and acceleration.
There are 342 instances of cars in the training set and 50 in the test set. If we just predicted the miles per gallon randomly, our accuracy would be 12.5%.

```python
>>> test('mpgTrainingSet.txt', 'mpgTestSet.txt')
56.00% correct
```

Without normalization the accuracy is 32%.

How can we improve the accuracy of our predictions?
Will improving the classification algorithm help?
How about increasing the size of our training set?
How about having more attributes.
Tune in to the next chapter to find out!
Heads Up on Normalization

In this chapter we talked the importance of normalizing data. This is critical when attributes have drastically different scales (for example, income and age). In order to get accurate distance measurements, we should rescale the attributes so they all have the same scale.

While most data miners use the term ‘normalization’ to refer to this rescaling, others make a distinction between ‘normalization’ and ‘standardization.’ For them, normalization means scaling values so they lie on a scale from 0 to 1. Standardization, on the other hand, refers to scaling an attribute so the average (mean or median) is 0, and other values are deviations from this average (standard deviation or absolute standard deviation). So for these data miners, Standard Score and Modified Standard Score are examples of standardization.

Recall that one way to normalize an attribute on a scale between 0 and 1 is to find the minimum (min) and maximum (max) values of that attribute. The normalized value of a value is then

\[
\frac{\text{value} - \text{min}}{\text{max} - \text{min}}
\]

Let’s compare the accuracy of a classifier that uses this formula over one that uses the Modified Standard
Can you modify our classifier code so that it normalizes the attributes using the formula on our previous page?

You can test its accuracy with our three data sets:

<table>
<thead>
<tr>
<th>data set</th>
<th>classifier built</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>using no normalization</td>
</tr>
<tr>
<td>Athletes</td>
<td>80.00%</td>
</tr>
<tr>
<td>Iris</td>
<td>100.00%</td>
</tr>
<tr>
<td>MPG</td>
<td>32.00%</td>
</tr>
</tbody>
</table>
Here are my results:

<table>
<thead>
<tr>
<th>data set</th>
<th>classifier built</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>using no normalization</td>
</tr>
<tr>
<td>Athletes</td>
<td>80.00%</td>
</tr>
<tr>
<td>Iris</td>
<td>100.00%</td>
</tr>
<tr>
<td>MPG</td>
<td>32.00%</td>
</tr>
</tbody>
</table>

Hmm. These are disappointing results compared with using Modified Standard Score.

It is fun playing with data sets and trying different methods. I obtained the Iris and MPG data sets from the UCI Machine Learning Repository (archive.ics.uci.edu/ml). I encourage you to go there, download a data set or two, convert the data to match data format, and see how well our classifier does.