

A Programmer's Guide to Data Mining



The Ancient Art of the Numerati

Ron Zacharski

A Programmer's Guide to Data Mining: The Ancient Art of the Numerati

www.guidetodatamining.com

by Ron Zacharski

Creative Commons Attribution Noncommercial 3.0 license

Attribution information for all photographs is available on the website.

Thanks to ...

my wife Cheryl



Roper

my son Adam



Roz and Bodhi



also a huge thanks to all the photographers who put their work in the Creative Commons

Preface



If you continue this simple practice every day, you will obtain some wonderful power. Before you attain it, it is something wonderful, but after you attain it, it is nothing special.

Shunryu Suzuki
Zen Mind, Beginner's Mind.

Before you work through this book you might think that systems like Pandora, Amazon's recommendations, and automatic data mining for terrorists, must be very complex and the math behind the algorithms must be extremely complex requiring a PhD to understand. You might think the people who work on developing these systems are like rocket scientists. One goal I have for this book is to pull back this curtain of complexity and show some of the rudimentary methods involved. Granted there are super-smart people at Google, the National Security Agency and elsewhere developing amazingly complex algorithms, but for the most part data mining relies on easy-to-understand principles. Before you start the book you might think data mining is pretty amazing stuff. By the end of the book, I hope you will be able to say nothing special.

The Japanese characters above, Shoshin, represent the concept of Beginner's Mind—the idea of having an open mind that is eager to explore possibilities. Most of us have heard some version of the following story (possibly from Bruce Lee's *Enter the Dragon*). A professor is seeking enlightenment and goes to a wise monk for spiritual direction. The professor dominates the discussion outlining everything he has learned in his life and summarizing papers he has written. The monk asks tea? and begins to pour tea into the professor's cup. And continues to pour, and continues to pour, until the tea over pours the teacup, the table, and spills onto the floor. *What are you doing?* the professor shouts. *Pouring tea* the monk says and continues: *Your mind is like this teacup. It is so filled with ideas that nothing else will go in. You must empty your mind before we can begin.*

To me, the best programmers are empty cups, who constantly explore new technology (noSQL, node-js, whatever) with open minds. Mediocre programmers have surrounded their minds with cities of delusion—C++ is good, Java is bad, PHP is the only way to do web programming, MySQL is the only database to consider. My hope is that you will find some of the ideas in this book valuable and I ask that you keep a beginner's mind when reading it. As Shunryu Suzuki says:

In the beginner's mind there are many possibilities,

In the expert's mind there are few.

Chapter 1 The Intro

Intro to data mining & how to use this book

Imagine life in a small American town 150 years ago. Everyone knows one another. A crate of fabric arrives at the general store. The clerk notices that the pattern of a particular bolt would highly appeal to Mrs. Clancey because he knows that she likes bright floral patterns and makes a mental note to show it to her next time she comes to the store. Chow Winkler mentions to Mr. Wilson, the saloon keeper, that he is thinking of selling his spare Remington rifle. Mr. Wilson mentions that information to Bud Barclay, who he knows is looking for a quality rifle. Sheriff Valquez and his deputies know that Lee Pye is someone to keep an eye on as he likes to drink, has a short temper, and is strong. Life in a small town 100 years ago was all about connections.



People knew your likes and dislikes, your health, the state of your marriage. For better or worse, it was a personalized experience. And this highly personalized life in the community was true throughout most of the world.

Let's jump ahead one hundred years to the 1960s. Personalized interactions are less likely but they are still present. A regular coming into a local bookstore might be greeted with "The new James Michener is in"-- the clerk knowing that the regular loves James Michener books. Or the clerk might recommend to the regular *The Conscience of a Conservative* by Barry Goldwater, because the clerk knows the regular is a staunch conservative. A regular customer comes into a diner and the waitress says "The usual?"

Even today there are pockets of personalization. I go to my local coffee shop in Mesilla and the barista says "A venti latte with an extra shot?" knowing that is what I get every morning. I take my standard poodle to the groomers and the groomer doesn't need to ask what style of clip I want. She knows I like the no frills sports clip with the German style ears.

But things have changed since the small towns of 100 years ago. Large grocery stores and big box stores replaced neighborhood grocers and other merchants. At the start of this change choices were limited. Henry Ford once said "Any customer can have a car painted any color that he wants so long as it is black." The record store carried a limited number of records; the bookstore carried a limited number of books. Want ice cream? The choices were vanilla, chocolate, and maybe strawberry. Want a washing machine? In 1950 you had two choices at the local Sears: the standard model for \$55 or the deluxe for \$95.

Welcome to the 21st century

In the 21st century those limited choices are a thing of the past. I want to buy some music? iTunes has some 11 million tracks to choose from. 11 million! They have sold 16 billion tracks as of October 2011. I need more choices? I can go to Spotify which has over 15 million songs.

I want to buy a book? Amazon has over 2 million titles to choose from.



I want to watch a video? There are plenty of choices:



I want to buy a laptop? When I type in *laptop* into the Amazon search box I get 3,811 results

I type in *rice cooker* and get over 1,000 possibilities.

In the near future there will be even more choice—billions of music tracks online—a wide variety of video—products that can be customized with 3D printing.



Finding Relevant Stuff

The problem is finding relevant stuff. Amid all those 11 million tracks on iTunes, there are probably quite a number that I will absolutely love, but how do I find them. I want to watch a streaming movie from Netflix tonight, what should I watch. I want to download a movie using P2P, but which movie. And the problem is getting worse. Every minute terabytes of media are added to the net. Every minute 100 new files are available on usenet. Every minute 24 hours of video is uploaded to YouTube. Every hour 180 new books are published. Every day there are more and more options of stuff to buy in the real world. It gets more and more difficult to find the relevant stuff in this ocean of possibilities.

If you are a producer of media—say Zee Avi from Malaysia—the danger isn't someone downloading your music illegally—the danger is obscurity.



But how to find stuff?

Years ago, in that small town, our **friends** helped us find stuff. That bolt of fabric that would be perfect for us; that new novel at the bookstore; that new 33 1/3 LP at the record store. Even today we rely on friends to help us find some relevant stuff.

We used **experts** to help us find stuff. Years ago Consumer Reports could evaluate all the washing machines sold—all 20 of them—or all the rice cookers sold-- all 10 of them and make recommendations. Today there are hundreds of different rice cookers available on Amazon and it is unlikely that a single expert source can rate all of them. Years ago, Roger Ebert would review virtually all the movies available. Today about 25,000 movies are made each year worldwide. Plus, we now have access to video from a variety of sources. Roger Ebert, or any single expert, cannot review all the movies that are available to us.

We also use **the thing itself** to help us find stuff. For example, I owned a Sears washing machine that lasted 30 years, I am going to buy another Sears washing machine. I liked one album by the Beatles—I will buy another thinking chances are good I will like that too.

These methods of finding relevant stuff—friends, experts, the thing itself—are still present today but we need some computational help to transform them into the 21st century where we have billions of choices.

These methods of finding relevant stuff—friends, experts, the thing itself—are still present today but we need some computational help to transform them into the 21st century where we have billions of choices. In this book we will explore methods of aggregating people's likes and dislikes, their purchasing history, and other data—exploiting the power of social net (friends)—to help us mine for relevant stuff. We will examine methods that use attributes of the thing itself. For example, I like the band Phoenix. The system might know attributes of Phoenix—that it uses electric rock instrumentation, has punk influences, has a subtle use of vocal harmony. It might recommend to me a similar band that has similar attributes, for example, The Strokes.

It's just not stuff...

Data mining is just not about recommending stuff to us, or having merchants sell more stuff. Consider these examples.

The mayor of that small town of 100 years ago, knew everybody. When he ran for re-election he knew how to tailor what he said to each individual.



Martha, I know you are interested in schools and I will do everything in my power to bring another teacher to town.

John, how is your bakery doing? I promise to get more parking in your area of downtown.

My father belonged to the United Auto Workers' Union. Around election time I remember the union representative coming to our house to remind my father what candidates to vote for:



Frank Zeidler was the Socialist mayor of Milwaukee from 1948 to 1960.

Hey Syl, how are the wife and kids? ... Now let me tell you why you should vote for Frank Zeidler, the Socialist candidate for mayor...

This individualized political message changed to the homogenous ads during the rise of television. Everyone got the exact same message. A good example of this is the famous Daisy television ad in support of

Lyndon Johnson (a young girl pulling petals off a daisy while a nuclear bomb goes off in the background). Now, with elections determined by small margins and the growing use of data mining, individualization has returned. You are interested in a women's right to choose? You might get a robo-call directed at that very issue.

The sheriff of that small town knew who the trouble makers were. Now, threats seem to be hidden, terrorists can be anywhere. On October 11, 2001 the US government passed the USA Patriot Act (short for **U**niting and **S**trengthening **A**merica by **P**roviding **A**ppropriate **T**ools **R**equired to **I**ntercept and **O**bstruct **T**errorism). In part this bill enables investigators to obtain records for a variety of sources including libraries (what books we read), hotels (who stayed where and for how long), credit card companies, toll roads registering that we passed by. For the most part the government uses private companies to keep data on us. Companies like Seisint have data on almost all of us, photos of us, where we live, what we drive, our income, our buying behavior, our friends. Seisint owns supercomputers that use data mining techniques to make predictions about people. Their product by the way is called...



The Matrix.



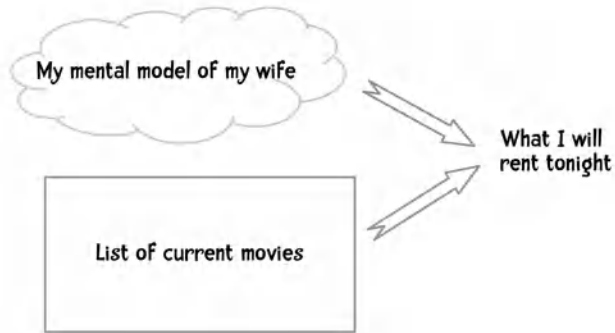
Data Mining Extends what we already do!

Stephen Baker begins his book *The Numerati* this way:

Imagine you are in a café, perhaps the noisy one I'm sitting in at this moment. A young women at a table to your right is typing on her laptop. You turn your head and look at her screen. She surfs the Internet. You watch.

Hours pass. She reads an online paper. You notice that she reads three articles about China. She scouts movies for Friday night and watches the trailer for Kung Fu Panda. She clicks on an ad that promises to connect her to old high school classmates. You sit there taking notes. With each passing minute, you're learning more about her. Now imagine that you could watch 150 million people surfing at the same time.

Data mining is focused on finding patterns in data. At the small scale, we are expert at building mental models and finding patterns. I want to watch a movie tonight with my wife. I have a mental model of what she likes. I know she dislikes violent movies (she didn't like District 9 for that reason). She likes movies by Charlie Kaufman. I can use that mental model I have of her movie preferences to predict what movies she may or may not like.



A friend is visiting from Europe. I know she is a vegetarian and I can use that information to predict she would not like the local rib joint. People are good at making models and making predictions. Data mining expands this ability and enables us to handle large quantities of information—the 150 million people in the Baker quote above. It enables the Pandora Music Service to tailor a music station to your specific musical preferences. It enables Netflix to make specific personalized movie recommendations for you.

Tera-mining is not something from Starcraft II

At the end of the 20th century a million word data set was considered large. When I was a graduate student in the 1990s (yes, I am that ancient) I worked as a programmer for a year on the Greek New Testament. It's only around 200,000 words but the analysis was too large to fit into the mainframe's memory necessitating spooling results off to magnetic tapes, which I had to request to be mounted.

The book resulting from this work is the Analytical Greek New Testament by Timothy and Barbara Friberg (available on Amazon). I was just one of three programmers on this project done at the University of Minnesota.



Today it is not unusual to be doing data mining on terabytes of information. Google has over 5 petabytes (that's 5,000 terabytes) of web data. In 2006 Google released a dataset to the research community based on one trillion words. The National Security Agency has call records for trillions of phone calls. Acxiom, a company that collects information (credit card purchases, telephone records, medical records, car registrations, etc) on 200 million adults in the US, has amassed over 1 petabyte of data.



a 1 petabyte server shipping container

Robert O'Harrow, Jr., author of No Place to Hide, in an effort to help us grasp how much information is 1 petabyte says it is the equivalent of 50,000 miles of stacked King James Bibles. I frequently drive 2,000 between New Mexico and Virginia. When I try to imagine bibles stacked along the entire way that seems like an unbelievable amount of data.



New Mexico to Virginia

The Library of Congress has around 20 terabytes of text. You could store the entire collection of the Library of Congress on a few thousand dollar's worth of hard drives! In contrast, Walmart has over 570 terabytes of data. All this data just doesn't sit there—it is constantly being mined, new associations made, patterns identified. Tera-mining.

Throughout this book we will be dealing with small datasets. It's good thing. We don't want our algorithm to run for a week only to discover we have some error in our logic. The biggest dataset we will use is under 100MB; the smallest just tens of lines of data.

The format of the book.

This book follows a learn-by-doing approach. Instead of passively reading the book, I encourage you to work through the exercises and experiment with the Python code I provide. Experimenting around, code hacking, and trying out methods with different data sets is the key to really gaining an understanding for the techniques.



I try to strike a balance between hands-on, nuts-and-bolts discussion of Python data mining code that you can use and modify, and the theory behind the data mining techniques. To try to prevent the brain freeze associated with reading theory, math, and Python code, I tried to stimulate a different part of your brain by adding drawings and pictures.

Peter Norvig, Director of Research at Google, had this to say in his great Udacity course. *Design of a Computer Program*:

“I’ll show you and discuss my solution. It’s important to note, there is more than one way to approach a problem. And I don’t mean that my solution is the ONLY way or the BEST way. My solutions are there to help you learn a style and some techniques for programming. If you solve problems a different way, that’s fine. Good for you.

All the learning that goes on happens inside of your head. Not inside of my head. So what’s important is that you understand the relation between your code and my code, that you get the right answer by writing out the solution yourself and then you can examine my code and maybe pick out some pointers and techniques that you can use later.”

I couldn’t agree more!



This book is not a comprehensive textbook on data mining techniques. There are textbooks, like *Introduction to Data Mining* by Pang-Ning Tan, Michael Steinbach, and Vipin Kumar that provide significantly better coverage of data mining methods and provide more in-depth analysis of the mathematic underpinnings of these methods. This book—the one you are holding—is intended more as a quick, gritty, hands-on introduction designed to give you a basic foundation of data mining techniques. Later, you can pick up a more comprehensive book to fill in any gaps that you wish.

Part of the usefulness of this book is the accompanying Python code and the datasets. I think the inclusion of both these make it easier for the learner to understand key concepts, but at the same time, not shoe-horn the learner into a scripted exploration.

What will you be able to do when you finish this book?

When you finish this book you will be able to design and implement recommendation systems for websites using Python or any language you know. For example, when you look at a product on Amazon, or a tune on Pandora, you are presented with a list of recommendations (You might also like ...). You will learn how to develop such systems. In addition, the book should provide you with the necessary vocabulary to enable you to work in development teams on data mining efforts.

As part of this goal, this book should help shed the mystery of recommendation systems, terrorist identification systems, and other data mining systems. You should at least have a rough idea of how they work.

Why – why does this matter?

Why should you use your time reading (and working through) this book on data mining? At the beginning of this chapter I gave examples related to the importance of data mining. The summary of that section would go as follows. There's lots of stuff out there (movies, music, books, rice cookers). There's going to be a huge growth in the amount of stuff out there. The problem with having all this stuff available is finding the stuff that is relevant to us. Of all the movies out there, what movie should I watch. What's the next book I should read? This problem of identifying relevant stuff is what data mining is about. Most websites will have some component dealing with 'finding stuff'. In addition to the movies, music, books, and rice cookers mentioned above, you might want recommendations about what friends to follow. How about a personalized newspaper showing just the news you are most interested in? If you are a programmer, particularly a web developer, it would be useful to know data mining techniques.

Okay, so you can see the reason to devote some of your time to learning data mining, but why this book? There are books that give you a non-technical overview of data mining. They are a quick read, entertaining, inexpensive, and can be read late at night (no hairy technical bits). A great example of this is *The Numerati* by Stephen Baker. I recommend this book—I listened to the audio version of it while driving between Virginia and New Mexico. It was engrossing. On the other extreme are college textbooks on data mining. They are

comprehensive and provide an in-depth analysis of data mining theory and practice. Again, I recommend books in this category. I wrote this book to fill a gap. It's a book designed for people who love to program—hackers.



The book is intended to be read at a computer so the reader can participate and mess with code.

Eeeks!

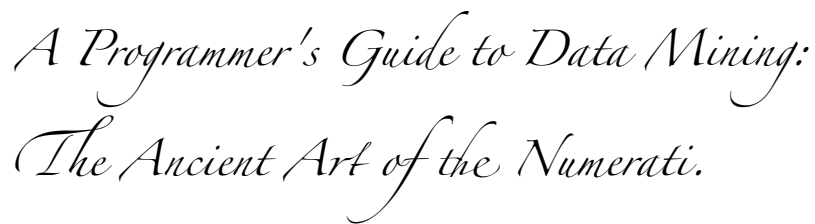
The book has math formulas but I try to explain them in a way that is intelligible to average programmers, who may have forgotten a hunk of the math they took in college.

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

If that doesn't convince you, this book is also free (as in no cost) and free as in you can share it.

What's with the 'Ancient Art of the Numerati' part of the title

In June of 2010 I was trying to come up with a title for this book. I like clever titles, but unfortunately, I have no talent in the area. I recently published a paper titled *Linguistic Dumpster Diving: Geographical Classification of Arabic Text* (yep, a data mining paper). I like the title and it is clever because it fits with the content of the paper, but I have to confess my wife came up with the title. I co-wrote a paper *Mood and Modality: Out of the theory and into the fray*. My co-author Marjorie McShane came up with the title. Anyway, back to June, 2010. All my clever title ideas were so vague that you wouldn't have a clue what the book was about. I finally settled on *A Programmer's Guide to Data Mining* as part of the title. I believe that bit is a concise description of the content of the book—I intend the book be a guide for the working programmer. You might wonder what is the meaning of the part after the colon:



*A Programmer's Guide to Data Mining:
The Ancient Art of the Numerati.*

The Numerati is a term coined by Stephen Baker. Each one of us generates an amazing amount of digital data everyday. credit card purchases, Twitter posts, Gowalla posts, Foursquare check-ins, cell phone calls, email messages, text messages, etc.

You get up. The Matrix knows you boarded the subway at the Foggy Bottom Station at 7:10 and departed the Westside Station at 7:32. The Matrix knows you got a venti latte and a blueberry scone at the Starbucks on 5th and Union at 7:45; you used Gowalla to check-in at work at 8:05; you made an Amazon purchase for the P90X Extreme Home Fitness Workout Program 13 DVD set and a chin-up bar at 9:35; you had lunch at the Golden Falafel.

Stephen Baker writes:

The only folks who can make sense of the data we create are crack mathematicians, computer scientists, and engineers. What will these Numerati learn about us as they run us into dizzying combinations of numbers? First they need to find us. Say you're a potential SUV shopper in the northern suburbs of New York, or a churchgoing, antiabortion Democrat in Albuquerque. Maybe you're a Java programmer ready to relocate to Hyderabad, or a jazz-loving, Chianti-sipping Sagittarius looking for walks in the country and snuggles by the fireplace in Stockholm, or—heaven help us—maybe you're eager to strap bombs to your waist and climb onto a bus. Whatever you are—and each of us is a lot of things—companies and governments want to identify and locate you.

Baker

As you can probably guess, I like this term *Numerati* and Stephen Baker's description of it.



Chapter 2: Collaborative Filtering

I like what you like

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



Customers Who Viewed This Item Also Bought

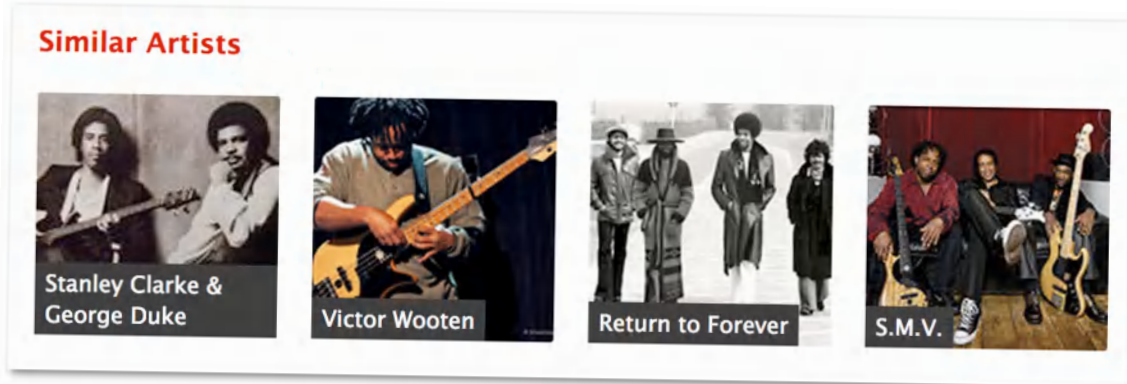
The Diamond Sutra
Red Pine
★★★★★ (20)
Paperback
\$13.57

The Heart Sutra
Red Pine
★★★★★ (21)
Paperback
\$10.17

The Lotus Sutra
Burton Watson
★★★★★ (27)
Paperback
\$18.21

The image shows a horizontal banner with a left-pointing arrow on the left side. It features three book covers with their respective titles, authors, star ratings, and prices. Each book cover has a small 'LOOK INSIDE!' banner at the top.

to last.fm recommending music or concerts:



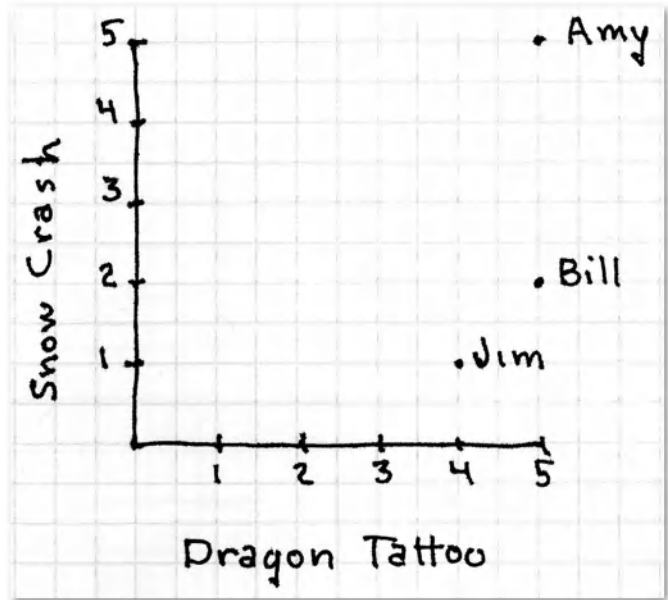
In the Amazon example, above, Amazon combines two bits of information to make a recommendation. The first is that I viewed *The Lotus Sutra* translated by Gene Reeves; the second, that customers who viewed that translation of the *Lotus Sutra* also viewed several other translations.

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

How do I find someone who is similar?

So the first step is to find someone who is similar. Here's the simple 2D (dimensional) explanation.

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



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

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

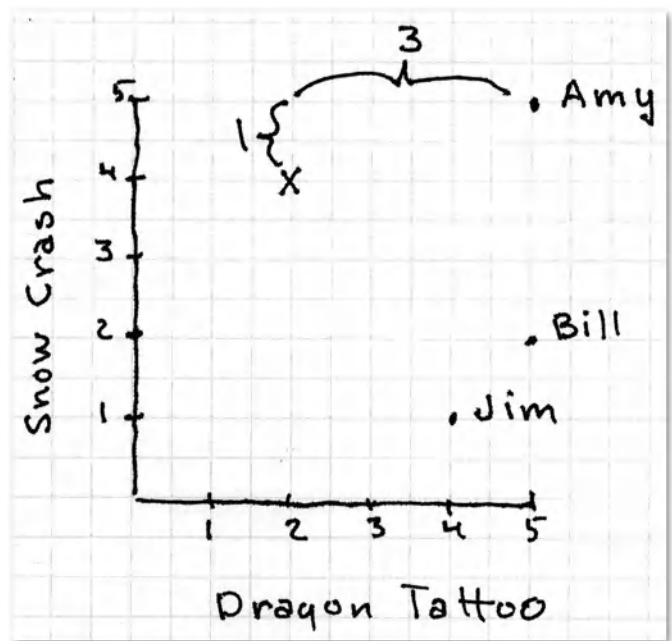
I would like to recommend a book to the mysterious Ms. X who rated *Snow Crash* 4 stars and *The Girl with the Dragon Tattoo* 2 stars. The first task is to find the person who is most similar, or closest, to Ms. X. I do this by computing distance.

Manhattan Distance

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

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

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



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

	Distance from Ms. X
Amy	4
Bill	5
Jim	5

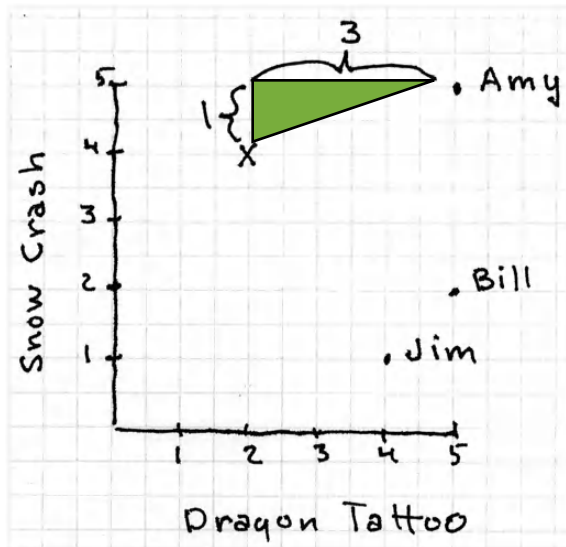
Amy is the closest match. We can look in her history and see, for example, that she gave five stars to Paolo Bacigalupi's *The Windup Girl* and we would recommend that book to Ms. X.

Euclidean Distance

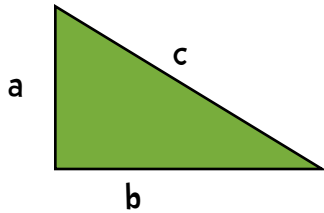
One benefit of Manhattan Distance is that it is fast to compute. If we are Facebook and are trying to find who among one million users is most similar to little Danny from Kalamazoo, fast is good.

Pythagorean Theorem

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



The Pythagorean Theorem tells us how to compute that distance.



$$c = \sqrt{a^2 + b^2}$$

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

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

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

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

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

Computing the rest of the distances we get

	Distance from Ms. X
Amy	3.16
Bill	3.61
Jim	3.61

N-dimensional thinking

Let's branch out slightly from just looking at rating two books (and hence 2D) to looking at something slightly more complex. Suppose we work for an online streaming music service and we want to make the experience more compelling by recommending bands. Let's say users can rate bands on a star system 1-5 stars and they can give half star ratings (for example, you can give a band 2.5 stars). The following chart shows 8 users and their ratings of eight bands.

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

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

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

The Manhattan Distance row, the last row of the table, is simply the sum of the differences: $(1.5 + 1.5 + 3 + 2 + 1)$.

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

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

To parse that out a bit more:

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

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

$$= \sqrt{2.25 + 2.25 + 9 + 4 + 1}$$

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



Got it?

Try an example on your own...

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



sharpen your pencil

Compute the Euclidean Distance between Hailey and Veronica.

Compute the Euclidean Distance between Hailey and Jordyn



sharpen your pencil - solution

Compute the Euclidean Distance between Hailey and Veronica.

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

Compute the Euclidean Distance between Hailey and Jordyn

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

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

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

A Flaw

It looks like we discovered a flaw with using these distance measures. When we computed the distance between Hailey and Veronica, we noticed they only rated two bands in common (Norah Jones and The Strokes), whereas when we computed the distance between Hailey and Jordyn, we noticed they rated five bands in common. This seems to skew our distance measurement, since the Hailey-Veronica distance is in 2 dimensions while the Hailey-Jordyn

distance is in 5 dimensions. Manhattan Distance and Euclidean Distance work best when there are no missing values. Dealing with missing values is an active area of scholarly research. Later in the book we will talk about how to deal with this problem. For now just be aware of the flaw as we continue our first exploration into building a recommendation system.

A generalization

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

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

When

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



Arghhhh Math!

When you see formulas like this in a book you have several options. One option is to see the formula--brain neurons fire that say *math formula*--and then you quickly skip over it to the next English bit. I have to admit that I was once a skipper. The other option is to see the formula, pause, and dissect it.



Many times you'll find the formula quite understandable. Let's dissect it now. When $r = 1$ the formula reduces to Manhattan Distance:

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

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

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

That difference column represents the absolute value of the difference and we sum those up to get 9.

When $r = 2$, we get the Euclidean distance:

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



Here's the scoop!

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

Representing the data in Python (finally some coding)

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

Remember,

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

```

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

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

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

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

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

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

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

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

```

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

```

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

>>>

```

The code to compute Manhattan distance

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

```
def manhattan(rating1, rating2):
    """Computes the Manhattan distance. Both rating1 and rating2 are
    dictionaries of the form
    {'The Strokes': 3.0, 'Slightly Stoopid': 2.5 ...}"""

    distance = 0
    for key in rating1:
        if key in rating2:
            distance += abs(rating1[key] - rating2[key])
    return distance
```

To test the function:

```
>>> manhattan(users['Hailey'], users['Veronica'])
2.0
>>> manhattan(users['Hailey'], users['Jordyn'])
7.5
>>>
```

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

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

And just a quick test of that function:

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

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

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

And now to make recommendations for Hailey:

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

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

```
>>> recommend('Chan', users)
```



```
[('The Strokes', 4.0), ('Vampire Weekend', 1.0)]
```

```
>>> recommend('Sam', users)
[('Deadmau5', 1.0)]
```

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

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

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

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

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

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

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

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



exercise

1) Implement the Minkowski Distance function.

2) Alter the `computeNearestNeighbor` function to use Minkowski Distance.



exercise - solution

1) Implement the Minkowski Distance function.

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

2) Alter the computeNearestNeighbor function to use Minkowski Distance.

just need to alter the distance = line to

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

(the 2 as the r argument implements Euclidean)

Blame the users

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

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

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

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

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

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

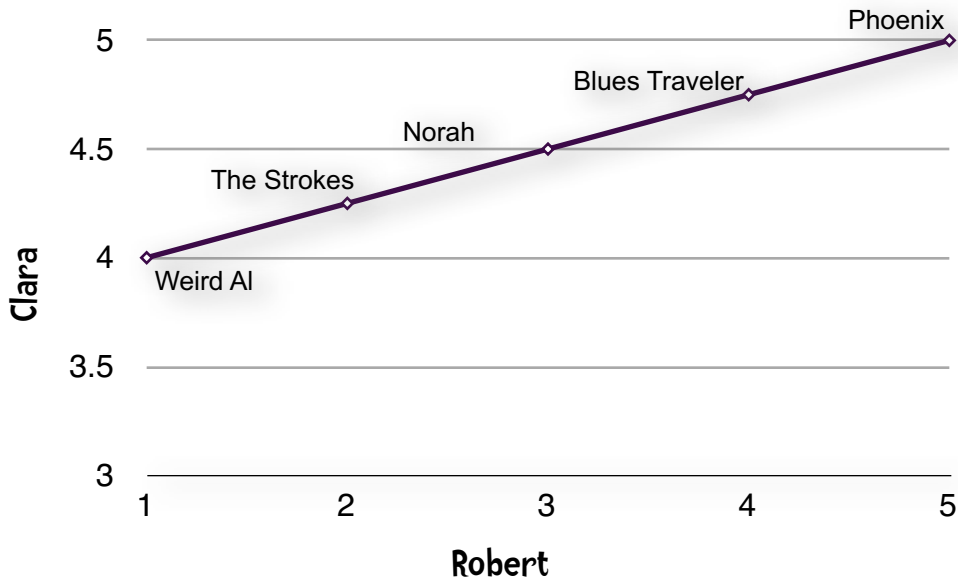


Pearson Correlation Coefficient

One way to fix this problem is to use the Pearson Correlation Coefficient. First, the general idea. Consider the following data (not from the data set above):

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

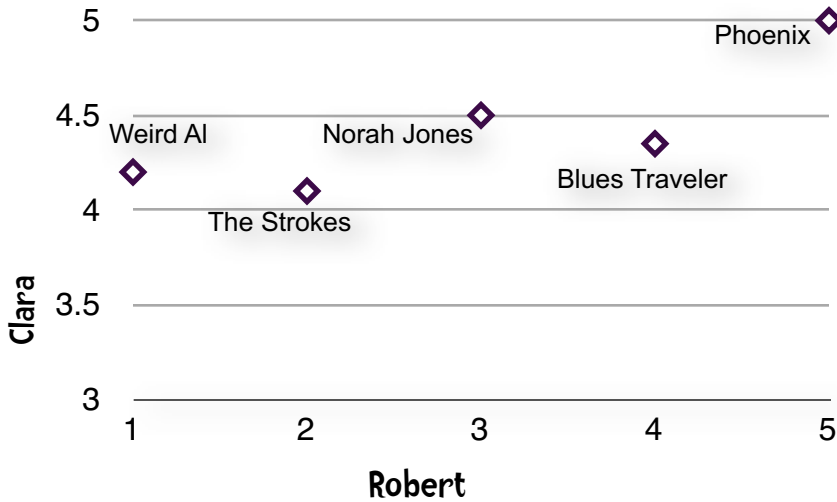
This is an example of what is called 'grade inflation' in the data mining community. Clara's lowest rating is 4—all her ratings are between 4 and 5. If we are to graph this chart it would look like



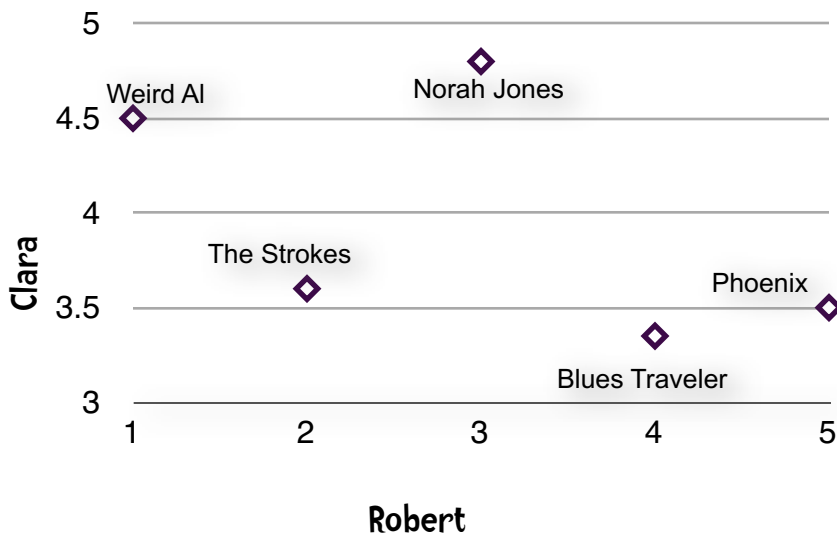
Straight line = Perfect Agreement!!!

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

Pretty Good Agreement:



Not So Good Agreement:



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

The formula for the Pearson Correlation Coefficient is

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



Arghhhh Math Again!

Here's a personal confession. I have a Bachelor of Fine Arts degree in music. While I have taken courses in ballet, modern dance, and costume design, I did not have a single math course as an undergrad. Before that, I attended an all boys trade high school where I took courses in plumbing and automobile repair, but no courses in math other than the basics. Either due to this background or some innate wiring in my brain, when I read a book that has formulas like the one above, I tend to skip over the formulas and continue with the text below them. If you are like me I would urge you to fight



that urge and actually look at the formula. Many formulas that on a quick glimpse look complex are actually understandable by mere mortals.

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

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

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

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

To start with, let us compute

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

Which is in the first expression in the numerator. Here the x and y represent Clara and Robert.

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

For each band we are going to multiple Clara's and Robert's rating together and sum the results:

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

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

Sweet! Now let's compute the rest of the numerator:

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

So the

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

is the sum of Clara's ratings, which is 22.5. The sum of Robert's is 15 and they rated 5 bands:

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

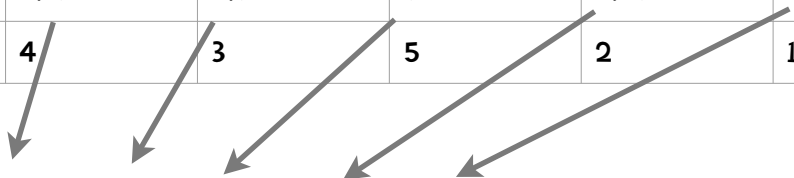
So the numerator in the formula on page 26 is $70 - 67.5 = 2.5$

Now let's dissect the denominator.

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

First,

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



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

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

Putting that together:

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

Next we do the same computation for Robert:

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

Putting this altogether we get:

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

So 1 means there was perfect agreement between Clara and Robert!

Take a break before moving on!!





exercise

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

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

For this implementation you will need 2 Python functions `sqrt` (square root) and power operator `**` which raises its left argument to the power of its right argument:

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



exercise - solution

Here is my implementation of Pearson

```
def pearson(rating1, rating2):
    sum_xy = 0
    sum_x = 0
    sum_y = 0
    sum_x2 = 0
    sum_y2 = 0
    n = 0
    for key in rating1:
        if key in rating2:
            n += 1
            x = rating1[key]
            y = rating2[key]
            sum_xy += x * y
            sum_x += x
            sum_y += y
            sum_x2 += x**2
            sum_y2 += y**2
    # if no ratings in common return 0
    if n == 0:
        return 0
    # now compute denominator
    denominator = sqrt(sum_x2 - (sum_x**2) / n) *
                  sqrt(sum_y2 - (sum_y**2) / n)
    if denominator == 0:
        return 0
    else:
        return (sum_xy - (sum_x * sum_y) / n) / denominator
```

One last Formula – Cosine Similarity

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

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

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

So what is the problem?

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

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

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



A similar case can be made when we are comparing text documents using words. Suppose we liked a certain book, say *Tom Corbett Space Cadet: The Space Pioneers* by Carey Rockwell and we want to find a similar book. One possible way is to use word frequency. The attributes will be individual words and the values of those attributes will be the frequency of those words in the book. So 6.13% of the words in *The Space Pioneers* are occurrences of the word *the*, 0.89% are the word *Tom*, 0.25% of the words are *space*. I can compute the similarity of this book to others by using these word frequencies. However, the same problem related to sparseness of data occurs here. There are 6,629 unique words in *The Space Pioneers* and there are a bit over one million unique words in English. So if our attributes are English words, there will be relatively few non-zero attributes for *The Space*

Pioneers or any other book. Again, any measure of similarity should not depend on the shared-zero values.

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

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

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

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

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

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

The two vectors are:

$$x = (4.75, 4.5, 5, 4.25, 4)$$

$$y = (4, 3, 5, 2, 1)$$

then

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

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

The dot product is

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

And the cosine similarity is

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

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



sharpen your pencil

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

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



sharpen your pencil - solution

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

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

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

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

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

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

The dot product is

$$x \cdot y =$$

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

Cosine Similarity is

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

Which similarity measure to use?

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

If the data is subject to grade-inflation (different users may be using different scales) use Pearson.

If your data is dense (almost all attributes have non-zero values) and the magnitude of the attribute values is important, use distance measures such as Euclidean or Manhattan.

If the data is sparse consider using Cosine Similarity.

Good job, guys, nailed it!



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



Jake: hardcore Fan of Country



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

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

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

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

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

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



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

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

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



Cooper



Kelsey

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

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

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

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

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

Why?

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

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

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

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

$$\begin{aligned} d_{Kelsey, Jake} &= (5 - 0) + (4 - 0) + (4 - 0) + (5 - 4.5) + (5 - 5) + (5 - 4.5) + (5 - 0) \\ &\quad + (5 - 0) + (4 - 0) + (4 - 0) \\ &= 5 + 4 + 4 + 0.5 + 0 + .5 + 5 + 5 + 4 + 4 = 32 \end{aligned}$$

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

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

Weirdnesses

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



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



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

K-nearest neighbor

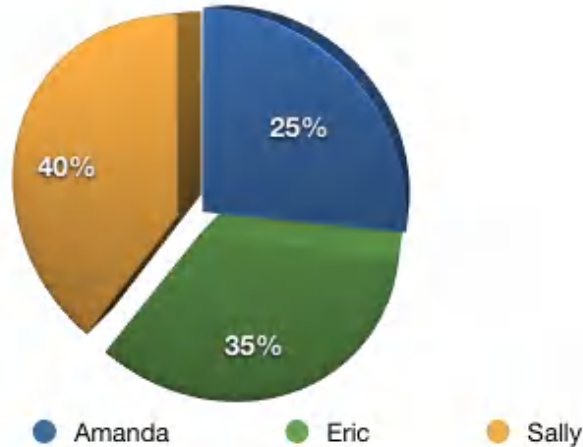
In the k -nearest neighbor approach to collaborative filtering we use k most similar people to determine recommendations. The best value for k is application specific—you will need to do some experimentation. Here's an example to give you the basic idea.

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

Person	Pearson
Sally	0.8
Eric	0.7
Amanda	0.5

$$0.8 + 0.7 + 0.5 = 2.0$$

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



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

Person	Grey Wardens Rating
Amanda	4.5
Eric	5
Sally	3.5

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

$$\text{Projected rating} = (4.5 \times 0.25) + (5 \times 0.35) + (3.5 \times 0.4)$$

$$= 4.275$$



sharpen your pencil

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

Person	Pearson
Sally	0.8
Eric	0.7
Amanda	0.5

Person	Grey Wardens Rating
Amanda	4.5
Eric	5
Sally	3.5



solution

Person	Pearson
Sally	0.8
Eric	0.7
Amanda	0.5

Person	Grey Wardens Rating
Amanda	4.5
Eric	5
Sally	3.5

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

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

$$= (3.5 \times .5333) + (5 \times 0.4667)$$

$$= 1.867 + 2.333$$

$$= 4.2$$

A Python Recommendation Class

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

```
import codecs
from math import sqrt

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

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

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

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

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

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

```

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

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

```

```
class recommender:
```

```

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

```

```

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

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

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

```



```

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

```

```

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

```

```

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

```

```

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

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

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

```

```

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

```

Example of this program executing

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

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

Some simple examples using these band ratings:

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

A New Dataset

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

The CSV files represent three tables:

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

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

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

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

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

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

```
>>> r.recommend('17118')
[("The Godmother's Web by Elizabeth Ann Scarborough", 10.0), ("The Irrational Season (The Crosswicks Journal, Book 3) by Madeleine L'Engle", 10.0), ("The Godmother's Apprentice by Elizabeth Ann Scarborough", 10.0), ("A Swiftly Tilting Planet by Madeleine L'Engle", 10.0), ('The Girl Who Loved Tom Gordon by Stephen King', 9.0), ('The Godmother by Elizabeth Ann Scarborough', 8.0)]
```

```
>>> r.userRatings('17118', 5)
Ratings for toronto, ontario, canada
2421
The Careful Writer by Theodore M. Bernstein    10
Wonderful Life: The Burgess Shale and the Nature of History by Stephen Jay Gould 10
Pride and Prejudice (World's Classics) by Jane Austen    10
The Wandering Fire (The Fionavar Tapestry, Book 2) by Guy Gavriel Kay    10
Flowering trees and shrubs: The botanical paintings of Esther Heins by Judith Leet    10
```

Projects

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

1. Implement Manhattan distance and Euclidean distance and compare the results of these three methods.
2. The book website has a file containing movie ratings for 25 movies. Create a function that loads the data into your classifier. The recommend method described above should recommend movies for a specific person.

Chapter 3: Collaborative Filtering

Implicit ratings and item based filtering

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

Explicit ratings

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



Foundation 05 // Brian Wong

kevinrose + Subscribe 54 videos ▾

9,504

233 likes, 1 dislike

Uploaded by kevinrose on Jul 7, 2011

Kevin Rose interviews Brian Wong, CEO/Founder of Kiip

And Amazon's star system:

Most Recent Customer Reviews

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

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

Published 29 days ago by M. Gonzales

★★★★☆ **The Long Sutra**

Gene Reeves did an excellent job in translating The Lotus Sutra from the Chinese into English. It is here, the entire work, not abridged. [Read more](#)

Published 8 months ago by Eric Maroney

Implicit Ratings

For implicit ratings, we don't ask users to give any ratings—we just observe their behavior. An example of this is keeping track of what a user clicks on in the online New York Times.



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



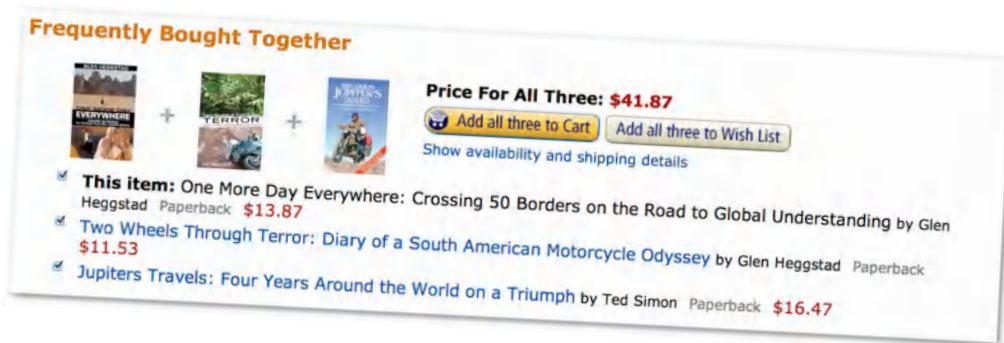
Consider what information we can gain from recording what products a user clicks on in Amazon. On your personalized Amazon front page this information is displayed:

More Items to Consider

You viewed	Customers who viewed this also viewed		
<p>Jupiters Travels: Four Years Around... Ted Simon Paperback ★★★★★ (65) \$24.95 \$16.47</p>	<p>Long Way Round Ewan McGregor, Charley Boorman, ... DVD ★★★★★ (325) \$24.95 \$16.93</p>	<p>One More Day Everywhere: Crossing 50... Glen Heggstad Paperback ★★★★★ (47) \$18.95 \$13.87</p>	<p>Dreaming of Jupiter Ted Simon Paperback ★★★★★ (6) \$16.22</p>

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

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



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

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

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

First, there's the fact that I added a song to iTunes. That indicates minimally that I was interested enough in the song to do so. Then there is the Play Count information. In the image above, I've listened to Zee Avi's "Anchor" 52 times. That suggests that I like that song (and in fact I do). If I have a song in my library for awhile and only listened to it once, that might indicate that I don't like the song.



brain calisthenics

Do you think having a user explicitly give a rating to an item is more accurate?

Or do you think watching what a user buys or does (for example, the play count) is a more accurate judge of what an individual likes?

Jim



Explicit Rating:
match.com bio:

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

Implicit Ratings:



what we found in
Jim's pocket

Receipts for:

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

Problems with explicit ratings

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

First, users will typically not bother to rate items. I imagine most of you have bought a substantial amount of stuff on Amazon. I know I have. In the last month I bought a microHelicopter, a 1TB hard drive, a USB-SATA converter, a bunch of vitamins, two Kindle books (*Murder City: Ciudad Juarez and the Global Economy's New Killing Fields* and *Ready Player One*) and the physical books *No Place to Hide*, *Dr. Weil's 8 Weeks to Optimum Health*, *Anticancer: A new way of life*, and *Rework*. That's twelve items. How many have I rated? Zero. I imagine most of you are the same. You don't rate the items you buy.

I have a gimp knee. I like hiking in the mountains and as a result own a number of trekking poles including some cheap ones I bought on Amazon that have taken a lot of abuse. Last year I flew to Austin for the 3 day Austin City Limits music festival. I aggravated my knee injury dashing from one flight to another and ended up going to REI to buy a somewhat pricey REI branded trekking pole. It broke in less than a day of walking on flat grass at a city park. Here I own \$10 poles that don't break during constant use of hiking around in the Rockies and this pricey model broke on flat ground. At the time of the festival, as I was fuming, I planned to rate and write a review of the pole on the REI site. Did I? No, I am too lazy. So even in this extreme case I didn't rate the item. I think there are a lot of lazy people like me. People in general are too lazy or unmotivated to rate products.



my slightly bent REI pole ↩

Problem 2: People may lie or give only partial information.

Let's say someone gets over that initial laziness and actually rates a product. That person may lie. This is illustrated in the drawing a few pages back. They can lie directly—giving inaccurate ratings or lie via omission—providing only partial information. Ben goes on a first date with Ann to see the 2010 Cannes Film Festival Winner, a Thai film, *Uncle Boonmee Who Can Recall His Past Lives*. They go with Ben's friend Dan and Dan's friend Clara. Ben thinks it was the worst film he ever saw. All the others absolutely loved it and gushed about it afterwards at the restaurant. It would not be surprising if Ben upped his rating of the film on online rating sites that his friends might see or just not rate the film.

Problem 3: People don't update their ratings.

Suppose I am motivated by writing this chapter to rate my Amazon purchases. That 1TB hard drive works well—it's very speedy and also very quiet. I rate it five stars. That microHelicopter is great. It is easy to fly and great fun and it survived multiple crashes. I rate it five stars. A month goes by. The hard drive dies and as a result I lose all my downloaded movies and music—a major bummer. The microHelicopter suddenly stops working—it looks like the motor is fried. Now I think both products suck. Chances are pretty good that I will not go to Amazon and update my ratings (laziness again). People still think I would rate both 5 stars.



Consider Mary, a college student. For some reason, she loves giving Amazon ratings. Ten years ago she rated her favorite music albums with five stars: Giggling and Laughing: Silly Songs for Kids, and Sesame Songs: Sing Yourself Silly! Her most recent ratings included 5 stars for Wolfgang Amadeus Phoenix and The Twilight Saga: Eclipse Soundtrack. Based on these recent ratings she ends up being the closest neighbor to another college student Jen. It would be odd to recommend Giggling and Laughing: Silly Songs for Kids to Jen. This is a slightly different type of update problem than the one above, but a problem none-the-less.



brain calisthenics

What do you think are the problems with implicit ratings?

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

A few pages ago I gave a list of items I bought at Amazon in the last month. It turns out I bought two of those items for other people. I bought the anticancer book for my cousin and the Rework book for my son. To see why this is a problem, let me come up with a more compelling example by going further back in my purchase history. I bought some kettlebells and the book *Enter the Kettlebell! Secret of the Soviet Supermen* as a gift for my son and a Plush Chase Border Collie stuffed animal for my wife because our 14-year-old border collie died. Using purchase history as an implicit rating of what a person likes, might lead you to believe that people who like kettlebells, like stuffed animals, like microHelicopters, books on anticancer, and the book *Ready Player One*. Amazon's purchase history can't distinguish between purchases for myself and purchases I make as gifts. Stephen Baker describes a related example:



Baker 2008.60-61.

Figuring out that a certain white blouse is business attire for a female baby boomer is merely step one for the computer. The more important task is to build a profile of the shopper who buys that blouse. Let's say it's my wife. She goes to Macy's and buys four or five items for herself. Underwear, pants, a couple of blouses, maybe a belt. All of the items fit that boomer profile. She's coming into focus. Then, on the way out she remembers to buy a birthday present for our 16-year-old niece. Last time we saw her, this girl was wearing black clothing with a lot of writing on it, most of it angry. She told us she was a goth. So my wife goes into an "alternative" section and—what the hell—picks up one of those dog collars bristling with sharp spikes.

If we are attempting to build a profile of a person—what a particular person likes—this dog collar purchase is problematic.

Finally, consider a couple sharing a Netflix account. He likes action flicks with lots of explosions and helicopters; she likes intellectual movies and romantic comedies. If we just look at rental history, we build an odd profile of someone liking two very different things.

Recall that I said my purchase of the book *Anticancer: A New Way of Life* was as a gift to my cousin. If we mine my purchase history a bit more we would see that I bought this book before. In fact, in the last year I purchased multiple copies of three books. One can imagine that I am making these multiple purchases not because I am losing the books, or that I am losing my mind and forgetting that I read the books. The most rational reason, is that I liked the books so much I am in a sense recommending these books to others by giving them as gifts. So we can gain a substantial amount of information from a person's purchase history.



brain calisthenics

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



Implicit Data:

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

This just scratches the surface!

Keep in mind that the algorithms described in chapter 2 can be used regardless of whether the data is explicit or implicit.

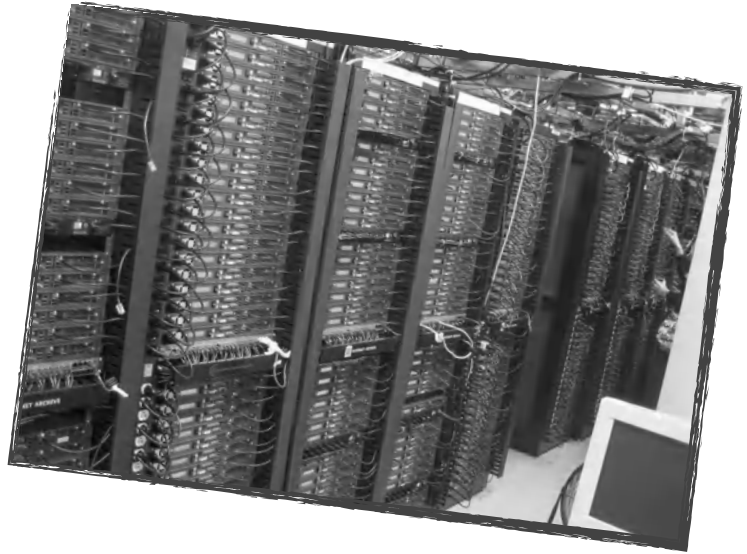
The problems of success

You have a successful streaming music service with a built in recommendation system. What could possibly go wrong?

Suppose you have one million users. Every time you want to make a recommendation for someone you need to calculate one million distances (comparing that person to the 999,999 other people). If we are making multiple recommendations per second, the number of calculations get extreme. Unless you throw a lot of iron at the problem the system will get slow. To say this in a more formal way, latency can be a major drawback of neighbor-based

recommendation systems. Fortunately, there is a solution.

Lots of iron:
a server farm



User-based filtering.

So far we have been doing user-based collaborative filtering. We are comparing a user with every other user to find the closest matches. There are two main problems with this approach:

1. **Scalability.** As we have just discussed, the computation increases as the number of users increases. User-based methods work fine for thousands of users, but scalability gets to be a problem when we have a million users.
2. **Sparsity.** Most recommendation systems have many users and many products but the average user rates a small fraction of the total products. For example, Amazon carries millions of books but the average user rates just a handful of books. Because of this the algorithms we covered in chapter 2 may not find any nearest neighbors.

Because of these two issues it might be better to do what is called item-based filtering.

Item-based Filtering.

Suppose I have an algorithm that identifies products that are most similar to each other. For example, such an algorithm might find that Phoenix's album *Wolfgang Amadeus Phoenix* is similar to Passion Pit's album, *Manners*. If a user rates *Wolfgang Amadeus Phoenix* highly we could recommend the similar album *Manners*. Note that this is different than what we did for user-based filtering. In user-based filtering we had a user, found the most similar person (or users) to that user and used the ratings of that similar person to make recommendations. In item-based filtering, ahead of time we find the most similar items, and combine that with a user's rating of items to generate a recommendation.

Can you give me an example?

Suppose our streaming music site has m users and n bands, where the users rate bands. This is shown in the following table. As before, the rows represent the users and the columns represent bands.

	Users	...	Phoenix	...	Passion Pit	...	n
1	Tamera Young		5				
2	Jasmine Abbey				4		
3	Arturo Alvarez		1		2		
...	...						
u	Cecilia De La Cueva		5		5		
...	...						
m-1	Jessica Nguyen		4		5		
m	Jordyn Zamora		4				

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

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



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



Adjusted Cosine Similarity.

To compute the similarity between items we will use Cosine Similarity which was introduced in chapter 2. We also already talked about grade inflation where a user gives higher ratings than expected. To compensate for this grade inflation we will subtract the user's average rating from each rating. This gives us the adjusted cosine similarity formula shown on the following page.

I like Phoenix, I'll give them a '5'. I don't like Passion Pit, I'll give them a '3'!

Phoenix is awesome, They're definitely a '4'. Passion Pit sucks. A definite 0!



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

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

This formula is from a seminal article in collaborative filtering: “Item-based collaborative filtering recommendation algorithms” by Badrul Sarwar, George Karypis, Joseph Konstan, and John Reidl (http://www.grouplens.org/papers/pdf/www10_sarwar.pdf)

$$(R_{u,i} - \bar{R}_u)$$

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

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

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

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

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

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

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

Ben's ratings

Chris's ratings

Tori's ratings

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

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

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

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



sharpen your pencil

Compute the rest of the values in the table above!



sharpen your pencil - solution

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

To compute these values I wrote a small Python script:

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

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

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



sharpen your pencil - solution cont'd

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

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

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



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

English, please!

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

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

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

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

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

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

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

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

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





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

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

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

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

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

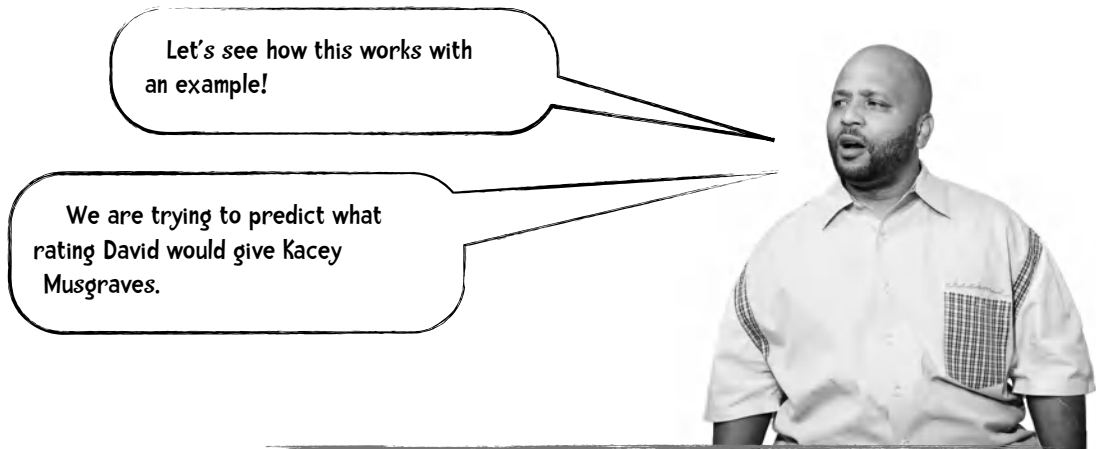
$$NR_{u,N} = \frac{2(R_{u,N} - Min_R) - (Max_R - Min_R)}{(Max_R - Min_R)} = \frac{2(2 - 1) - (5 - 1)}{(5 - 1)} = \frac{-2}{4} = -0.5$$

and to go the other way ...

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

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

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



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

David's Ratings

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

We will learn more about normalization in the next chapter!

Similarity Matrix

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

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

And we will be using the normalized ratings!



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

$$\frac{\text{Imagine Dragons} \quad \text{Daft Punk} \quad \text{Lorde} \quad \text{Fall Out Boy}}{(.5260 \times 0) + (1.00 \times 1) + (.321 \times 0.5) + (-.955 \times -1)} = \frac{0.5260 + 1.000 + 0.321 + 0.955}{}$$

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

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 Similarity is a Model-Based Collaborative 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 that 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

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

$P^{wS1}(u)_j$ 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 ($\text{item2} \neq \text{item}$):

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 $270,000 \times 270,000$ 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.

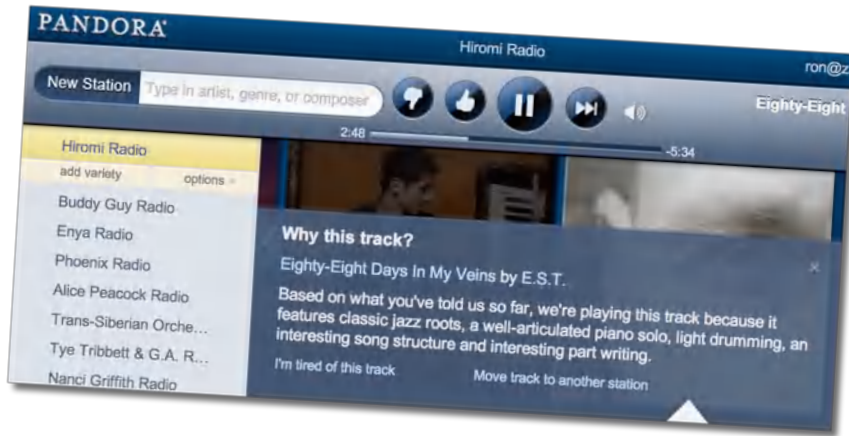
"These recommenders can create a rich-get-richer effect for popular products and vice-versa for unpopular ones"

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

E1 Ten Eleven		My Only Swerving	
Beats per Minute:	110	major tonality:	5
swinging 16ths:	0	electric guitar riffs:	5
well articulated piano solo:	2	repetitive melodic phrasing:	4
block chords:	3	drumming:	3
acoustic instrumentation:	5	electric instrumentation:	4

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.

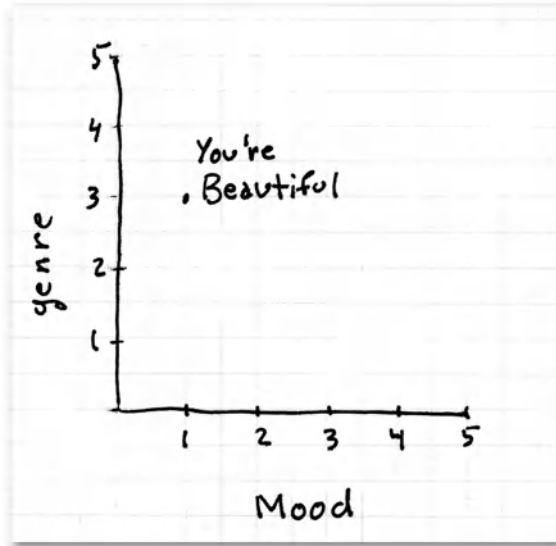
The importance of selecting appropriate values

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:



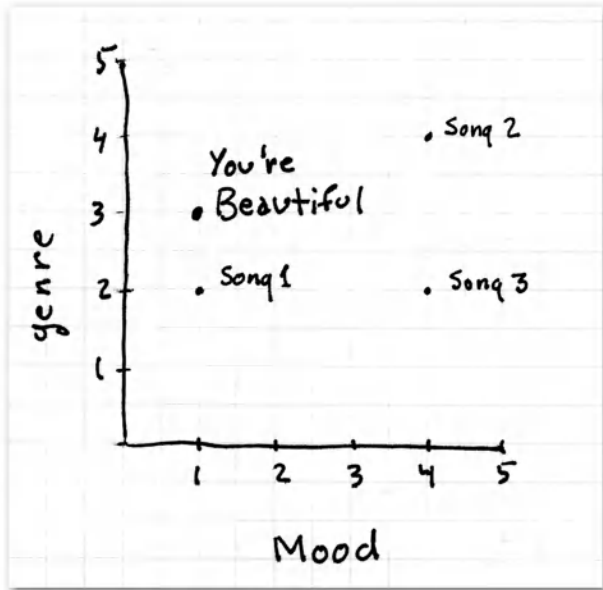
FACT:
In a Rolling Stone poll on the Most Annoying Songs ever, You're Beautiful placed #7!

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



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?



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

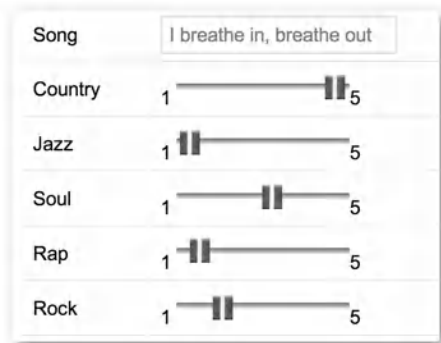
If we are trying to use any distance metrics with this scheme we are saying that jazz is closer to rock than it is to soul (the distance between jazz and rock is 1 and the distance between

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 $\frac{1}{2}$ 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):

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.

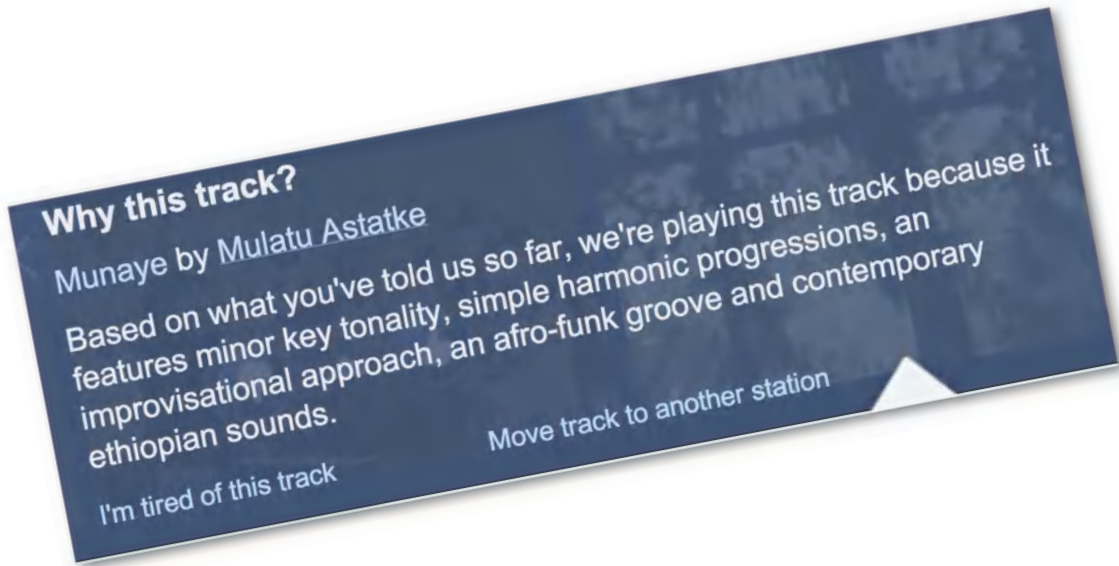
NOTE:

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.

Argghhh.

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.

Shhh. I'm normalizing



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!

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


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

$$\begin{aligned}
 & \begin{array}{ccccccc}
 \text{Yun's salary} & & \text{Allie's salary} & & \text{Daniela's salary} & & \text{etc.} \\
 \swarrow & & \swarrow & & \swarrow & & \\
 \sqrt{\frac{(75,000 - 72,125)^2 + (55,000 - 72,125)^2 + (45,000 - 72,125)^2 + \dots}{8}}
 \end{array} \\
 & = \sqrt{\frac{8,265,625 + 293,265,625 + 735,765,625 + \dots}{8}} = \sqrt{602,395,375} \\
 & = 24,543.01
 \end{aligned}$$

Again, the standard score is

(each value) - (mean)

(standard deviation)

So the Standard Score for Yun's salary is

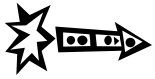
$$\frac{75000 - 72125}{24543.01} = \frac{2875}{24543.01} = 0.117$$



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

$$(100 * \$10 + 6,000,000 / (40 * 52)) / 101$$
$$= (1000 + 2885) / 101 = \$38/\text{hr}.$$

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:

(each value) - (median)

(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

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

The absolute standard deviation is

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

$$\begin{aligned} \text{asd} &= \frac{1}{8} (|43,000 - 69,500| + |45,000 - 69,500| + |55,000 - 69,500| + \dots) \\ &= \frac{1}{8} (26,500 + 24,500 + 14,500 + 500 + \dots) \\ &= \frac{1}{8} (153,000) = 19,125 \end{aligned}$$

Now let us compute the Modified Standard Score for Yun.

Modified Standard Score: $\frac{(\text{each value}) - (\text{median})}{(\text{absolute standard deviation})}$
--

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



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.

$$\begin{aligned} asd &= \frac{1}{5} (|3 - 12| + |7 - 12| + |12 - 12| + |15 - 12| + |21 - 12|) \\ &= \frac{1}{5} (9 + 5 + 0 + 3 + 9) = \frac{1}{5} (26) = 5.2 \end{aligned}$$

Step 3. Computing the Modified Standard Scores.

$$\text{Power / Marcus Miller: } (21 - 12) / 5.2 = 9 / 5.2 = 1.7307692$$

$$\text{I Breathe In, I Breathe Out / Chris Cagle: } (15 - 12) / 5.2 = 3 / 5.2 = 0.5769231$$

$$\text{Blessed / Jill Scott: } (12 - 12) / 5.2 = 0$$

$$\text{Europa / Santana: } (3 - 12) / 5.2 = -9 / 5.2 = -1.7307692$$

$$\text{Santa Fe / Beirut: } (7 - 12) / 5.2 = -5 / 5.2 = -0.961538$$

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):

Bill = {0, 0, 0, 1, 1, 1, 1, 0, 1, 0 ... }

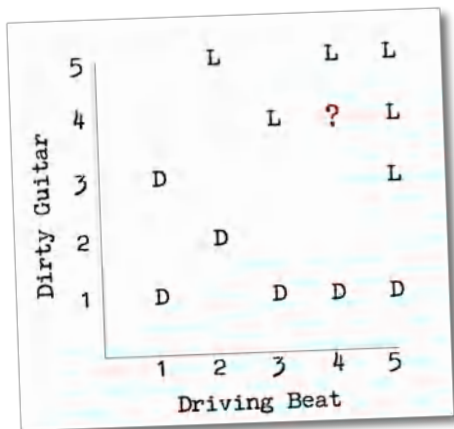
Obviously there is no need to normalize this data. What about the Pandora case: all variables lie on a scale from 1 to 5 inclusive. Should we normalize or not? It probably wouldn't hurt the accuracy of the algorithm if we normalized, but keep in mind that there is a computational cost involved with normalizing. 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).

	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

Earlier in this chapter we developed a Python representation of this data:

```
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},
```

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:

```
#
# 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 gives to songs. Because each user doesn't rate all songs (sparse data) I will go with the dictionary of dictionaries approach:

```
users = {"Angelica": {"Dr Dog/Fate": "L", "Phoenix/Lisztomania": "L",
                    "Heartless Bastards/Out at Sea": "D",
                    "Todd Snider/Don't Tempt Me": "D",
                    "The Black Keys/Magic Potion": "D",
                    "Glee Cast/Jessie's Girl": "L",
                    "La Roux/Bulletproof": "D",
                    "Mike Posner": "D",
                    "Black Eyed Peas/Rock That Body": "D",
                    "Lady Gaga/Alejandro": "L"},
        "Bill": {"Dr Dog/Fate": "L", "Phoenix/Lisztomania": "L",
                "Heartless Bastards/Out at Sea": "L",
                "Todd Snider/Don't Tempt Me": "D",
                "The Black Keys/Magic Potion": "L",
                "Glee Cast/Jessie's Girl": "D",
                "La Roux/Bulletproof": "D", "Mike Posner": "D",
                "Black Eyed Peas/Rock That Body": "D",
                "Lady Gaga/Alejandro": "D"} }
```

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.

```
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])
"└"
```

We are predicting she will like it! Why are we predicting that?


```
computeNearestNeighbor('Angelica', 'Cagle', [1, 5, 2.5, 1, 1, 5, 1])  
  
[(4.5, 'Lady Gaga/Alejandro'), (6.0, "Glee Cast/Jessie's Girl"), (7.5,  
"Todd Snider/Don't Tempt Me"), (8.0, 'Mike Posner'), (9.5, 'Heartless  
Bastards/Out at Sea'), (10.5, 'Black Eyed Peas/Rock That Body'), (10.5,  
'Dr Dog/Fate'), (10.5, 'La Roux/Bulletproof'), (10.5, 'Phoenix/  
Lisztomania'), (14.0, 'The Black Keys/Magic Potion')]
```

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*.

I like Phoenix, Lady Gaga and Dr. Dog. I don't like The Black Keys and Mike Posner!

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.

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.

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.

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.

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.

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.

Name	Sport	Age	Height	Weight
Asuka Teramoto	Gymnastics	16	54	66
Brittainey Raven	Basketball	22	72	162
Chen Nan	Basketball	30	78	204
Gabby Douglas	Gymnastics	16	49	90
Helalia Johannes	Track	32	65	99
Irina Miketenko	Track	40	63	106
Jennifer Lacy	Basketball	27	75	175
Kara Goucher	Track	34	67	123
Linlin Deng	Gymnastics	16	54	68
Nakia Sanford	Basketball	34	76	200
Nikki Blue	Basketball	26	68	163
Qiushuang Huang	Gymnastics	20	61	95
Rebecca Tunney	Gymnastics	16	58	77
Rene Kalmer	Track	32	70	108
Shanna Crossley	Basketball	26	70	155
Shavonte Zellous	Basketball	24	70	155
Tatyana Petrova	Track	29	63	108
Tiki Gelana	Track	25	65	106
Valeria Straneo	Track	36	66	97
Viktoria Komova	Gymnastics	17	61	76

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?





brain calisthenics - cont'd

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...





brain calisthenics - cont'd



We can use the Modified Standard Score!!!

(each value) - (median)

(absolute standard deviation)

Test Data.

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

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

Let's build a classifier!

Python Coding

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 data files and the Python code are on the book's website, guidetodatamining.com.

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.



Hey, you look what... maybe five foot eleven and 150? I bet your name is Clara Coleman.

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']), ...)
```



brain calisthenics - answer

a dictionary of the form:

```
{'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:

```
[['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:

```
(('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('\t')
    self.data = []
    for line in lines[1:]:
        fields = line.strip().split('\t')
        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))
```



code it

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:



```
>>> 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?

AssertionError?
See next page

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:

```
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:

```
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:

```
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:

```
[('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:

```
# 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.



code it

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:

```
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

$$mss = \frac{x_i - \tilde{x}}{asd} = \frac{70 - 65.5}{5.95} = \frac{4.5}{5.95} = 0.7563$$

and the second:

$$mss = \frac{x_i - \tilde{x}}{asd} = \frac{140 - 107}{33.65} = \frac{33}{33.65} = 0.98068$$

The python method that does this is:

```
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

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

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

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



code it

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.

```
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:

```
>>> 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.



Sir Fisher was a remarkable person. He revolutionized statistics and Richard Dawkins called him "the greatest biologist since Darwin."



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):



Sepal length	Sepal width	Petal Length	Petal Width	Species
5.1	3.5	1.4	0.2	I.setosa
4.9	3.0	1.4	0.2	I setosa

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?

```
>>> 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

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

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%.

```
>>> 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!

odds and ends



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



 You say normalize and I
 say standardize  You say
 tomato and I say tomato 



code it

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:

data set	classifier built		
	using no normalization	using the formula on previous page	using Modified Standard Score
Athletes	80.00%	?	80.00%
Iris	100.00%	?	93.33%
MPG	32.00%	?	56.00%



my results

Here are my results:

data set	classifier built		
	using no normalization	using the Formula on previous page	using Modified Standard Score
Athletes	80.00%	60.00%	80.00%
Iris	100.00%	83.33%	93.33%
MPG	32.00%	36.00%	56.00%

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.

Chapter 5: Further Explorations in Classification

Evaluating algorithms and kNN

Let us return to the athlete example from the previous chapter.



In that example we built a classifier which took the height and weight of an athlete as input and classified that input by sport—gymnastics, track, or basketball.

So Marissa Coleman, pictured on the left, is 6 foot 1 and weighs 160 pounds. Our classifier correctly predicts she plays basketball:

```
>>> cl = Classifier('athletesTrainingSet.txt')
```

```
>>> cl.classify([73, 160])
```

```
'Basketball'
```

and predicts that someone 4 foot 9 and 90 pounds is likely to be a gymnast:

```
>>> cl.classify([59, 90])
```

```
'Gymnastics'
```

Once we build a classifier, we might be interested in answering some questions about it such as:



How can we answer these questions?

Training set and test set.

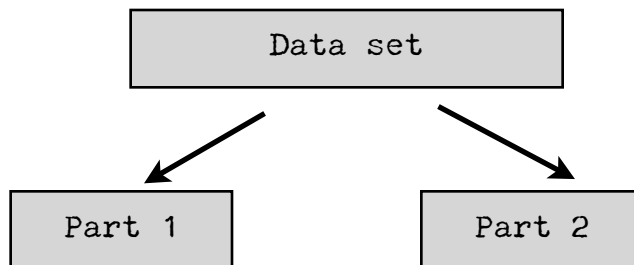
At the end of the previous chapter we worked with three different datasets: the women athlete dataset, the iris dataset, and the auto miles-per-gallon one. We divided each of these datasets in turn into two subsets. One subset we used to construct the classifier. This data set is called the *training set*. The other set was used to evaluate the classifier. That data is called the *test set*. *Training set* and *test set* are common terms in data mining.

People in data mining never test with the data they used to train the system.

You can see why we don't use the training data for testing if we consider the nearest neighbor algorithm. If Marissa Coleman the basketball player from the above example, was in our training data, she at 6 foot 1 and 160 pounds would be the nearest neighbor of herself. So when evaluating a nearest neighbor algorithm, if our test set is a subset of our training data we would always be close to 100% accurate. More generally, in evaluating any data mining algorithm, if our test set is a subset of our training data the results will be optimistic and often overly optimistic. So that doesn't seem like a great idea.

How about the idea we used in the last chapter? We divide our data into two parts. The larger part we use for training and the smaller part we use for evaluation. As it turns out that has its problems too. We could be extremely unlucky in how we divide up our data. For example, all the basketball players in our test set might be short (like Debbie Black who is only 5 foot 3 and weighs 124 pounds) and get classified as marathoners. And all the track people in the test set might be short and lightweight for that sport like Tatyana Petrova (5 foot 3 and 108 pounds) and get classified as gymnasts. With a test set like this, our accuracy will be poor. On the other hand, we could be very lucky in our selection of a test set. Every person in the test set is the prototypical height and weight for their respective sports and our accuracy is near 100%. In either case, the accuracy based on a single test set may not reflect the true accuracy when our classifier is used with new data.

A solution to this problem might be to repeat the process a number of times and average the results. For example, we might divide the data in half. Let's call the parts Part 1 and Part 2:



We can use the data in Part 1 to train our classifier and the data in Part 2 to test it. Then we will repeat the process, this time training with Part 2 and testing with Part 1. Finally we average the results. One problem with this though, is that we are only using 1/2 the data for training during each iteration. But we can fix this by increasing the number of parts. For example, we can have three parts and for each iteration we will train on 2/3 of the data and test on 1/3. So it might look like this

iteration 1	train with parts 1 and 2	test with part 3
iteration 2	train with parts 1 and 3	test with part 2
iteration 3	train with parts 2 and 3	test with part 1

Average the results.

In data mining, the most common number of parts is 10, and this method is called ...

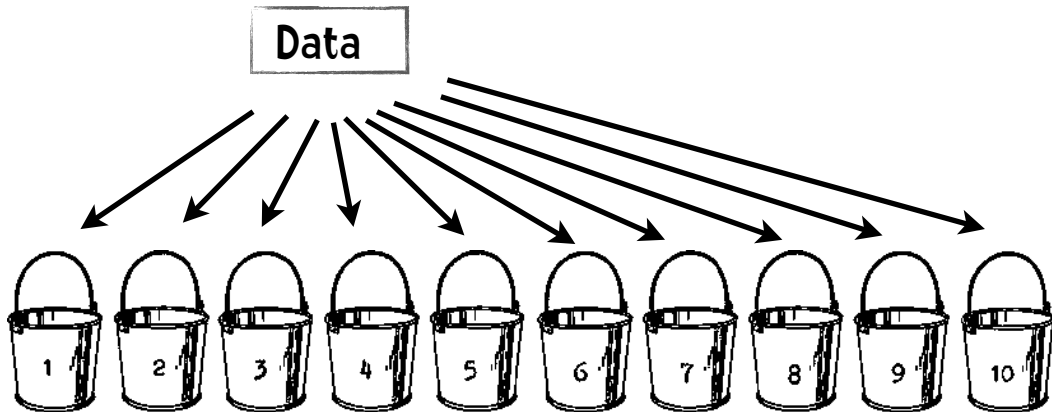
10-fold Cross Validation

With this method we have one data set which we divide randomly into 10 parts. We use 9 of those parts for training and reserve one tenth for testing. We repeat this procedure 10 times each time reserving a different tenth for testing.

Let's look at an example. Suppose I want to build a classifier that just answers yes or no to the question *Is this person a professional basketball player?* My data consists of information about 500 basketball players and 500 non-basketball players.

ten-fold cross validation example:

Step 1, we equally divide the data into 10 buckets:



So we will put 50 basketball players in each bucket and 50 non-players. Each bucket holds information on 100 individuals.

Step 2, we iterate through the following steps ten times:

- 2.1. During each iteration hold back one of the buckets. For iteration 1, we will hold back bucket 1, iteration 2, bucket 2, and so on.
- 2.2. We will train the classifier with data from the other buckets. (during the first iteration we will train with the data in buckets 2 through 10).
- 2.3. We will test the classifier we just built using data from the bucket we held back and save the results. In our case these results might be:

35 of the basketball players were classified correctly
29 of the non basketball players were classified correctly

Step 3, we sum up the results.

Often we will put the final results in a table that looks like this:

	classified as a basketball player	classified as not a basketball player
really a basketball player	372	128
really not a basketball player	220	280

So of the 500 basketball players 372 of them were classified correctly. One thing we could do is add things up and say that of the 1,000 people we classified 652 (372 + 280) of them correctly. So our accuracy is 65.2%. The measures we obtain using ten-fold cross-validation are more likely to be truly representative of the classifiers performance compared with two-fold, or three-fold cross-validation. This is so, because each time we train the classifier we are using 90% of our data compared with using only 50% for two-fold cross-validation.

Hmmm. I have an idea. If 10-fold cross validation is good because we are training on 90% of the data, how about using n-fold cross validation where n is the number of entries in our data set?

For example, if we have 1,000 entries, we will train our classifier on 999 of them and test on 1, and repeat this process 1,000 times. Using the largest possible amount of our data for training should result in a highly accurate classifier.

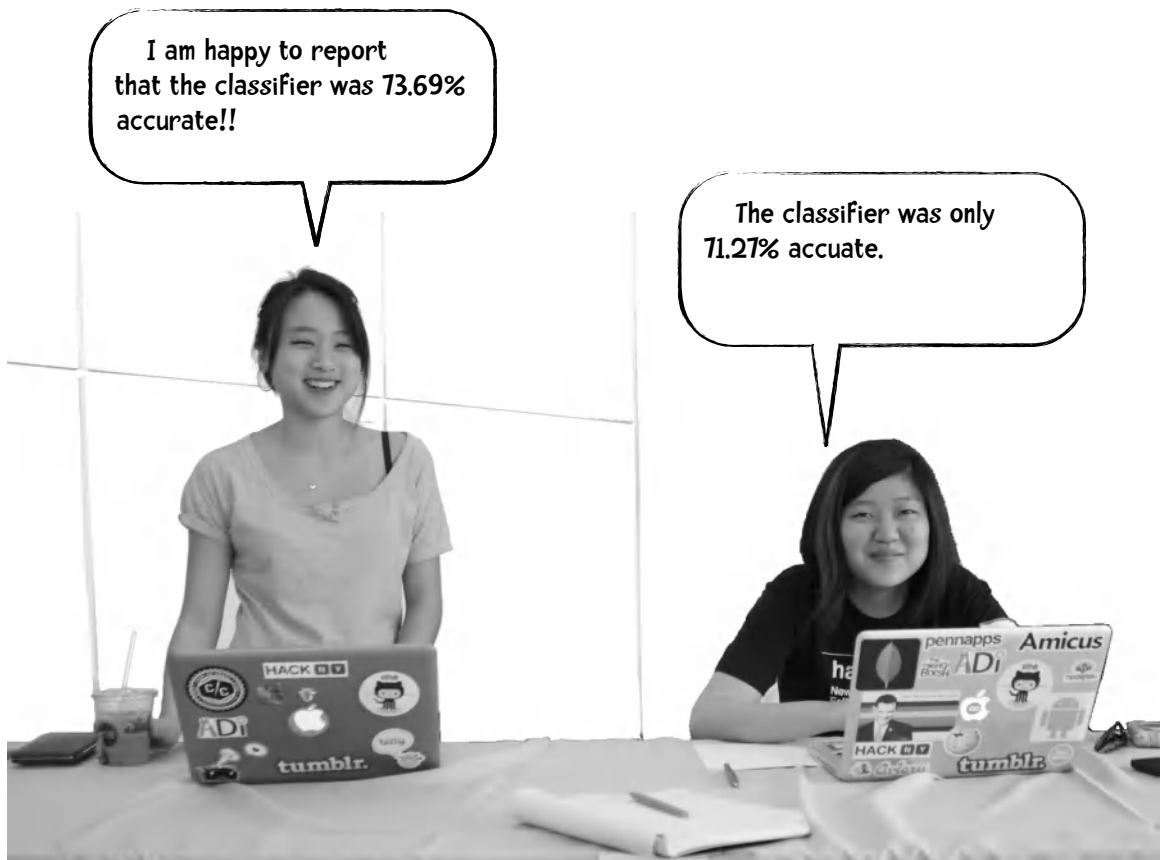


Leave-One-Out

In the machine learning literature, n -fold cross validation (where n is the number of samples in our data set) is called leave-one-out. We already mentioned one benefit of leave-one-out—at every iteration we are using the largest possible amount of our data for training. The other benefit is that it is deterministic.

What do we mean by ‘deterministic’?

Suppose Lucy spends an intense 80 hour week creating and coding a new classifier. It is Friday and she is exhausted so she asks two of her colleagues (Emily and Li) to evaluate the classifier over the weekend. She gives each of them the classifier and the same dataset and asks them to use 10-fold cross validation. On Monday she asks for the results ...



Hmm. They did not get the same results. Did Emily or Li make a mistake? Not necessarily. In 10-fold cross validation we place the data randomly into 10 buckets. Since there is this random element, it is likely that Emily and Li did not divide the data into buckets in exactly the same way. In fact, it is highly unlikely that they did. So when they train the classifier, they are not using exactly the same data and when they test this classifier they are using different test sets. So it is quite logical that they would get different results. This result has nothing to do with the fact that two different people were performing the evaluation. If Lucy herself ran 10-fold cross validation twice, she too would get slightly different results. The reason we get different results is that there is a random component to placing the data into buckets. So 10-fold cross validation is called non-deterministic because when we run the test again we are not guaranteed to get the same result. In contrast, the leave-one-out method is deterministic. Every time we use leave-one-out on the same classifier and the same data we will get the same result. That is a good thing!

The disadvantages of leave-one-out

The main disadvantage of leave-one-out is the computational expense of the method. Consider a modest-sized dataset of 1,000 instances and that it takes one minute to train a classifier. For 10-fold cross validation we will spend 10 minutes in training. In leave-one-out we will spend 16 hours in training. If our dataset contains a million entries the total time spent in training would nearly be two years. Eeeks!



**I'll get that report
to you in two years!**

The other disadvantage of leave-one-out is related to stratification.

Stratification.

Let us return to an example from the previous chapter—building a classifier that predicts what sport a woman plays (basketball, gymnastics, or track). When training the classifier we want the training data to be representative and contain data from all three classes. Suppose we assign data to the training set in a completely random way. It is possible that no basketball players would be included in the training set and because of this, the resulting classifier would not be very good at classifying basketball players. Or consider creating a data set of 100 athletes. First we go to the Women’s NBA website and write down the info on 33 basketball players; next we go to Wikipedia and get 33 women who competed in gymnastics, at the 2012 Olympics and write that down; finally, we go again to Wikipedia to get information on women who competed in track at the Olympics and record data for 34 people. So our dataset looks like this:

comment	class	num	num
Anika Teramoto	Gymnastics	54	66
Brittney Raven	Basketball	72	162
Chen Nian	Basketball	78	204
Gabby Douglas	Gymnastics	49	90
Helala Johannes	Track	65	99
Irina Mikitenko	Track	63	106
Jennifer Lacy	Basketball	75	175
Kara Goucher	Track	67	123
Lidia Deng	Gymnastics	54	68
Nakia Sanford	Basketball	76	200
Nikki Blue	Basketball	68	163
Qushuang Huang	Gymnastics	61	95
Rebecca Tunney	Gymnastics	58	77
Rene Kalmr	Track	70	108
Shanna Croudey	Basketball	70	155
Shavonte Zellous	Basketball	70	155
Tatjana Petrova	Track	63	108
Tiki Gelana	Track	65	106
Valeria Straneo	Track	66	97
Viktoria Komova	Gymnastics	61	76
comment	class	num	num
Anika Teramoto	Gymnastics	54	66
Brittney Raven	Basketball	72	162
Chen Nian	Basketball	78	204
Gabby Douglas	Gymnastics	49	90
Helala Johannes	Track	65	99
Irina Mikitenko	Track	63	106
Jennifer Lacy	Basketball	75	175
Kara Goucher	Track	67	123
Lidia Deng	Gymnastics	54	68
Nakia Sanford	Basketball	76	200
Nikki Blue	Basketball	68	163
Qushuang Huang	Gymnastics	61	95
Rebecca Tunney	Gymnastics	58	77
Rene Kalmr	Track	70	108
Shanna Croudey	Basketball	70	155
Shavonte Zellous	Basketball	70	155
Tatjana Petrova	Track	63	108
Tiki Gelana	Track	65	106
Valeria Straneo	Track	66	97
Viktoria Komova	Gymnastics	61	76
comment	class	num	num
Anika Teramoto	Gymnastics	54	66
Brittney Raven	Basketball	72	162
Chen Nian	Basketball	78	204
Gabby Douglas	Gymnastics	49	90
Helala Johannes	Track	65	99
Irina Mikitenko	Track	63	106
Jennifer Lacy	Basketball	75	175
Kara Goucher	Track	67	123
Lidia Deng	Gymnastics	54	68
Nakia Sanford	Basketball	76	200
Nikki Blue	Basketball	68	163
Qushuang Huang	Gymnastics	61	95
Rebecca Tunney	Gymnastics	58	77
Rene Kalmr	Track	70	108
Shanna Croudey	Basketball	70	155
Shavonte Zellous	Basketball	70	155
Tatjana Petrova	Track	63	108
Tiki Gelana	Track	65	106
Valeria Straneo	Track	66	97
Viktoria Komova	Gymnastics	61	76

33 women basketball players

33 women gymnasts

34 women marathoners

Let's say we are doing 10-fold cross validation. We start at the beginning of the list and put every ten people in a different bucket. In this case we have 10 basketball players in both the first and second buckets. The third bucket has both basketball players and gymnasts. The fourth and fifth buckets solely contain gymnasts and so on. None of our buckets are representative of the dataset as a whole and you would be correct in thinking this would skew our results. The preferred method of assigning instances to buckets is to make sure that the classes (basketball players, gymnasts, marathoners) are representing in the same proportions as they are in the complete dataset. Since one-third of the complete dataset consists of basketball players, one-third of the entries in each bucket should also be basketball players. And one-third the entries should be gymnasts and one-third marathoners. This is called **stratification** and this is a good thing. The problem with the leave-one-out evaluation method is that necessarily all the test sets are non-stratified since they contain only one instance. In sum, while leave-one-out may be appropriate for very small datasets, 10-fold cross validation is by far the most popular choice.

Confusion Matrices



So far, we have been evaluating our classifier by computing the percent accuracy. That is, **number of test cases correctly classified**

Total number of test cases

sometimes we may want a more detailed picture of the performance of our classification algorithm and one such detailed visualization is a table called the *confusion matrix*. The rows of the confusion matrix represent the actual class of the test cases, the columns represent what our classifier predicted.

The name *confusion matrix* comes from the observation that it is easy for us to see where our algorithm gets confused. Let's look at an example using our women athlete domain. Suppose we have a dataset that consists of attributes for 100 women gymnasts, 100 players in the Women's National Basketball Association, and 100 women marathoners. We evaluate the classifier using 10-fold cross-validation. In 10-fold cross-validation we use each instance of our dataset exactly once for testing. The results of this test might be the following confusion matrix:

	gymnast	basketball player	marathoner
gymnast	83	0	17
basketball player	0	92	8
marathoner	9	16	75

Again, the real class of each instance is represented by the rows; the class predicted by our classifier is represented by the columns. So, for example, 83 instances of gymnasts were classified correctly as gymnasts but 17 were misclassified as marathoners. 92 basketball players were classified correctly as basketball players but 8 were misclassified as marathoners. 75 marathoners were classified correctly but 9 were misclassified as gymnasts and 16 misclassified as basketball players.

The diagonal of the confusion matrix represents instances that were classified correctly.

	gymnast	basketball player	marathoner
gymnast	83	0	17
basketball player	0	92	8
marathoner	9	16	85

In this case the accuracy of the algorithm is:

$$\frac{83 + 92 + 75}{300} = \frac{250}{300} = 83.33\%$$

It is easy to inspect the matrix to get an idea of what type of errors our classifier is making. In this example, it seems our algorithm is pretty good at distinguishing between gymnasts and basketball players. Sometimes gymnasts and basketball players get misclassified as marathoners and marathoners occasionally get misclassified as gymnasts or basketball players.



**Confusion matrices
are not that
confusing!**

A programming example

Let us go back to a dataset we used in the last chapter, the Auto Miles Per Gallon data set from Carnegie Mellon University. The format of the data looked like:

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

I am trying to predict the miles per gallon of a vehicle based on number of cylinders, displacement (cubic inches), horsepower, weight, and acceleration. I put all 392 instances in a file named mpgData.txt and wrote the following short Python program that divided the data into ten buckets using a stratified method. (Both the data file and Python code are available on the website guidetodatamining.com.)

```

import random
def buckets(filename, bucketName, separator, classColumn):
    """the original data is in the file named filename
    bucketName is the prefix for all the bucket names
    separator is the character that divides the columns
    (for ex., a tab or comma) and classColumn is the column
    that indicates the class"""

    # put the data in 10 buckets
    numberOfBuckets = 10
    data = {}
    # first read in the data and divide by category
    with open(filename) as f:
        lines = f.readlines()
    for line in lines:
        if separator != '\t':
            line = line.replace(separator, '\t')
        # first get the category
        category = line.split()[classColumn]
        data.setdefault(category, [])
        data[category].append(line)
    # initialize the buckets
    buckets = []
    for i in range(numberOfBuckets):
        buckets.append([])
    # now for each category put the data into the buckets
    for k in data.keys():
        #randomize order of instances for each class
        random.shuffle(data[k])
        bNum = 0
        # divide into buckets
        for item in data[k]:
            buckets[bNum].append(item)
            bNum = (bNum + 1) % numberOfBuckets
    # write to file
    for bNum in range(numberOfBuckets):
        f = open("%s-%02i" % (bucketName, bNum + 1), 'w')
        for item in buckets[bNum]:
            f.write(item)
        f.close()

buckets("mpgData.txt", 'mpgData', '\t', 0)

```


Executing this code will produce ten files labelled *mpgData01*, *mpgData02*, etc.



code it

Can you revise the nearest neighbor code from the last chapter so the function test performs 10-fold cross validation on the 10 data files we just created (you can download them at guidetodatamining.com)?

Your program should output a confusion matrix that looks something like:

		predicted MPG							
		10	15	20	25	30	35	40	45
ac- tual MPG	10	3	10	0	0	0	0	0	0
	15	3	68	14	1	0	0	0	0
	20	0	14	66	9	5	1	1	0
	25	0	1	14	35	21	6	1	1
	30	0	1	3	17	21	14	5	2
	35	0	0	2	8	9	14	4	1
	40	0	0	1	0	5	5	0	0
	45	0	0	0	2	1	1	0	2

53.316% accurate
total of 392 instances



code it - one solution

One solution involves only

- Changing the initializer method to read in data from 9 buckets.
- Adding a new method to test from data in one bucket
- Adding a new procedure that performs 10-fold cross-validation

Let us look at these in turn.

initializer method `__init__`

The signature of the init method looks like:

```
def __init__(self, bucketPrefix, testBucketNumber, dataFormat):
```

The filenames of the buckets will be something like `mpgData-01`, `mpgData-02`, etc. In this case, `bucketPrefix` will be “`mpgData`”. `testBucketNumber` is the bucket containing the test data. If `testBucketNumber` is 3, the classifier will be trained on buckets 1, 2, 4, 5, 6, 7, 8, 9, and 10. `dataFormat` is a string specifying how to interpret the columns in the data. For example,

```
"class    num    num    num    num    num    comment"
```

specifies that the first column represents the class of the instance. The next 5 columns represent numerical attributes of the instance and the final column should be interpreted as a comment.

The complete, new initializer method is as follows:

```

import copy

class Classifier:
    def __init__(self, bucketPrefix, testBucketNumber, dataFormat):

        """ a classifier will be built from files with the bucketPrefix
        excluding the file with testBucketNumber. dataFormat is a
        string that describes how to interpret each line of the data
        files. For example, for the mpg data the format is:
        "class num num num num num comment"
        """
        self.medianAndDeviation = []

        # reading the data in from the file
        self.format = dataFormat.strip().split('\t')
        self.data = []
        # for each of the buckets numbered 1 through 10:
        for i in range(1, 11):
            # if it is not the bucket we should ignore, read the data
            if i != testBucketNumber:
                filename = "%s-%02i" % (bucketPrefix, i)
                f = open(filename)
                lines = f.readlines()
                f.close()
                for line in lines:
                    fields = line.strip().split('\t')
                    ignore = []
                    vector = []
                    for i in range(len(fields)):
                        if self.format[i] == 'num':
                            vector.append(float(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))
        self.rawData = copy.deepcopy(self.data)
        # 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)

```

testBucket method

Next, we write a new method that will test the data in one bucket:

```
def testBucket(self, bucketPrefix, bucketNumber):
    """Evaluate the classifier with data from the file
    bucketPrefix-bucketNumber"""

    filename = "%s-%02i" % (bucketPrefix, bucketNumber)
    f = open(filename)
    lines = f.readlines()
    totals = {}
    f.close()
    for line in lines:
        data = line.strip().split('\t')
        vector = []
        classInColumn = -1
        for i in range(len(self.format)):
            if self.format[i] == 'num':
                vector.append(float(data[i]))
            elif self.format[i] == 'class':
                classInColumn = i
        theRealClass = data[classInColumn]
        classifiedAs = self.classify(vector)
        totals.setdefault(theRealClass, {})
        totals[theRealClass].setdefault(classifiedAs, 0)
        totals[theRealClass][classifiedAs] += 1
    return totals
```

This takes as input a bucketPrefix and a bucketNumber. If the prefix is "mpgData " and the number is 3, the test data will be read from the file mpgData-03. testBucket will return a dictionary in the following format:

```
{'35': {'35': 1, '20': 1, '30': 1},
 '40': {'30': 1},
 '30': {'35': 3, '30': 1, '45': 1, '25': 1},
 '15': {'20': 3, '15': 4, '10': 1},
 '10': {'15': 1},
 '20': {'15': 2, '20': 4, '30': 2, '25': 1},
 '25': {'30': 5, '25': 3}}
```

The key of this dictionary represents the true class of the instances. For example, the first line represents results for instances whose true classification is 35 mpg. The value for each key is another dictionary that represents how our classifier classified the instances. For example, the line

```
'15': {'20': 3, '15': 4, '10': 1},
```

represents a test where 3 of the instances that were really 15mpg were misclassified as 20mpg, 4 were classified correctly as 15mpg, and 1 was classified incorrectly as 10mpg.

procedure to perform 10-fold cross-validation.

Finally, we need to write a procedure that will perform 10-fold cross-validation. That is, it builds 10 classifiers. Each classifier is trained on 9 of the buckets and tested on data from the remaining bucket.

```
def tenfold(bucketPrefix, dataFormat):
    results = {}
    for i in range(1, 11):
        c = Classifier(bucketPrefix, i, dataFormat)
        t = c.testBucket(bucketPrefix, i)
        for (key, value) in t.items():
            results.setdefault(key, {})
            for (ckey, cvalue) in value.items():
                results[key].setdefault(ckey, 0)
                results[key][ckey] += cvalue

    # now print results
    categories = list(results.keys())
    categories.sort()
    print( "\n      Classified as: ")
    header = "      "
    subheader = "      +"
    for category in categories:
        header += category + "      "
        subheader += "----+"
    print (header)
    print (subheader)
    total = 0.0
    correct = 0.0
```

```

for category in categories:
    row = category + "    |"
    for c2 in categories:
        if c2 in results[category]:
            count = results[category][c2]
        else:
            count = 0
        row += " %2i |" % count
        total += count
        if c2 == category:
            correct += count
    print(row)
print(subheader)
print("\n%5.3f percent correct" %((correct * 100) / total))
print("total of %i instances" % total)

```

```
tenfold("mpgData", "class    num    num    num    num    num    comment")
```

Running the program yields the following results:

Classified as:		10	15	20	25	30	35	40	45
10	5	8	0	0	0	0	0	0	0
15	8	63	14	1	0	0	0	0	0
20	0	14	67	8	5	1	1	0	0
25	0	1	13	35	22	6	1	1	1
30	0	1	3	17	21	14	5	2	2
35	0	0	2	7	10	13	5	1	1
40	0	0	1	0	5	5	0	0	0
45	0	0	0	2	1	1	0	2	2

```
52.551 percent correct
total of 392 instances
```

Kappa Statistic!

At the start of this chapter we mentioned some of the questions we might be interested in answering about a classifier including *How good is this classifier*. We also have been refining our evaluation methods and looked at 10-fold cross-validation and confusion matrices. In the example on the previous pages we determined that our classifier for predicted miles per gallon of selected car models was 53.316% accurate. But does 53.316% mean our classifier is good or not so good? To answer that question we are going to look at one more statistics, the Kappa Statistic.



The Kappa Statistic compares the performance of a classifier to that of a classifier that makes predictions based solely on chance. To show you how this works I will start with a simpler example than the mpg one and again return to the women athlete domain. Here are the results of a classifier in that domain:

	gymnast	basketball player	marathoner	TOTALS
gymnast	35	5	20	60
basketball player	0	88	12	100
marathoner	5	7	28	40
TOTALS	40	100	60	200

I also show the totals for the rows and columns. To determine the accuracy we sum the numbers on the diagonal ($35 + 88 + 28 = 151$) and divide by the total number of instances (200) to get $151 / 200 = .755$

Now I am going to generate another confusion matrix that will represent the results of a random classifier (a classifier that makes random predictions). First, we are going to make a copy of the above table only containing the totals:

	gymnast	basketball player	marathoner	TOTALS
gymnast				60
basketball player				100
marathoner				40
TOTALS	40	100	60	200

Looking at the bottom row, we see that 50% of the time (100 instances out of 200) our classifier classifies an instance as “Basketball Player”, 20% of the time (40 instances out of 200) it classifies an instance as “gymnast” and 30% as “marathoner.”

Classifier:

gymnast: 20%
basketball player: 50%
marathoner: 30%

We are going to use these percentages to fill in the rest of the table. There were 60 total real gymnasts. Our random classifier will classify 20% of those as gymnasts. 20% of 60 is 12 so we put a 12 in the table. It will classify 50% as basketball players (or 30 of them) and 30% as marathoners.

	gymnast	basketball player	marathoner	TOTALS
gymnast	12	30	18	60
basketball player				100
marathoner				40
TOTALS	40	100	60	200

And we will continue in this way. There are 100 real basketball players. The random classifier will classify 20% of them (or 20) as gymnasts, 50% as basketball players and 30% as marathoners. And so on:

	gymnast	basketball player	marathoner	TOTALS
gymnast	12	30	18	60
basketball player	20	50	30	100
marathoner	8	20	12	40
TOTALS	40	100	60	200

To determine the accuracy of the random method we sum the numbers on the diagonal and divide by the total number of instances:

$$P(r) = \frac{12 + 50 + 12}{200} = \frac{74}{200} = .37$$

The Kappa Statistic will tell us how much better the real classifier is compared to this random one. The formula is

$$\kappa = \frac{P(c) - P(r)}{1 - P(r)}$$

where $P(c)$ is the accuracy of the real classifier and $P(r)$ is the accuracy of the random one. In this case the accuracy of the real classifier was .755 and that of the random one was .37 so

$$\kappa = \frac{.755 - .37}{1 - .37} = \frac{.385}{.63} = .61$$

How do we interpret that .61? Does that mean our classifier is poor, good, or great? Here is a chart that will help us interpret that number:

A commonly cited* scale on how to interpret Kappa	
< 0:	less than chance performance
0.01-0.20	slightly good
0.21-0.40	fair performance
0.41-0.60	moderate performance
0.61-0.80	substantially good performance
0.81-1.00	near perfect performance

* Landis, JR, Koch, GG. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33:159-74



sharpen your pencil

Suppose we developed a somewhat silly classifier that predicts the major of current university students based on how well they liked 10 movies. We have a data set of 600 students consisting of computer science (cs) majors, education majors (ed), English majors (eng) and psychology majors (psych). The confusion matrix is shown below. Can you compute the Kappa Statistic and interpret what that statistic means?

	predicted major				Total
	cs	ed	eng	psych	
cs	50	8	15	7	
ed	0	75	12	33	
eng	5	12	123	30	
psych	5	25	30	170	

accuracy = 0.697



solution

How good is our classifier? Can you compute the Kappa Statistic and interpret what that statistic means?

First, we sum all the columns:

	cs	ed	eng	psych	TOTAL
SUM	60	120	180	240	600
%	10%	20%	30%	40%	100%

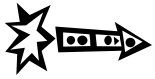
Next, we construct the confusion matrix for the random classifier

predicted major

	cs	ed	eng	psych	Total
cs	8	16	24	32	80
ed	12	24	36	48	120
eng	17	34	51	68	170
psych	23	46	69	92	230
Total	60	120	180	240	600

The accuracy of this random classifier is:

$$(8 + 24 + 51 + 92) / 600 = (175 / 600) = 0.292$$



solution continued

So the accuracy of our classifier $P(c)$ is 0.697
and that of the random classifier $P(r)$ is 0.292

The Kappa Statistic is

$$\kappa = \frac{P(c) - P(r)}{1 - P(r)}$$

$$\kappa = \frac{0.697 - 0.292}{1 - 0.292} = \frac{0.405}{0.708} = 0.572$$

This suggests our algorithm performs moderately well.



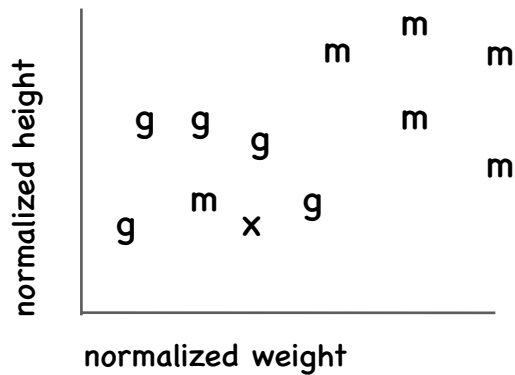
Improvements to the Nearest Neighbor Algorithm!

One trivial example of a classifier is the **Rote Classifier**, which just memorizes the entire training set and only classifies an instance if that instance exactly matches one in the training set. If we only evaluated classifiers on instances in the training data, the Rote Classifier would always be 100% accurate. In real life, the rote classifier is not a good choice because there will be instances we want to classify that are not in the training set. You can view the nearest neighbor algorithm we have been working with as an extension of the rote classifier. Instead of requiring exact matches we are looking at instances that are close matches. Pang-Ning Tan, Michael Steinbach, and Vipin Kumar in their data mining textbook¹ call this the *If it walks like a duck, quacks like a duck, and looks like a duck, then it's probably a duck* approach.



One problem with the nearest neighbor algorithm occurs when we have outliers. Let me explain what I mean by that. And let us return, yet again, to the women athlete domain; this time only looking at gymnasts and marathoners. Suppose we have a particularly short and lightweight marathoner. In diagram form, this data might be represented as on the next page, where m indicates 'marathoner' and g , 'gymnast'.

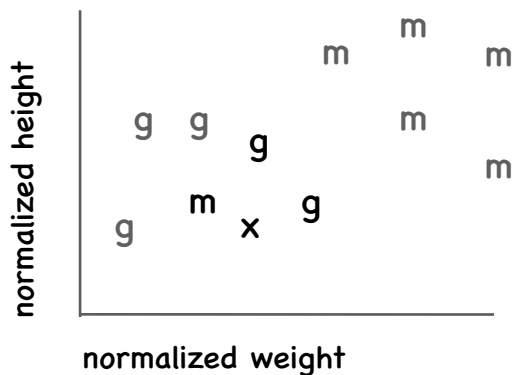
¹ Introduction to Data Mining. 2005. Addison-Wesley



We can see that short lightweight marathoner as the sole *m* in the group of *g*'s. Suppose *x* is an instance we would like to classify. Its nearest neighbor is that outlier *m*, so it would get classified as a marathoner. If we just eyeballed the diagram we would say that *x* is most likely a gymnast since it appears to be in the group of gymnasts.

kNN

One way to improve our current nearest neighbor approach is instead of looking at one nearest neighbor we look at a number of nearest neighbors—*k* nearest neighbors or *k*NN. Each neighbor will get a vote and the algorithm will predict that the instance will belong to the class with the highest number of votes. For example, suppose we are using three nearest neighbors ($k = 3$). In that case we have 2 votes for *gymnast* and one for *marathoner*, so we would predict *x* is a gymnast:





So when we are trying to predict a **discrete class** (marathoners, gymnasts, or basketball players, for example) we can use this voting method. The class with the most votes will be the one assigned to the instance. If there is a tie the predicted class will be selected randomly from the classes that are tied. When we are trying to predict a **numeric value** like how many stars a person will give the band *Funky Meters* we can apportion influence from the nearest neighbors to compute a distance-weighted value. Let me parse that out a bit more. Suppose we are trying to predict how well Ben will like *Funky Meters* and Ben's three closest neighbors are Sally, Tara, and Jade. Here are their distances from Ben and their ratings for *Funky Meters*.

User	Distance	Rating
Sally	5	4
Tara	10	5
Jade	15	5

So Sally was closest to Ben and she gave *Funky Meters* a 4. Because I want the rating of the closest person to be weighed more heavily in the final value than the other neighbors, the first step we will do is to convert the distance measure to make it so that the larger the number the closer that person is. We can do this by computing the inverse of the distance (that is, 1 over the distance). So the inverse of Sally's distance of 5 is

$$\frac{1}{5} = 0.2$$

User	Inverse Distance	Rating
Sally	0.2	4
Tara	0.1	5
Jade	0.067	5

Now I am going to divide each of those inverse distances by the sum of all the inverse distances. The sum of the inverse distances is $0.2 + 0.1 + 0.067 = 0.367$.

User	Influence	Rating
Sally	0.545	4
Tara	0.272	5
Jade	0.183	5

We should notice two things. First, that the sum of the influence values totals 1. The second thing to notice is that with the original distance numbers Sally was twice as close to Ben as Tara was, and that is preserved in the final numbers were Sally has twice the influence as

Tara does. Finally we are going to multiple each person's influence and rating and sum the results:

predicted Score for Ben

$$= 0.545 \times 4 + 0.272 \times 5 + 0.183 \times 5$$

$$= 2.18 + 1.36 + 0.915 = 4.455$$



sharpen your pencil

I am wondering how well Sofia will like the jazz pianist Hiromi. What is the predicted value given the following data using the k nearest neighbor algorithm with $k = 3$?

person	distance from Sofia	rating for Hiromi
Gabriela	4	3
Ethan	8	3
Jayden	10	5



sharpen your pencil - solution

the first thing to do is to compute the inverse (1 over the distance) of each distance:

Person	Inverse Distance	Rating
Gabriela	$1/4 = 0.25$	3
Ethan	$1/8 = 0.125$	3
Jayden	$1/10 = 0.1$	5

The sum of the inverse distances is 0.475. Next I am going to compute the influence of each person by dividing the inverse distance by the sum of each distance

Person	Influence	Rating
Gabriela	0.526	3
Ethan	0.263	3
Jayden	0.211	5

Finally, I multiply the influence by the rating and sum the results:

$$= (0.526 \times 3) + (0.263 \times 3) + (0.211 \times 5)$$

$$= 1.578 + 0.789 + 1.055 = 3.422$$

A new dataset and a challenge!

It is time to look at a new dataset, the Pima Indians Diabetes Data Set developed by the United States National Institute of Diabetes and Digestive and Kidney Diseases.



Astonishingly, over 30% of Pima people develop diabetes. In contrast, the diabetes rate in the United States is 8.3% and in China it is 4.2%.

Each instance in the dataset represents information about a Pima woman over the age of 21 and belonged to one of two classes: a person who developed diabetes within five years, or a person that did not. There are eight attributes:

attributes:

1. Number of times pregnant
2. Plasma glucose concentration
3. Diastolic blood pressure (mm Hg)
4. Triceps skin fold thickness (mm)
5. 2-Hour serum insulin (μ U/ml)
6. Body mass index (weight in kg/(height in m)²)
7. Diabetes pedigree function
8. Age (years)

Here is an example of the data (the last column represents the class—0=no diabetes; 1=diabetes):

2	99	52	15	94	24.6	0.637	21	0
3	83	58	31	18	34.3	0.336	25	0
5	139	80	35	160	31.6	0.361	25	1
3	170	64	37	225	34.5	0.356	30	1

So, for example, the first woman has had 2 children, has a plasma glucose concentration of 99, a diastolic blood pressure of 52 and so on.





code it - part 1

There are two files on our website. `pimaSmall.zip` is a zip file containing 100 instances of the data divided into 10 files (buckets). `pima.zip` is a zip file containing 393 instances. When I used the `pimaSmall` data with the nearest neighbor classifier we built in the previous chapter using 10-fold cross-validation I got these results:

```

Classified as:
  0    1
+----+----+
0 | 45 | 14 |
1 | 27 | 14 |
+----+----+

```

59.000 percent correct
total of 100 instances

Hint: The python function `heapq.nsmallest(n, list)` will return a list with the `n` smallest items.

Here is your task:

Download the classifier code from our website and implement the kNN algorithm. Let us change the initializer method of the class to add another argument, `k`:

```
def __init__(self, bucketPrefix, testBucketNumber, dataFormat, k):
```

The method signature should look like `def knn(self, itemVector):`. It should make use of `self.k` (remember to set that value in the init method) and return the class (in this Pima Cancer dataset case '0' or '1'). You should also modify the procedure `tenfold` to pass `k` to the initializer.



code it - answer

My modification to `__init__` was simply:

```
def __init__(self, bucketPrefix, testBucketNumber, dataFormat, k):
    self.k = k
    ...
```

My knn method was

```
def knn(self, itemVector):
    """returns the predicted class of itemVector using k
    Nearest Neighbors"""
    # changed from min to heapq.nsmallest to get the
    # k closest neighbors
    neighbors = heapq.nsmallest(self.k,
                               [(self.manhattan(itemVector, item[1]), item)
                                for item in self.data])
    # each neighbor gets a vote
    results = {}
    for neighbor in neighbors:
        theClass = neighbor[1][0]
        results.setdefault(theClass, 0)
        results[theClass] += 1
    resultList = sorted([(i[1], i[0]) for i in results.items()],
                       reverse=True)
    #get all the classes that have the maximum votes
    maxVotes = resultList[0][0]
    possibleAnswers = [i[1] for i in resultList if i[0] == maxVotes]
    # randomly select one of the classes that received the max votes
    answer = random.choice(possibleAnswers)
    return( answer)
```

My slight modification to tenfold was:

```
def tenfold(bucketPrefix, dataFormat, k):  
    results = {}  
    for i in range(1, 11):  
        c = Classifier(bucketPrefix, i, dataFormat, k)
```

...

*You can download this code at guidetodatamining.com.
Remember, this is just one way to implement this method,
and it is not necessarily the best way.*



code it - part 2

Which makes the most difference? Having more data (comparing the results from pimaSmall and pima) or having a better algorithm (comparing k=1 to k=3)?



code it - results!

Here are my accuracy results (k=1 is the nearest neighbor algorithm from the last chapter):

	pimaSmall	pima
k=1	59.00%	71.247%
k=3	61.00%	72.519%

So it seems that roughly tripling the amount of data increases the accuracy much more than improving the algorithm does.





sharpen your pencil

Hmm. 72.519% seems like pretty good accuracy but is it? Compute the Kappa Statistic to find out:

	no diabetes	diabetes
no diabetes	219	44
diabetes	64	66

Performance:

- slightly good
- fair
- moderate
- substantially good
- near perfect



sharpen your pencil – answer

	no diabetes	diabetes	TOTAL
no diabetes	219	44	263
diabetes	64	66	130
TOTAL	283	110	393
ratio	0.7201	0.2799	

random (r) classifier:

	no diabetes	diabetes
no diabetes	189.39	73.61
diabetes	93.61	36.39

accuracy

$$p(r) = \frac{189.39 + 36.39}{393} = .5745$$

$$K = \frac{P(c) - P(r)}{1 - P(r)} = \frac{.72519 - .5745}{1 - .5745} = \frac{.15069}{.4255} = .35415$$

Only Fair performance

More data, better algorithms & a broken bus

Several years ago I was at a conference in Mexico City. This conference was a bit unusual in that it alternated between a day of presentations and a day of touring (the Monarch Butterflies, Inca ruins, etc). The days of touring involved riding long distances on a bus and the bus had a tendency to break down. As a result, a bunch of us PhD types spend a good deal of time standing at the side of road talking to one another as the bus



was being attended to. These roadside exchanges were the highpoint of the conference for me. One of the people I talked to was a person named Eric Brill. Eric Brill is famous for developing what is called the Brill tagger, which does part-of-speech tagging. Similar to what we have been doing in the last few chapters, the Brill tagger classifies data—in this case, it classifies words by their part of speech (noun, verb, etc.). The algorithm Brill came up with was significantly better than its predecessors (and as a result Brill became famous in natural language processing circles). At the side of that Mexican road, I got to talking with Eric Brill about improving the performance of algorithms. His view is that you get more of an improvement by getting more data for the training set, than you would by improving the algorithm. In fact, he felt that if he kept the original part-of-speech tagging algorithm and just increased the size of the training data, the improvement would exceed that of his famous algorithm. Although, he said, you cannot get a PhD for just collecting more data, but you can for developing an algorithm with marginally improved performance!

Here's another example. In various machine translation competitions, Google always places at the top. Granted that Google has a large number of very bright people developing great algorithms, much of Google's dominance is due to its enormous training sets it acquired from web.

更多数据 ⇔ Más dades ⇔ More data

This isn't to say that you shouldn't pick the best algorithm for the job. As we have already seen picking a good algorithm makes a significant difference. However, if you are trying to solve a practical problem (rather than publish a research paper) it might not be worth your while to spend a lot of time researching and tweaking algorithms. You will perhaps get more bang for your buck—or a better return on your time—if you concentrate on getting more data.

With that nod toward the importance of data, I will continue my path of introducing new algorithms.

People have used kNN classifiers for

- recommending items at Amazon
- assessing consumer credit risk
- classifying land cover using image analysis
- recognizing faces
- classifying the gender of people in images
- recommending web pages
- recommending vacation packages

Chapter 6: Probability and Naive Bayes

Naive Bayes

Let us return yet again to our women athlete example. Suppose I ask you what sport Brittney Griner participates in (gymnastics, marathon running, or basketball) and I tell you she is 6 foot 8 inches and weighs 207 pounds. I imagine you would say basketball and if I ask you how confident you feel about your decision I imagine you would say something along the lines of “pretty darn confident.”

Now I ask you what sport Heather Zurich (pictured on the right) plays. She is 6 foot 1 and weighs 176 pounds. Here I am less certain how you will answer. You might say ‘basketball’ and I ask you how confident you are about your prediction. You probably are less confident than you were about your prediction for Brittney Griner. She could be a tall marathon runner.

Finally, I ask you about what sport Yumiko Hara participates in; she is 5 foot 4 inches tall and weighs 95 pounds. Let’s say you say ‘gymnastics’ and I ask how confident you feel about your decision. You will probably say something along the lines of “not too confident.” A number of marathon runner have similar heights and weights.

With the nearest neighbor algorithms, it is difficult to quantify confidence about a classification. With classification methods based on probability—



Bayesian methods—we can not only make a classification but we can make probabilistic classifications—this athlete is 80% likely to be a basketball player, this patient has a 40% chance of getting diabetes in the next five years, the probability of rain in Las Cruces in the next 24 hours is 10%.

Nearest Neighbor approaches are called **lazy learners**. They are called this because when we give them a set of training data, they just basically save—or remember—the set. Each time it classifies an instance, it goes through the entire training dataset. If we have a 100,000 music tracks in our training data, it goes through the entire 100,000 tracks each time it classifies an instance.



Bayesian methods are called **eager learners**. When given a training set eager learners immediately analyze the data and build a model. When it wants to classify an instance it uses this internal model. Eager learners tend to classify instances faster than lazy learners.

The ability to make probabilistic classifications, and the fact that they are eager learners are two advantages of Bayesian methods.

Probability

I am assuming you have some basic knowledge of probability. I flip a coin; what is the probability of it being a 'heads'? I roll a 6 sided fair die, what is the probability that I roll a '1'? that sort of thing. I tell you I picked a random 19 year old and have you tell me the probability of that person being female and without doing any research you say 50%. These are examples of what is called prior probability and is denoted $P(h)$ —the probability of hypothesis h .

So for a coin:

$$P(\text{heads}) = 0.5$$

For a six sided dice, the probability of rolling a '1':

$$P(1) = 1/6$$

If I have an equal number of 19 yr. old male and females →

$$P(\text{female}) = .5$$

Suppose I give you some additional information about that 19 yr. old—the person is a student at the Frank Lloyd Wright School of Architecture in Arizona. You do a quick Google search, see that the student body is 86% female and revise your estimate of the likelihood of the person being female to 86%.

This we denote as $P(h|D)$ —the probability of the hypothesis h given some data D . For example:

$$P(\text{female} \mid \text{attends Frank Lloyd Wright School}) = 0.86$$

which we could read as “The probability the person is female given that person attends the Frank Lloyd Wright School is 0.86

The formula is

$$P(A | B) = \frac{P(A \cap B)}{P(B)}$$

An example.

In the following table I list some people and the types of laptops and phones they have:

name	laptop	phone
Kate	PC	Android
Tom	PC	Android
Harry	PC	Android
Annika	Mac	iPhone
Naomi	Mac	Android
Joe	Mac	iPhone
Chakotay	Mac	iPhone
Neelix	Mac	Android
Kes	PC	iPhone
B'Elanna	Mac	iPhone

What is the probability that a randomly selected person uses an iPhone?

There are 5 iPhone users out of 10 total users so

$$P(iPhone) = \frac{5}{10} = 0.5$$

What is the probability that a randomly selected person uses an iPhone given that person uses a Mac laptop?

$$P(iPhone | mac) = \frac{P(mac \cap iPhone)}{P(mac)}$$

First, there are 4 people who use both a Mac and an iPhone:

$$P(mac \cap iPhone) = \frac{4}{10} = 0.4$$

and the probability of a random person using a mac is

$$P(mac) = \frac{6}{10} = 0.6$$

So the probability of that some person uses an iPhone given that person uses a Mac is

$$P(iPhone | mac) = \frac{0.4}{0.6} = 0.667$$

That is the formal definition of posterior probability. Sometimes when we implement this we just use raw counts:

$$P(iPhone | mac) = \frac{\text{number of people who use a mac and an iPhone}}{\text{number of people who use a mac}}$$

$$P(iPhone | mac) = \frac{4}{6} = 0.667$$



sharpen your pencil

What's the probability of a person owning a mac given that they own an iPhone

i.e., $P(mac | iPhone)$?



tip

If you feel you need practice with basic probabilities please see the links to tutorials at guidetodatamining.com.



sharpen your pencil – solution

What's the probability of a person owning a mac given that they own an iPhone

i.e., $P(\text{mac}|\text{iPhone})$?

$$P(\text{mac} | \text{iPhone}) = \frac{P(\text{iPhone} \cap \text{mac})}{P(\text{iPhone})}$$
$$= \frac{0.4}{0.5} = 0.8$$



Some terms:

$P(h)$, the probability that some hypothesis h is true, is called the **prior probability** of h . Before we have any evidence, the probability of a person owning a Mac is 0.6 (the evidence might be knowing that the person also owns an iPhone).

$P(h|d)$ is called the **posterior probability** of h . After we observe some data d what is the probability of h ? For example, after we observe that a person owns an iPhone, what is the probability of that same person owning a Mac? It is also called **conditional probability**.

In our quest to build a Bayesian Classifier we will need two additional probabilities, $P(D)$ and $P(D|h)$. To explain these consider the following example.

Microsoft Shopping Cart

Did you know that Microsoft makes smart grocery store shopping carts? Yep, they do. Well, actually, Microsoft has contracted with a company called Chaotic Moon to develop them. Chaotic Moon's slogan is *We are smarter than you. We are more creative than you.* You can decide whether they are arrogant, cheeky, or something else. Anyway, the cart combines a shopping cart with a Windows 8 tablet, a Kinect, a Bluetooth speaker (so the cart can talk to you), and a mobile robotics platform (so the cart can follow you around the store).

You come in with your grocery store loyalty card. The cart recognizes you. It has recorded all previous purchases (as well as the purchases of everyone else in the store).



Suppose the cart software wants to determine whether to show you a targeted ad for Japanese Sencha Green Tea. It only wants to show that ad if you are likely to purchase the tea.

The cart system has accumulated the small dataset shown on the next page from other shoppers

$P(D)$ is the probability that some training data will be observed. For example, looking on the next page we see that the probability that the zip code will be 88005 is 5/10 or 0.5.

$$P(88005) = 0.5$$

$P(D|h)$ is the probability that some data value holds given the hypothesis. For example, the probability of the zip code being 88005 given that the person bought Sencha Green Tea or $P(88005|Sencha\ Tea)$.

Customer ID	Zipcode	bought organic produce?	bought Sencha green tea?
1	88005	Yes	Yes
2	88001	No	No
3	88001	Yes	Yes
4	88005	No	No
5	88003	Yes	No
6	88005	No	Yes
7	88005	No	No
8	88001	No	No
9	88005	Yes	Yes
10	88003	Yes	Yes

Zipcodes are a set of postal codes used in the U.S.

In this case we are looking at all the instances where the person bought Sencha Tea. There are 5 such instances. Of those, 3 are with the 88005 zip code.

$$P(88005 | \text{SenchaTea}) = \frac{3}{5} = 0.6$$



sharpen your pencil

What's the probability of the zip code being 88005 given that the person did not buy Sencha tea?



sharpen your pencil – solution

What's the probability of the zip code being 88005 given that the person did not buy Sencha tea?

There are 5 occurrences of a person not buying Sencha tea. Of those, 2 lived in the 88005 zip code. So

$$P(88005 | \neg \text{SenchaTea}) = \frac{2}{5} = 0.4$$

That \neg symbol means 'not'.



sharpen your pencil

This is key to understanding the rest of the chapter so let us practice just a bit more.

1. What is the probability of a person being in the 88001 zipcode (without knowing anything else)?
2. What is the probability of a person being in the 88001 zipcode knowing that they bought Sencha tea?
3. What is the probability of a person being in the 88001 zipcode knowing that they did not buy Sencha tea?



sharpen your pencil – solution

This is key to understanding the rest of the chapter so let us practice just a bit more.

1. What is the probability of a person being in the 88001 zipcode (without knowing anything else)?

There are 10 total entries in our database and only 3 of them are from 88001 so $P(88001)$ is 0.3

2. What is the probability of a person being in the 88001 zipcode knowing that they bought Sencha tea?

There are 5 instances of buying Sencha tea and only 1 of them is from the 88001 zipcode so

$$P(88001 | \text{SenchaTea}) = \frac{1}{5} = 0.2$$

3. What is the probability of a person being in the 88001 zipcode knowing that they did not buy Sencha tea?

There are 5 instances of not buying Sencha tea and 2 of them are from the 88001 zipcode:

$$P(88001 | \neg \text{SenchaTea}) = \frac{2}{5} = 0.4$$

Bayes Theorem

Bayes Theorem describes the relationship between $P(h)$, $P(h|D)$, $P(D)$, and $P(D|h)$:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

This theorem is the cornerstone of all Bayesian methods. Usually in data mining we use this theorem to decide among alternative hypotheses. Given the evidence, is the person a gymnast, marathoner, or basketball player. Given the evidence, will this person buy Sencha tea, or not. To decide among alternatives we compute the probability for each hypothesis. For example,

We want to display an ad for Sencha Tea on our smart shopping cart display only if we think that person is likely to buy the tea. We know that person lives in the 88005 zipcode.

There are two competing hypotheses:

The person will buy Sencha tea.
We compute $P(\text{buySenchaTea}|88005)$

The person will not buy Sencha tea.
We compute $P(\neg\text{buySenchaTea}|88005)$

We pick the hypothesis with the highest probability!

So if $P(\text{buySenchaTea}|88005) = 0.6$ and

$P(\neg\text{buySenchaTea}|88005) = 0.4$

So it is more likely that the person will buy the tea so we will display the ad.

Suppose we work for an electronics store and we have three sales flyers in email form. One flyer features a laptop, another features a desktop and the final flyer a tablet. Based on what we know about each customer we will email that customer the flyer that will most likely generate a sale. For example, I may know that a customer lives in the 88005 zipcode, that she has a college age daughter living at home, and that she goes to yoga class. Should I send her the flyer with the laptop, desktop, or tablet?

Let D represent all that I know about that customer:

- lives in 88005 zipcode
- has college age daughter
- goes to yoga class

My hypotheses are which flyer is the best: laptop, desktop, tablet. So I compute:

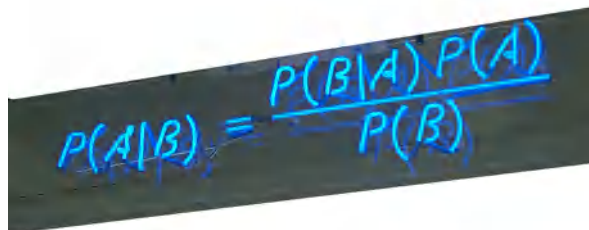
$$P(\text{laptop} \mid D) = \frac{P(D \mid \text{laptop})P(\text{laptop})}{P(D)}$$

$$P(\text{desktop} \mid D) = \frac{P(D \mid \text{desktop})P(\text{desktop})}{P(D)}$$

$$P(\text{tablet} \mid D) = \frac{P(D \mid \text{tablet})P(\text{tablet})}{P(D)}$$

And pick the hypothesis with the highest probability.

More abstractly, in a classification task we have a number of possible hypotheses: h_1, h_2, \dots, h_n . These hypotheses are the different categories of our task (for example, basketball players, marathoners, gymnasts, or 'will get diabetes', 'will not get diabetes').


$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

$$P(h_1 | D) = \frac{P(D | h_1)P(h_1)}{P(D)}, \quad P(h_2 | D) = \frac{P(D | h_2)P(h_2)}{P(D)}$$

,

$$\dots \quad P(h_n | D) = \frac{P(D | h_n)P(h_n)}{P(D)}$$

Once we compute all these probabilities, we will pick the hypothesis with the highest probability. This is called **the maximum a posteriori hypothesis**, or h_{MAP} .



We can translate that English description of calculating the maximum a posteriori hypothesis into the following formula:

$$h_{MAP} = \arg \max_{h \in H} P(h | D)$$

H is the set of all the hypotheses. So $h \in H$ means “for every hypothesis in the set of hypotheses.” The full formula means something like “for every hypothesis in the set of hypotheses compute $P(h|D)$ and pick the hypothesis with the largest probability.” Using Bayes Theorem we can convert that formula to:

$$h_{MAP} = \arg \max_{h \in H} \frac{P(D | h)P(h)}{P(D)}$$

So for every hypothesis we are going to compute:

$$\frac{P(D | h)P(h)}{P(D)}$$

You might notice that for all these calculations, the denominators are identical— $P(D)$. Thus, they are independent of the hypotheses. If a specific hypothesis has the max probability with the formula used above, it will still be the largest if we did not divide all the hypotheses by $P(D)$. If our goal is to find the most likely hypothesis, we can simplify our calculations:

$$h_{MAP} = \arg \max_{h \in H} P(D | h)P(h)$$

To see how this works, we will use an example from Tom M. Mitchell’s book, *Machine Learning*. Tom Mitchell is chair of the Machine Learning Department at Carnegie Mellon University. He is a great researcher and an extremely nice guy. On to the example from the book. Consider a medical domain where we want to determine whether a patient has a particular kind of cancer or not. We know that only 0.8% of the people in the U.S. have this form of cancer. There is a simple blood test we can do that will help us determine whether someone has it. The test is a binary one—it comes back either POS or NEG. When the disease is present the test returns a correct POS result 98% of the time; it returns a correct NEG result 97% of the time in cases when the disease is not present.

Our hypotheses:

- The patient has the particular cancer
- The patient does not have that particular cancer.



sharpen your pencil

Let's translate what I wrote above into probability notation. Please match up the English statements below with their associated notations and write in the probabilities. If there is no English statement matching a probability, please write one.

We know that only 0.8% of the people in the U.S. have this form of cancer.

$$P(\text{POS}|\text{cancer}) = \underline{\hspace{2cm}}$$

$$P(\text{POS}|\neg\text{cancer}) = \underline{\hspace{2cm}}$$

When the disease is present the test returns a correct POS result 98% of the time;

$$P(\text{cancer}) = \underline{\hspace{2cm}}$$

$$P(\neg\text{cancer}) = \underline{\hspace{2cm}}$$

it returns a correct NEG result 97% of the time in cases when the disease is not present

$$P(\text{NEG}|\text{cancer}) = \underline{\hspace{2cm}}$$

$$P(\text{NEG}|\neg\text{cancer}) = \underline{\hspace{2cm}}$$



sharpen your pencil – solution

We know that only 0.8% of the people in the U.S. have this form of cancer.

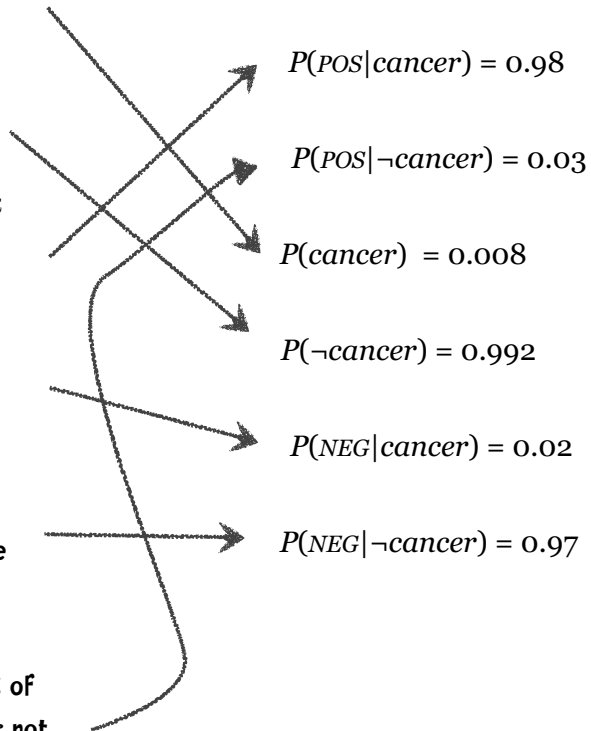
99.2% of people don't have this cancer

When the disease is present the test returns a correct POS result 98% of the time;

When the disease is present the test returns a incorrect NEG result 2% of

it returns a correct NEG result 97% of the time in cases when the disease is not present

it returns an incorrect POS result 3% of the time in cases when the disease is not present





sharpen your pencil – solution

Suppose Ann, comes into the doctor's office

A blood test for cancer is given and the test result is POS.

This is not looking good for Ann. After all, the test is 98% accurate.

Using Bayes Theorem determine whether it is more likely that Ann has cancer or that she does not.

$$P(\text{cancer}) = 0.008$$

$$P(\neg\text{cancer}) = 0.992$$

$$P(\text{POS}|\text{cancer}) = 0.98$$

$$P(\text{POS}|\neg\text{cancer}) = 0.03$$

$$P(\text{NEG}|\text{cancer}) = 0.02$$

$$P(\text{NEG}|\neg\text{cancer}) = 0.97$$





sharpen your pencil – solution

Suppose Ann, comes into the doctor's office

A blood test for the cancer is given and the test result is POS.

This is not looking good for Ann. After all, the test is 98% accurate.

Using Bayes Theorem determine whether it is more likely that Ann has cancer or that she does not.

We are finding the maximum a posteriori probability:

$$P(\text{POS} \mid \text{cancer})P(\text{cancer}) = .98(.008) = .0078$$

$$P(\text{POS} \mid \neg \text{cancer})P(\neg \text{cancer}) = .03(.992) = .0298$$

We select hMAP and classify the patient as not having cancer.

If we want to know the exact probability we can normalize these values by having them sum to 1:

$$P(\text{cancer} \mid \text{POS}) = \frac{0.0078}{0.0078 + 0.0298} = 0.21$$

Ann has a 21% chance of having cancer.



You may think "That just doesn't make sense. After all, the test is 98% accurate, but yet you're telling me Ann is most likely not to have cancer."

You are in good company. 85% of medical doctors get the answer wrong as well.

I just didn't make that 85% number up. See, among others,

Casscells, W., Schoenberger, A., and Grayboys, T. (1978): "Interpretation by physicians of clinical laboratory results." *N Engl J Med.* 299:999-1001.

Gigerenzer, Gerd and Hoffrage, Ulrich (1995): "How to improve Bayesian reasoning without instruction: Frequency formats." *Psychological Review.* 102: 684-704.

Eddy, David M. (1982): "Probabilistic reasoning in clinical medicine: Problems and opportunities." In D. Kahneman, P. Slovic, and A. Tversky, eds, *Judgement under uncertainty: Heuristics and biases.* Cambridge University Press, Cambridge, UK.

Here is why the results seem so counterintuitive. Most people see the statistic that 98% of the people who have this particular cancer will have a positive test result and also conclude that 98% of the people who have a positive test result have this particular cancer. This fails to take into account that this cancer affects only 0.8% of the population. Let's say we give the test to everyone in a city of 1 million people. That means that 8,000 people have cancer and 992,000 do not. First, let's consider giving the test to the 8,000 people with cancer. We know that 98% of the time when we give the test to people with cancer the test correctly returns a positive result. So 7,840 people have a correct positive result and 160 of those people with cancer have an incorrect negative result. Now let's turn to the 992,000 people without cancer. When we give the test to them, 97% of the time we get a correct negative result so $(992,000 * 0.97)$ or 962,240 of them have a correct negative result and 30,000 have an incorrect positive result. I have summarized these results on the following page.

	positive test result	negative test result
people with cancer	7,840	160
people without cancer	30,000	962,240

Now, consider Ann getting a positive test result and the data in the ‘positive test result’ column. 30,000 of the people with a positive test result had no cancer while only 7,840 of them had cancer. So it seems probable that Ann does not have cancer.

Still don't get it?
 Don't worry. Many people don't.
 After more practice you will gain a better understanding.

Why do we need Bayes Theorem?

Yet again, Bayes Theorem is

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

Let us return to the shopping cart example presented earlier. In that example, we obtained the information on the right from customers.

Say we know a customer lives in the 88005 zipcode and our two competing hypotheses are that they will buy Sencha tea or they will not. So:

$$P(h_1|D) = P(\text{buySenchaTea}|88005)$$

and

$$P(h_2|D) = P(\neg \text{buySenchaTea}|88005)$$

Customer ID	Zipcode	bought organic produce?	bought Sencha green tea?
1	88005	Yes	Yes
2	88001	No	No
3	88001	Yes	Yes
4	88005	No	No
5	88003	Yes	No
6	88005	No	Yes
7	88005	No	No
8	88001	No	No
9	88005	Yes	Yes
10	88003	Yes	Yes

In this case you may wonder why we need to compute

$$\frac{P(88005 | buySenchaTea)P(buySenchaTea)}{P(88005)}$$

when we can just as easily compute $P(buySenchaTea | 88005)$ directly from the data in the table. In this simple case you would be correct but for many real world problems it is very difficult to compute $P(h|D)$ directly.

Consider the previous medical example where we were interested in determining whether a person had cancer or not given that a certain test returned a positive result.

$$P(cancer | POS) \approx P(POS | cancer)P(cancer)$$

$$P(\neg cancer | POS) \approx P(POS | \neg cancer)P(\neg cancer)$$

It is relatively easy to compute the items on the right hand side. We can estimate $P(POS|cancer)$ by giving the cancer test to a representative sample of people with cancer and $P(POS|\neg cancer)$ by giving the test to a sample of people without cancer. $P(cancer)$ seems like a statistic that would be available on government websites and $P(\neg cancer)$ is simply

$$1 - P(cancer)$$

However, computing $P(cancer|POS)$ directly would be significantly more challenging. This is asking us to determine the probability that when we give the test to a random average person in the entire population and the test result is POS then that person has cancer. To do this we want a representative sample of the population but since only 0.8% of people have cancer a sample size of 1,000 people would only have 8 people with cancer—far too few to feel that our counts are representative of the population as a whole. So we would need an extremely large sample size. So Bayes Theorem provides a strategy for computing $P(h|D)$ when it is hard to do so directly.

Naïve Bayes

Most of the time we have more evidence than just a single piece of data. In the Sencha tea example we had two types of evidence: zip code and whether the person purchased organic food. To compute the probability of an hypothesis given all the evidence, we simply multiply the individual probabilities. In this example

Code:

tea = Person buy Sencha tea

¬ tea = Person does not buy Sencha tea

$P(88005|tea)$ = probability that a person lives in the 88005 zipcode given that person bought Sencha tea.

etc.

Customer ID	Zipcode	bought organic produce?	bought Sencha green tea?
1	88005	Yes	Yes
2	88001	No	No
3	88001	Yes	Yes
4	88005	No	No
5	88003	Yes	No
6	88005	No	Yes
7	88005	No	No
8	88001	No	No
9	88005	Yes	Yes
10	88003	Yes	Yes

We would like to know whether a person who lives in the 88005 zipcode and bought organic produce will likely buy tea:

$P(tea|88005 \ \& \ organic)$ and for that we simply multiply the probabilities:

$$P(tea|88005 \ \& \ organic) = P(88005 \ | \ tea) P(organic \ | \ tea) P(tea) = .6(.8)(.5) = .24$$

$$P(\neg tea|88005 \ \& \ organic) = P(88005 \ | \ \neg tea) P(organic \ | \ \neg tea) P(\neg tea) = .4(.25)(.5) = .05$$

So a person who lives in the trendy 88005 zip code area and buys organic food is more likely to buy Sencha Green tea than not. So let's display the Green Tea ad on the shopping cart display!

Here's how Stephen Baker describes the smart shopping cart technology:

... here's what shopping with one of these carts might feel like. You grab a cart on the way in and swipe your loyalty card. The welcome screen pops up with a shopping list. It's based on patterns of your last purchases. Milk, eggs, zucchini, whatever. Smart systems might provide you with the quickest route to each item. Or perhaps they'll allow you to edit the list, to tell it, for example, never to promote cauliflower or salted peanuts again. This is simple stuff. But according to Accenture's studies, shoppers forget an average of 11 percent of the items they intend to buy. If stores can effectively remind us of what we want, it means fewer midnight runs to the convenience store for us and more sales for them.

Baker. 2008. P49.

The Numerati

I've mentioned this book by Stephen Baker several times. I highly encourage you to read this book. The paperback is only \$10 and it is a good late night read.

i100 i500

Let's say we are trying to help iHealth, a company that sells wearable exercise monitors that compete with the Nike Fuel and the Fitbit Flex. iHealth sells two models that increase in functionality: the i100 and the i500:



iHealth100:

heart rate, GPS (to compute miles per hour, etc), wifi to automatically connect to iHealth website to upload data.



iHealth500:

i100 features + pulse oximetry (oxygen in blood) + free 3G connection to iHealth website

They sell these online and they hired us to come up with a recommendation system for their customers. To get data to build our system when someone buys a monitor, we ask them to fill out the questionnaire. Each question in the questionnaire relates to an attribute. First, we ask them what their main reason is for starting an exercise program and have them select among three options: health, appearance or both. We ask them what their current exercise level is: sedentary, moderate, or active. We ask them how motivated they are: moderate or aggressive. And finally we ask them if they are comfortable with using technological devices. Our results are as follows.

Main Interest	Current Exercise Level	How Motivated	Comfortable with tech. Devices?	Model #
both	sedentary	moderate	yes	i100
both	sedentary	moderate	no	i100
health	sedentary	moderate	yes	i500
appearance	active	moderate	yes	i500
appearance	moderate	aggressive	yes	i500
appearance	moderate	aggressive	no	i100
health	moderate	aggressive	no	i500
both	active	moderate	yes	i100
both	moderate	aggressive	yes	i500
appearance	active	aggressive	yes	i500
both	active	aggressive	no	i500
health	active	moderate	no	i500
health	sedentary	aggressive	yes	i500
appearance	active	moderate	no	i100
health	sedentary	moderate	no	i100



sharpen your pencil

Using the naïve Bayes method, which model would you recommend to a person whose
 main interest is health
 current exercise level is moderate
 is moderately motivated
 and is comfortable with technological devices

Turn the page if you need a hint!



sharpen your pencil clue

Ok. So we want to compute

$P(i100 \mid \text{health, moderateExercise, moderateMotivation, techComfortable})$

and

$P(i500 \mid \text{health, moderateExercise, moderateMotivation, techComfortable})$

and pick the model with the highest probability.

Let me lay out what we need to do for the first one:

$P(i100 \mid \text{health, moderateExercise, moderateMotivation, techComfortable}) =$

$\frac{P(\text{health} \mid i100) P(\text{moderateExercise} \mid i100) P(\text{moderateMotivated} \mid i100) P(\text{techComfortable} \mid i100)}{P(i100)}$

So here is what we need to first compute

$P(\text{health} \mid i100) = 1/6$ ←

There were 6 occurrences of people buying i100s and only one of those people had a main interest of 'health'

$P(\text{moderateExercise} \mid i100) =$

$P(\text{moderateMotivated} \mid i100) =$

$P(\text{techComfortable} \mid i100) =$

$P(i100) = 6 / 15$

That was my clue. Now hopefully you can figure out the example



sharpen your pencil solution

First we compute

$P(i100 \mid \text{health, moderateExercise, moderateMotivation, techComfortable})$

which equals the product of all these terms:

$P(\text{health} \mid i100) P(\text{moderateExercise} \mid i100) P(\text{moderateMotivated} \mid i100)$
 $P(\text{techComfortable} \mid i100) P(i100)$

$P(\text{health} \mid i100) = 1/6$

$P(\text{moderateExercise} \mid i100) = 1/6$

$P(\text{moderateMotivated} \mid i100) = 5/6$

$P(\text{techComfortable} \mid i100) = 2/6$

$P(i100) = 6 / 15$

so

$P(i100 \mid \text{evidence}) = .167 * .167 * .833 * .333 * .4 = .00309$

Now we compute

$P(i500 \mid \text{health, moderateExercise, moderateMotivation, techComfortable})$

$P(\text{health} \mid i500) = 4/9$

$P(\text{moderateExercise} \mid i500) = 3/9$

$P(\text{moderateMotivated} \mid i500) = 3/9$

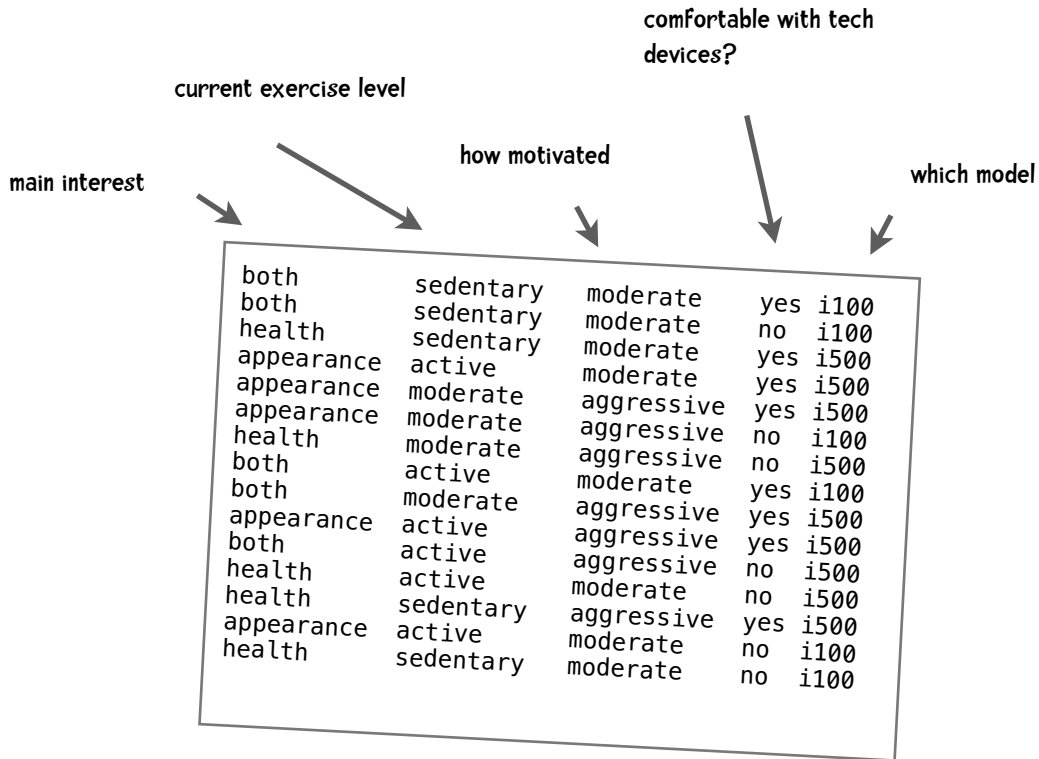
$P(\text{techComfortable} \mid i500) = 6/9$

$P(i500) = 9 / 15$

$P(i500 \mid \text{evidence}) = .444 * .333 * .333 * .667 * .6 = .01975$

Doing it in Python

Great! Now that we understand how a Naïve Bayes Classifier works let us consider how to implement it in Python. The format of the data files will be the same as that in the previous chapter, a text file where each line consists of tab-separated values. For our iHealth example, the data file would look like the following:



Shortly we will be using an example with substantially more data and I would like to keep the ten-fold cross validation methods we used in code from the previous chapter. Recall that that method involved dividing the data into ten buckets (files). We would train on nine of them and test the classifier on the remaining bucket. And we would repeat this ten times; each time withholding a different bucket for testing. The simple iHealth example, with only 15 instances, was designed so we could work through the Naïve Bayes Classifier method by hand. With only 15 instances it seems silly to divide them into 10 buckets. The ad hoc, not very elegant solution we will use, is to have ten buckets but all the 15 instances will be in one bucket and the rest of the buckets will be empty.

The Naïve Bayes Classifier code consists of two components, one for training and one for classifying.

Training

The output of training needs to be:

- a set of prior probabilities—for example, $P(i100) = 0.4$
- a set of conditional probabilities—for example, $P(\text{health}|i100) = 0.167$

I am going to represent the set of prior probabilities as a Python dictionary (hash table):

```
self.prior = {'i500': 0.6, 'i100': 0.4}
```

The conditional probabilities are a bit more complex. My way of doing this—and there are probably better methods—is to associate a set of conditional probabilities with each class.

```
{'i500': {1: {'appearance': 0.3333333333333333, 'health': 0.4444444444444444,
             'both': 0.2222222222222222},
          2: {'sedentary': 0.2222222222222222, 'moderate': 0.3333333333333333,
             'active': 0.4444444444444444},
          3: {'moderate': 0.3333333333333333, 'aggressive': 0.6666666666666666},
          4: {'no': 0.3333333333333333, 'yes': 0.6666666666666666}},
 'i100': {1: {'appearance': 0.3333333333333333, 'health': 0.1666666666666666,
             'both': 0.5},
          2: {'sedentary': 0.5, 'moderate': 0.1666666666666666,
             'active': 0.3333333333333333},
          3: {'moderate': 0.8333333333333334, 'aggressive': 0.1666666666666666},
          4: {'no': 0.6666666666666666, 'yes': 0.3333333333333333}}}
```

The 1, 2, 3, 4 represent column numbers. So the first line of the above is “the probability of the value of the first column being ‘appearance’ given that the device is i500 is 0.333.”

The first step in computing these probabilities is simply to count things. Here are the first few lines of the input file:

```
both      sedentary  moderate  yes i100
both      sedentary  moderate  no  i100
health    sedentary  moderate  yes i500
appearance active      moderate  yes i500
```

Yet again I am going to use dictionaries. One, called, `classes`, which will count the occurrences of each class or category. So, after the first line `classes` will look like

```
{'i100': 1}
```

After the second line:

```
{'i100': 2}
```

And after the third:

```
{'i500': 1, 'i100': 2}
```

After I process all the data, the value of `classes` is

```
{'i500': 9, 'i100': 6}
```

To obtain the prior probabilities I simply divide those number by the total number of instances.

To determine the conditional probabilities I am going to count the occurrences of attribute values in the different columns in a dictionary called `counts`. and I am going to maintain separate counts for each class. So, in processing the string 'both' in the first line, `counts` will be:

```
{'i100': {1: {'both': 1}}}
```

and at the end of processing the data, the value of `counts` will be

Counting things



Prior probability

Conditional probability

```
{'i100': {1: {'appearance':2, 'health': 1, 'both': 3},
          2: {'active': 2, 'moderate': 1, 'sedentary': 3},
          3: {'moderate': 5, 'aggressive': 1},
          4: {'yes': 2, 'no': 4}},
         'i500': {1: {'health': 4, 'appearance': 3, 'both': 2},
                  2: {'active': 4, 'moderate': 3, 'sedentary': 2},
                  3: {'moderate': 3, 'aggressive': 6},
                  4: {'yes': 6, 'no': 3}}}
```

So, in the first column of the i100 instances there were 2 occurrences of ‘appearance’, 1 of ‘health’ and 3 of ‘both’. To obtain the conditional probabilities we divide those numbers by the total number of instances of that class. For example, there are 6 instances of i100 and 2 of them had a value of ‘appearance’ for the first column, so

$$P(\text{‘appearance’} | i100) = 2/6 = .333$$

With that background here is the Python code for training the classifier (remember, you can download this code at guidetodatamining.com).

```
# _____

class BayesClassifier:
    def __init__(self, bucketPrefix, testBucketNumber, dataFormat):

        """ a classifier will be built from files with the bucketPrefix
        excluding the file with textBucketNumber. dataFormat is a
        string that describes how to interpret each line of the data
        files. For example, for the iHealth data the format is:
        "attr attr attr attr class"
        """

        total = 0
        classes = {}
        counts = {}

        # reading the data in from the file

        self.format = dataFormat.strip().split('\t')
        self.prior = {}
        self.conditional = {}
```

```

# for each of the buckets numbered 1 through 10:
for i in range(1, 11):
    #if it is not the bucket we should ignore, read in the data
    if i != testBucketNumber:
        filename = "%s-%02i" % (bucketPrefix, i)
        f = open(filename)
        lines = f.readlines()
        f.close()
        for line in lines:
            fields = line.strip().split('\t')
            ignore = []
            vector = []
            for i in range(len(fields)):
                if self.format[i] == 'num':
                    vector.append(float(fields[i]))
                elif self.format[i] == 'attr':
                    vector.append(fields[i])
                elif self.format[i] == 'comment':
                    ignore.append(fields[i])
                elif self.format[i] == 'class':
                    category = fields[i]
            # now process this instance
            total += 1
            classes.setdefault(category, 0)
            counts.setdefault(category, {})
            classes[category] += 1
            # now process each attribute of the instance
            col = 0
            for columnValue in vector:
                col += 1
                counts[category].setdefault(col, {})
                counts[category][col].setdefault(columnValue, 0)
                counts[category][col][columnValue] += 1

```

```

#
# ok done counting. now compute probabilities
#
# first prior probabilities p(h)
#
for (category, count) in classes.items():
    self.prior[category] = count / total
#
# now compute conditional probabilities p(h|D)
#
for (category, columns) in counts.items():
    self.conditional.setdefault(category, {})
    for (col, valueCounts) in columns.items():
        self.conditional[category].setdefault(col, {})
        for (attrValue, count) in valueCounts.items():
            self.conditional[category][col][attrValue] = (
                count / classes[category])

```

**That's it for training! No Complex math.
Just basic counting!!!**

Classifying

Okay, we have trained the classifier. Now we want to classify various instances. For example, which model should we recommend for someone whose primary interest is health, and who is moderately active, moderately motivated, and is comfortable with technology:

```
c.classify(['health', 'moderate', 'moderate', 'yes'])
```

For this we need to compute

$$h_{MAP} = \arg \max_{h \in H} P(D|h)P(h)$$

When we did this by hand we computing the probability of each hypothesis given the evidence and we simply translate that method to code:

```
def classify(self, itemVector):
    """Return class we think item Vector is in"""
    results = []
    for (category, prior) in self.prior.items():
        prob = prior
        col = 1
        for attrValue in itemVector:
            if not attrValue in self.conditional[category][col]:
                # we did not find any instances of this attribute value
                # occurring with this category so prob = 0
                prob = 0
            else:
                prob = prob * self.conditional[category][col][attrValue]
                col += 1
        results.append((prob, category))
    # return the category with the highest probability
    return(max(results)[1])
```

And when I try the code I get the same results we received when we did this by hand:

```
>>c = Classifier("iHealth/i", 10, "attr\tattr\tattr\tattr\tclass")
>>print(c.classify(['health', 'moderate', 'moderate', 'yes']))
i500
```



Republicans vs. Democrats

Let us look at a new data set, the Congressional Voting Records Data Set, available from the Machine Learning Repository (<http://archive.ics.uci.edu/ml/index.html>). It is available in a form that can be used by our programs at <http://guidetodatamining.com>. The data consists of the voting record of United States Congressional Representatives. The attributes are how that representative voted on 16 different bills.

Attribute Information:

1. Class Name: 2 (democrat, republican)
2. handicapped-infants: 2 (y,n)
3. water-project-cost-sharing: 2 (y,n)
4. adoption-of-the-budget-resolution: 2 (y,n)
5. physician-fee-freeze: 2 (y,n)
6. el-salvador-aid: 2 (y,n)
7. religious-groups-in-schools: 2 (y,n)
8. anti-satellite-test-ban: 2 (y,n)
9. aid-to-nicaraguan-contras: 2 (y,n)
10. mx-missile: 2 (y,n)
11. immigration: 2 (y,n)
12. synfuels-corporation-cutback: 2 (y,n)
13. education-spending: 2 (y,n)
14. superfund-right-to-sue: 2 (y,n)
15. crime: 2 (y,n)
16. duty-free-exports: 2 (y,n)
17. export-administration-act-south-africa: 2 (y,n)



The file consists of tab separated values:

democrat	y	n	y	n	n	n	y	y	y	n	n	n	n	n	y	y
democrat	y	y	y	n	n	n	y	y	y	y	n	n	n	n	y	y
democrat	y	y	y	n	n	n	y	y	n	n	n	n	n	y	n	y
republican	y	y	y	n	n	y	y	y	y	y	n	n	n	n	n	y

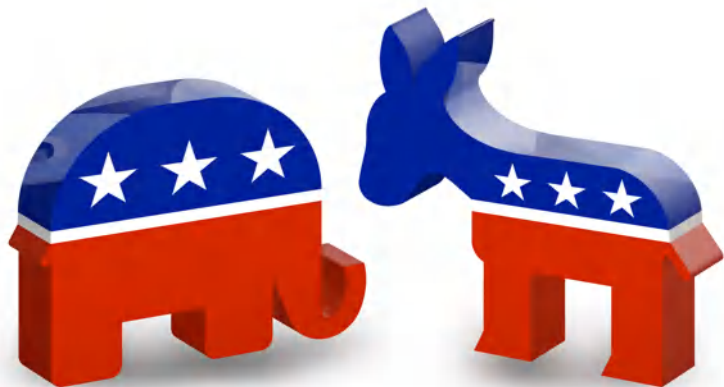
	CISPA	Reader Privacy Act	Internet Sales Tax	Internet Snooping Bill
Republican	0.99	0.01	0.99	0.5
Democrat	0.01	0.99	0.01	1.0

% voting 'yes'

This table shows that 99% of Republicans in the sample voted for the CISPA (Cyber Intelligence Sharing and Protection Act), only 1% voted for the Reader Privacy Act, 99% voted for Internet Sales Tax and 50% voted for the Internet Snooping Bill. (I made up these numbers and they do not reflect reality.) We pick a U.S. Representative who wasn't in our sample, Representative X, who we would like to classify as either a Democrat or Republican. I added how that representative voted to our table:

	CISPA	Reader Privacy Act	Internet Sales Tax	Internet Snooping Bill
Republican	0.99	0.01	0.99	0.5
Democrat	0.01	0.99	0.01	1.0
Rep. X	N	Y	N	N

Do you think the person is a Democrat or Republican?



DuckDuckGo

I would guess Democrat. Let us work through the example step-by-step using Naïve Bayes. The prior probabilities of $P(\text{Democrat})$ and $P(\text{Republican})$ are both 0.5 since there are 100 Republicans and 100 Democrats in the sample. We know that Representative X voted 'no' to CISPA and we also know

$$P(\text{Republican}|\text{C=no}) = 0.01 \quad \text{and} \quad P(\text{Democrat}|\text{C=no}) = 0.99$$

where C = CISPA. And with that bit of evidence our current $P(h|D)$ probabilities are

h=	p(h)	P(C=no h)				P(h D)
Republican	0.5	0.01				0.005
Democrat	0.5	0.99				0.495

Factoring in Representative X's 'yes' vote to the Reader Privacy Act and X's 'no' to the sales tax bill we get:

h=	p(h)	P(C=no h)	P(R=yes h)	P(T=no h)		P(h D)
Republican	0.5	0.01	0.01	0.01		0.0000005
Democrat	0.5	0.99	0.99	0.99		0.485

If we normalize these probabilities:

$$P(\text{Democrat} | D) = \frac{0.485}{0.485 + 0.0000005} = \frac{0.485}{0.4850005} = 0.99999$$

So far we are 99.99% sure Representative X is a Democrat.

Finally, we factor in Representative X's 'no' vote on the Internet Snooping Bill.

h=	p(h)	P(C=no h)	P(R=yes h)	P(T=no h)	P(S=no h)	P(h D)
Republican	0.5	0.01	0.01	0.01	0.50	2.5E-07
Democrat	0.5	0.99	0.99	0.99	0.00	0

Whoops. We went from 99.99% likelihood that X was a Democrat to 0%. This is so because we had 0 occurrences of a Democrat voting ‘no’ for the snooping bill. Based on these probabilities we predict that person X is a Republican. This goes against our intuition!

Estimating Probabilities

The probabilities in Naïve Bayes are really **estimates** of the true probabilities. True probabilities are those obtained from the entire population. For example, if we could give a cancer test to everyone in the entire population, we could, for example, get the true probability of the test returning a negative result given that the person does not have cancer. However, giving the test to everyone is near impossible. We can estimate that probability by selecting a random representative sample of the population, say 1,000 people, giving the test to them, and computing the probabilities. Most of the time this gives us a very good estimate of the true probabilities, but when the true probabilities are very small, these estimates are likely to be poor. Here is an example. Suppose the true probability of a Democrat voting no to the Internet Snooping Bill is 0.03— $P(S=no|Democrat) = 0.03$.



Brain Calisthenics

Suppose we try to estimate these probabilities by selected a sample of 10 Democrats and 10 Republicans. What is the most probable number of Democrats in the sample that voted no to the snooping bill?

0

1

2

3



Brain Calisthenics—answer

Suppose we try to estimate these probabilities by selected a sample of 10 Democrats and 10 Republicans. What is the most probable number of Democrats in the sample that voted no to the snooping bill?

0

So based on the sample $P(S=no|Democrat) = 0$.

As we just saw in the previous example, when a probability is 0 it dominates the Naïve Bayes calculation—it doesn't matter what the other values are. Another problem is that probabilities based on a sample produce a biased underestimate of the true probability.

Fixing this.

If we a trying to calculate something like $P(S=no|Democrat)$ our calculation has been

$$P(S=no|Democrat) = \frac{\text{the number that both are Democrats and voted no on the snooping bill.}}{\text{total number of Democrats}}$$

For expository ease let me simplify this by using shorter variable names:

$$P(x | y) = \frac{n_c}{n}$$

Here n is the total number of instances of class y in the training set; n_c is the total number of instances of class y that have the value x .



The problem we have is when n_c equals zero. We can eliminate this problem by changing the formula to:

$$P(x | y) = \frac{n_c + mp}{n + m}$$

This formula is from p179 of the book "Machine Learning" by Tom Mitchell.

m is a constant called the equivalent sample size. The method for determining the value of m varies. For now I will use the number of different values that attribute takes. For example, there are 2 values for how a person voted on the snooping bill, *yes*, or *no*. So I will use an m of 2. p is the prior estimate of the probability. Often we assume uniform probability. For example, what is the probability of someone voting no to the snooping bill knowing nothing about that person? $1/2$. So p in this case is $1/2$.



Let's go through the previous example to see how this works. First, here are tables showing the vote:

Republican Vote

	CISPA	Reader Privacy Act	Internet Sales Tax	Internet Snooping Bill
Yes	99	1	99	50
No	1	99	1	50

Democratic Vote

	CISPA	Reader Privacy Act	Internet Sales Tax	Internet Snooping Bill
Yes	1	99	1	100
No	99	1	99	0

The person we are trying to classify voted no to CISPA. First we compute the probability that he's a Republican given that vote. Our new formula is

$$P(x|y) = \frac{n_c + mp}{n + m}$$

n is the number of Republicans, which is 100 and n_c is the number of Republicans who voted no to CISPA, which is 1. m is the number of values for the attribute "how they voted on CISPA", which is 2 (*yes* or *no*). So plugging those number into our formula

$$P(cispa = no | republican) = \frac{1 + 2(.5)}{100 + 2} = \frac{2}{102} = 0.01961$$

We follow the same procedure for a person voting no to CISPA given they are a Democrat.

$$P(cispa = no | democrat) = \frac{99 + 2(.5)}{100 + 2} = \frac{100}{102} = 0.9804$$

With that bit of evidence our current P(h|D) probabilities are

h=	p(h)	P(C=no h)				P(h D)
Republican	0.5	0.01961				0.0098
Democrat	0.5	0.9804				0.4902

Factoring in Representative X's 'yes' vote to the Reader Privacy Act and X's 'no' to the sales



sharpen your pencil

Finish this problem and classify the individual as either a Republican or Democrat.

Recall, he voted no to Cisca, yes to the Reader Privacy act, and no both to the sales tax and snooping bills.



sharpen your pencil -answer

Finish this problem and classify the individual as either a Republican or Democrat.

Recall, he voted no to CISPA, yes to the Reader Privacy act, and no both to the Internet sales tax and snooping bills.

The calculations for the next 2 columns mirror those we did for the CISPA vote. The probability that this person voted no to the snooping bill given that he is a Republican is

$$P(s = no | republican) = \frac{50 + 2(.5)}{100 + 2} = \frac{51}{102} = 0.5$$

and that he voted no given that he is a Democrat:

$$P(s = no | democrat) = \frac{0 + 2(.5)}{100 + 2} = \frac{1}{102} = 0.0098$$

Multiplying those probabilities together gives us



h=	p(h)	P(C=no h)	P(R=yes h)	P(I=no h)	P(S=no h)	P(h D)
Republican	0.5	0.01961	0.01961	0.01961	0.5	0.000002
Democrat	0.5	0.9804	0.9804	0.9804	0.0098	0.004617

So unlike the previous approach we would classify this individual as a Democrat. This matches our intuitions.

A clarification

For this example, the value of m was 2 for all calculations. However, it is not the case that m remains necessarily constant across attributes. Consider the health monitor example discussed earlier in the chapter. The attributes for that example included:

survey

What is your main interest in getting a monitor?

- health
- appearance
- both

What is your current exercise level?

- sedentary
- moderate
- active

Are you comfortable with tech devices?

- yes
- no

For this attribute, $m = 3$ since the attribute can take one of 3 values (health, appearance, both). If we assume uniform probabilities, then $p = 1/3$ since the probability of the attribute being any one of the values is

This attribute also has $m = 3$ and $p = 1/3$

For this attribute, $m = 2$ since the attribute can take one of 2 values and $p = 1/2$ since the probability of the attribute being any one of those is $1/2$

Let us say the number of the people surveyed who own the i500 monitor is 100 (this is n). The number of people who own an i500 and are sedentary is 0 (n_c). So, the probability of someone being sedentary given they own an i500 is

$$P(\text{sedentary} | i500) = \frac{n_c + mp}{n + m} = \frac{0 + 3(.333)}{100 + 3} = \frac{1}{103} = 0.0097$$

Numbers

You probably noticed that I changed from **numerical** data which I used in all the nearest neighbor approaches I discussed to using **categorical** data for the naïve Bayes formula. By “categorical data” we mean that the data is put into discrete categories. For example, we divide people up in how they voted for a particular bill and the people who voted ‘yes’ go in one category and the people who voted ‘no’ go in another. Or we might categorize musicians by the instrument they play. So all saxophonists go in one bucket, all drummers in another, all pianists in another and so on. And these categories do not form a scale. So, for example, saxophonists are not ‘closer’ to pianists than they are to drummers. Numerical data is on a scale. An annual salary of \$105,000 is closer to a salary of \$110,000 than it is to one of \$40,000.

For Bayesian approaches we count things—how many occurrences are there of people who are sedentary—and it may not be initially obvious how to count things that are on a scale—for example, something like grade point average. There are two approaches.

Method 1: Making categories

One solution is to make categories by discretizing the continuous attribute. You often see this on websites and on survey forms. For example:

Age

- < 18
- 18-22
- 23-30
- 30-40
- > 40

Annual Salary

- > \$200,000
- 150,000 - 200,000
- 100,00 - 150,000
- 60,000-100,000
- 40,000-60,000

Once we have this information divided nicely into discrete values, we can use Naïve Bayes exactly as we did before.

Method 2: Gaussian distributions!

Harumph! Well I would take that income attribute and discretize it into distinct categories! Then we can use Naive Bayes!

That's sort of old school. I would just use a Gaussian distribution and deal with that attribute using a probability density function. We can still use Bayes.



The terms “Gaussian Distribution” and “Probability Density Function” sound cool, but they are more than good phrases to know so you can impress your friends at dinner parties. So what do they mean and how they can be used with the Naïve Bayes method? Consider the example we have been using with an added attribute of income:

Main Interest	Current Exercise Level	How Motivated	Comfortable with tech. Devices?	Income (in \$1,000)	Model #
both	sedentary	moderate	yes	60	i100
both	sedentary	moderate	no	75	i100
health	sedentary	moderate	yes	90	i500
appearance	active	moderate	yes	125	i500
appearance	moderate	aggressive	yes	100	i500
appearance	moderate	aggressive	no	90	i100
health	moderate	aggressive	no	150	i500
both	active	moderate	yes	85	i100
both	moderate	aggressive	yes	100	i500
appearance	active	aggressive	yes	120	i500
both	active	aggressive	no	95	i500
health	active	moderate	no	90	i500
health	sedentary	aggressive	yes	85	i500
appearance	active	moderate	no	70	i100
health	sedentary	moderate	no	45	i100

Let’s think of the typical purchaser of an i500, our awesome, premiere device. If I were to ask you to describe this person you might give me the average income:

$$mean = \frac{90 + 125 + 100 + 150 + 100 + 120 + 95 + 90 + 85}{9} = \frac{955}{9} = 106.111$$

And perhaps after reading chapter 4 you might give the standard deviation:

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

Recall that the standard deviation describes the range of scattering. If all the values are bunched up around the mean, the standard deviation is small; if the values are scattered the standard deviation is large



sharpen your pencil

What is the income standard deviation of the people who bought the i500? (those values are shown in the column below)

Income (in \$1,000)
90
125
100
150
100
120
95
90
85



sharpen your pencil - solution

What is the standard deviation of the income of the people who bought the i500?
(those values are shown in the column above)

Income (in \$1,000)	(x-106.111)	(x-106.111) ²
90	-16.111	259.564
125	18.889	356.794
100	-6.111	37.344
150	43.889	1926.244
100	-6.111	37.344
120	13.889	192.904
95	-11.111	123.454
90	-16.111	259.564
85	-21.111	445.674

$$\Sigma = 3638.889$$

$$sd = \sqrt{\frac{3638.889}{9}}$$

$$= \sqrt{404.321} = 20.108$$



Population standard deviation and sample standard deviation.

The formula for standard deviation that we just used is called the population standard deviation. It is called that because we use this formula when we have data on the entire population we are interested in. For example, we might give a test to 500 students and then compute the mean and standard deviation. In this case, we would use the population standard deviation, which is what we have been using. Often, though, we do not have data on the entire population. For example, suppose I am interested in the effects of drought on the deer mice in Northern New Mexico and as part of that study I want the average (mean) and standard deviation of their weights. In this case I am not going to weigh every mouse in Northern New Mexico. Rather I will collect and weigh some representative sample of mice.



For this sample, I can use the standard deviation formula I used above, but there is another formula that has been shown to be a better estimate of the entire population standard deviation. This formula is called the **sample standard deviation** and it is just a slight modification of the previous formula:

$$sd = \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{card(x) - 1}}$$

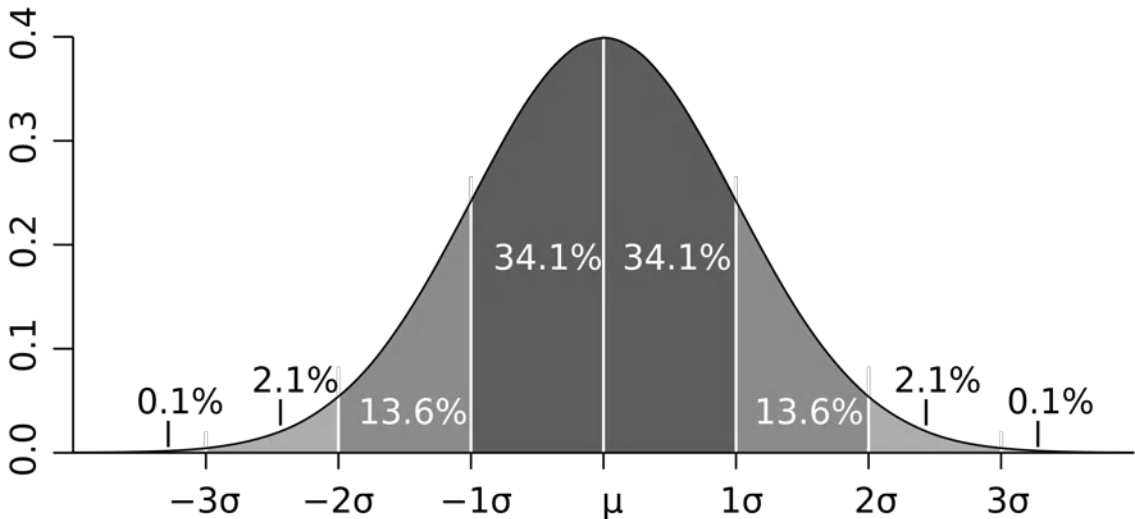
The sample standard deviation of the income example is

$$sd = \sqrt{\frac{3638.889}{9-1}} = \sqrt{\frac{3638.889}{8}}$$

$$= \sqrt{454.861} = 21.327$$

For the rest of this chapter we will be using sample standard deviation.

You probably have heard terms such as normal distribution, bell curve, and Gaussian distribution. Gaussian distribution is just a high falutin term for normal distribution. The function that describes this distribution is called the Gaussian function or bell curve. Most of the time the Numerati (aka data miners) assume attributes follow a Gaussian distribution. What it means is that about 68% of the instances in a Gaussian distribution fall within 1 standard deviation of the mean and 95% of the instances fall within 2 standard deviations of the mean:



In our case, the mean was 106.111 and the sample standard deviation was 21.327. So 95% of the people who purchase an i500 earn between \$42,660 and \$149,770. If I asked you if you thought $P(100k | i500)$ —the likelihood that an i500 purchaser earns \$100,000—was, you might think that's pretty likely. If I asked you what you thought the likelihood of $P(20k | i500)$ —the likelihood that an i500 purchaser earns \$20,000—was, you might think it was pretty unlikely.

To formalize this, we are going to use the mean and standard deviation to compute this probability as follows:

$$P(x_i | y_j) = \frac{1}{\sqrt{2\pi\sigma_{ij}}} e^{-\frac{(x_i - \mu_{ij})^2}{2\sigma_{ij}^2}}$$

Maybe putting the formula in a bigger font makes it look simpler!



Everytime I type a complex looking formula into this book, I feel the need to say something like “don’t panic.” It could be that none of you readers panic and I am just the one panicking.

However, let me say this. Data mining has professional terminology and formulas. Before you dive into data mining you might think “those things look difficult.” But after you study, even for a short time, these formulas become nothing special. It is just a matter of working through the formula out step-by-step.

Let’s jump right into dissecting this formula so we can see how simple it really is. Let us say we are interested in computing $P(100k|i500)$, the probability that a person earns \$100,000 (or 100k) given they purchased an i500. A few pages ago we computed the average income (mean) of people who bought the i500. We also computed the sample standard deviation. These values are shown on the following page. In Numerati speak, we represent the mean with the Greek letter μ (mu) and the standard deviation as σ (sigma).

$$P(x_i | y_j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} e^{-\frac{(x_i - \mu_{ij})^2}{2\sigma_{ij}^2}} \quad \begin{array}{l} \mu_{ij} = 106.111 \\ \sigma_{ij} = 21.327 \\ x_i = 100 \end{array}$$

Let's plug these values into the formula:

$$P(x_i | y_j) = \frac{1}{\sqrt{2\pi}(21.327)} e^{-\frac{(100-106.111)^2}{2(21.327)^2}}$$

and do some math:

$$P(x_i | y_j) = \frac{1}{\sqrt{6.283}(21.327)} e^{-\frac{(37.344)^2}{909.68}}$$

and more math:

$$P(x_i | y_j) = \frac{1}{53.458} e^{-0.0411}$$

The e is a mathematical constant that is the base of the natural logarithm. It's value is approximately 2.718.

$$P(x_i | y_j) = \frac{1}{53.458} (2.718)^{-0.0411} = (0.0187)(0.960) = 0.0180$$

So the probability that the income of a person who bought the i500 is \$100,000 is 0.0180.



sharpen your pencil

In the table below I have the horsepower ratings for cars that get 35 miles per gallon. I would like to know the probability of a Datsun 280z having 132 horsepower given it gets 35 miles per gallon.

car	HP
Datsun 210	65
Ford Fiesta	66
VW Jetta	74
Nissan Stanza	88
Ford Escort	65
Triumph tr7 coupe	88
Plymouth Horizon	70
Suburu DL	67

$$\mu_{ij} = \underline{\hspace{2cm}}$$

$$\sigma_{ij} = \underline{\hspace{2cm}}$$

$$x_i = \underline{\hspace{2cm}}$$



sharpen your pencil -solution - part 1

In the table below I have the horsepower ratings for cars that get 35 miles per gallon. I would like to know the probability of a Datsun 280z having 132 horsepower given it gets 35 miles per gallon.

car	HP
Datsun 210	65
Ford Fiesta	66
VW Jetta	74
Nissan Stanza	88
Ford Escort	65
Triumph tr7 coupe	88
Plymouth Horizon	70
Suburu DL	67

$$\mu_{ij} = 72,875$$

$$\sigma_{ij} = 9.804$$

$$x_i = 132$$

$$\sigma = \sqrt{\frac{(65 - \mu)^2 + (66 - \mu)^2 + (74 - \mu)^2 + (88 - \mu)^2 + (65 - \mu)^2 + (88 - \mu)^2 + (70 - \mu)^2 + (67 - \mu)^2}{7}}$$

$$\sigma = \sqrt{\frac{672.875}{7}} = \sqrt{96.126} = 9.804$$



sharpen your pencil -solution - part 2

In the table below I have the horsepower ratings for cars that get 35 miles per gallon. I would like to know the probability of a Datsun 280z having 132 horsepower given it gets 35 miles per gallon.

$$\mu_{ij} = 72,875$$

$$\sigma_{ij} = 9.804$$

$$P(x_i | y_j) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} e^{-\frac{(x_i - \mu_{ij})^2}{2\sigma_{ij}^2}}$$

$$x_i = 132$$

$$P(132hp | 35mpg) = \frac{1}{\sqrt{2\pi}(9.804)} e^{-\frac{(132-72.875)^2}{2(9.804)^2}}$$

$$= \frac{1}{\sqrt{6.283}(9.804)} e^{-\frac{-3495.766}{192.237}} = \frac{1}{24.575} e^{-18.185}$$

$$= 0.0407(0.00000001266)$$

$$= 0.0000000005152$$

Ok. it is extremely unlikely that a Datsun 280z, given that it gets 35 miles to the gallon has 132 horsepower. (but it does!)

A few implementation notes.

In the training phase for Naive Bayes, we will compute the mean and sample standard deviation of every numeric attribute. Shortly, we will see how to do this in detail.

In the classification phase, the above formula can be implemented with just a few lines of Python:

```
import math

def pdf(mean, ssd, x):
    """Probability Density Function computing P(x|y)
    input is the mean, sample standard deviation for all the items in y,
    and x."""
    ePart = math.pow(math.e, -(x-mean)**2/(2*ssd**2))
    return (1.0 / (math.sqrt(2*math.pi)*ssd)) * ePart
```

Let's test this with the examples we did above:

```
>>>pdf(106.111, 21.327, 100)
0.017953602706962717
```

```
>>>pdf(72.875, 9.804, 132)
5.152283971078022e-10
```

Whew! Time for a break!



Python Implementation

Training Phase

The Naïve Bayes method relies on prior and conditional probabilities. Let's go back to our Democrat/Republican example. Prior probabilities are the probabilities that hold before we have observed any evidence. For example, if I know there are 233 Republicans and 200 Democrats, then the prior probability of some arbitrary member of the U.S. House of Representatives being a Republican is

$$P(\text{republican}) = \frac{233}{433} = 0.54$$

This is denoted $P(h)$. Conditional Probability $P(h|D)$ is the probability that h holds given that we know D , for example, $P(\text{democrat}|\text{bill}_1\text{Vote}=\text{yes})$. In Naïve Bayes, we flip that probability and compute $P(D|h)$ —for example, $P(\text{bill}_1\text{Vote}=\text{yes}|\text{democrat})$.

In our existing Python program we store these conditional probabilities in a dictionary of the following form:

```
{'democrat': {'bill 1': {'yes': 0.333, 'no': 0.667},  
              'bill 2': {'yes': 0.778, 'moderate': 0.222}},  
 'republican': {'bill 1': {'yes': 0.811, 'no': 0.189},  
                'bill 2': {'yes': 0.250, 'no': 0.750}}}
```

So the probability that someone voted yes to bill 1 given that they are a Democrat ($P(\text{bill}_1=\text{yes}|\text{democrat})$) is 0.333.

We will keep this data structure for attributes whose values are discrete values (for example, 'yes', 'no', 'sex=male', 'sex=female'). However, when attributes are numeric we will be using the probability density function and we need to store the mean and sample standard deviation for that attribute. For these numeric attributes I will use the following structures:

```
mean = {'democrat': {'age': 57, 'years served': 12}
        'republican': {'age': 53, 'years served': 7}}

ssd = {'democrat': {'age': 7, 'years served': 3}
        'republican': {'age': 5, 'years served': 5}}
```

As before, each instance is represented by a line in a data file. The attributes of each instances are separated by tabs. For example, the first few lines of a data file for the Pima Indians Diabetes Data set might be:

3	78	50	32	88	31.0	0.248	26	1
4	111	72	47	207	37.1	1.390	56	1
1	189	60	23	846	30.1	0.398	59	1
1	117	88	24	145	34.5	0.403	40	1
3	107	62	13	48	22.9	0.678	23	1
7	81	78	40	48	46.7	0.261	42	0
2	99	70	16	44	20.4	0.235	27	0
5	105	72	29	325	36.9	0.159	28	0
2	142	82	18	64	24.7	0.761	21	0
1	81	72	18	40	26.6	0.283	24	0
0	100	88	60	110	46.8	0.962	31	0

The columns represent, in order, the number of times pregnant, plasma glucose concentration, blood pressure, triceps skin fold thickness, serum insulin level, body mass index, diabetes pedigree function, age, and a '1' in the last column represents that they developed diabetes and a '0' they did not.

Also as before, we are going to represent how the program should interpret each column by use of a format string, which uses the terms

- `at tr` identifies columns that should be interpreted as non-numeric attributes, and which will use the Bayes methods shown earlier in this chapter.
- `num` identifies columns that should be interpreted as numeric attributes, and which will use the Probability Density Function (so we will need to compute the mean and standard deviation during training).
- `class` identifies the column representing the class of the instance (what we are trying to learn)

In the Pima Indian Diabetes data set the format string will be

```
"num num num num num num num num class"
```

To compute the mean and sample standard deviation we will need some temporary data structures during the training phase. Again, let us look at a small sample of the Pima data set.

3	78	50	32	88	31.0				
4	111	72	47	207	37.1	0.248	26	1	
1	189	60	23	846	30.1	1.390	56	1	
2	142	82	18	64	24.7	0.398	59	1	
1	81	72	18	40	26.6	0.761	21	0	
0	100	88	60	110	46.8	0.283	24	0	
						0.962	31	0	

The last column represents the class of each instance. So the first three individuals developed diabetes and that last three did not. All the other columns represent numeric attributes. of which we need to compute the mean and standard deviation for each of the two classes. To compute the mean for each class and attribute I will need to keep track of the running total. In our existing code we already keep track of the total number of instances. I will implement this using a dictionary:

```
totals    {'1': {1: 8, 2: 378, 3: 182, 4: 102, 5: 1141,  
            6: 98.2, 7: 2.036, 8: 141},  
          {'0': {1: 3, 2: 323, 3: 242, 4: 96, 5: 214,  
            6: 98.1, 7: 2.006, 8: 76}}
```

So for class 1, the column 1 total is 8 (3 + 4 + 1), the column 2 total is 378, etc.

For class 0, the column 1 total is 3 (2 + 1 + 0), the column 2 total is 323 and so on.

For standard deviation, we will also need to keep the original data, and for that we will use a dictionary in the following format:

numericValues

```
{'1': 1: [3, 4, 1], 2: [78, 111, 189], ...},
{'0': {1: [2, 1, 0], 2: [142, 81, 100]}
```

I have added the code to create these temporary data structures to the `__init__()` method of our Classifier class as shown below:

```
import math

class Classifier:
    def __init__(self, bucketPrefix, testBucketNumber, dataFormat):

        """ a classifier will be built from files with the bucketPrefix
        excluding the file with textBucketNumber. dataFormat is a string that
        describes how to interpret each line of the data files. For example,
        for the iHealth data the format is:
        "attrattr attr attr class"
        """
        total = 0
        classes = {}
        # counts used for attributes that are not numeric
        counts = {}
        # totals used for attributes that are numereric
        # we will use these to compute the mean and sample standard deviation
        # for each attribute - class pair.
        totals = {}
        numericValues = {}

        # reading the data in from the file
        self.format = dataFormat.strip().split('\t')
        #
        self.prior = {}
        self.conditional = {}

        # for each of the buckets numbered 1 through 10:
        for i in range(1, 11):
            # if it is not the bucket we should ignore, read in the data
            if i != testBucketNumber:
                filename = "%s-%02i" % (bucketPrefix, i)
                f = open(filename)
```

```

lines = f.readlines()
f.close()
for line in lines:
    fields = line.strip().split('\t')
    ignore = []
    vector = []
    nums = []
    for i in range(len(fields)):
        if self.format[i] == 'num':
            nums.append(float(fields[i]))
        elif self.format[i] == 'attr':
            vector.append(fields[i])
        elif self.format[i] == 'comment':
            ignore.append(fields[i])
        elif self.format[i] == 'class':
            category = fields[i]
    # now process this instance
    total += 1
    classes.setdefault(category, 0)
    counts.setdefault(category, {})
    totals.setdefault(category, {})
    numericValues.setdefault(category, {})
    classes[category] += 1
    # now process each non-numeric attribute of the instance
    col = 0
    for columnValue in vector:
        col += 1
        counts[category].setdefault(col, {})
        counts[category][col].setdefault(columnValue, 0)
        counts[category][col][columnValue] += 1
    # process numeric attributes
    col = 0
    for columnValue in nums:
        col += 1
        totals[category].setdefault(col, 0)
        #totals[category][col].setdefault(columnValue, 0)
        totals[category][col] += columnValue
        numericValues[category].setdefault(col, [])
        numericValues[category][col].append(columnValue)

```

```

#
# ok done counting. now compute probabilities
# first prior probabilities p(h)
#
for (category, count) in classes.items():
    self.prior[category] = count / total
#
# now compute conditional probabilities p(h|D)
#
for (category, columns) in counts.items():
    self.conditional.setdefault(category, {})
    for (col, valueCounts) in columns.items():
        self.conditional[category].setdefault(col, {})
        for (attrValue, count) in valueCounts.items():
            self.conditional[category][col][attrValue] = (
                count / classes[category])
self.tmp = counts
#
# now compute mean and sample standard deviation
#

```



code it

Can you add the code to compute the means and standard deviations? Download the file `naiveBayesDensityFunctionTraining.py` from guidetodatamining.com.

Your program should produce the data structures `ssd` and `means`:

```

c = Classifier("pimaSmall/pimaSmall", 1,
              "num num num num num num num class")
>>> c.ssd
{'0': {1: 2.54694671925252, 2: 23.454755259159146, ...},
 '1': {1: 4.21137914295475, 2: 29.52281872377408,}}
>>> c.means
{'0': {1: 2.8867924528301887, 2: 111.90566037735849, ...},
 '1': {1: 5.25, 2: 146.05555555555554, ...}}

```



code it solution

Here is my solution:

```
#
# now compute mean and sample standard deviation
#
self.means = {}
self.ssd = {}
self.totals = totals
for (category, columns) in totals.items():
    self.means.setdefault(category, {})
    for (col, cTotal) in columns.items():
        self.means[category][col] = cTotal / classes[category]
# standard deviation

for (category, columns) in numericValues.items():

    self.ssd.setdefault(category, {})
    for (col, values) in columns.items():
        SumOfSquareDifferences = 0
        theMean = self.means[category][col]
        for value in values:
            SumOfSquareDifferences += (value - theMean)**2
        columns[col] = 0
        self.ssd[category][col] = math.sqrt(SumOfSquareDifferences
            / (classes[category] - 1))
```

The file containing this solution is `naiveBayesDensityFunctionTrainingSolution.py` at our website.



code it 2

Can you revise the `classify` method so it uses the probability density function for numeric attributes? The file to modify is `naiveBayesDensityFunctionTemplate.py`. Here is the original `classify` method:

```
def classify(self, itemVector, numVector):
    """Return class we think item Vector is in"""
    results = []
    sqrt2pi = math.sqrt(2 * math.pi)
    for (category, prior) in self.prior.items():
        prob = prior
        col = 1
        for attrValue in itemVector:
            if not attrValue in self.conditional[category][col]:
                # we did not find any instances of this attribute value
                # occurring with this category so prob = 0
                prob = 0
            else:
                prob = prob * self.conditional[category][col][attrValue]
            col += 1
    # return the category with the highest probability
    #print(results)
    return(max(results)[1])
```





code it 2 - solution

Can you revise the classify method so it uses the probability density function for numeric attributes? The file to modify is naiveBayesDensityFunctionTemplate.py.

Solution:

```
def classify(self, itemVector, numVector):
    """Return class we think item Vector is in"""
    results = []
    sqrt2pi = math.sqrt(2 * math.pi)
    for (category, prior) in self.prior.items():
        prob = prior
        col = 1
        for attrValue in itemVector:
            if not attrValue in self.conditional[category][col]:
                # we did not find any instances of this attribute
                # occurring with this category so prob = 0
                prob = 0
            else:
                prob = prob * self.conditional[category][col]
                [attrValue]
                col += 1
        col = 1
        for x in numVector:
            mean = self.means[category][col]
            ssd = self.ssd[category][col]
            ePart = math.pow(math.e, -(x - mean)**2/(2*ssd**2))
            prob = prob * ((1.0 / (sqrt2pi*ssd)) * ePart)
            col += 1
        results.append((prob, category))
    # return the category with the highest probability
    #print(results)
    return(max(results)[1])
```

Is this any better than the Nearest Neighbor Algorithm?

In Chapter 5 we evaluated how well the k Nearest Neighbor algorithm did with both the total Pima data set and a subset. Here are those results:

	pimaSmall	pima
k=1	59.00%	71.247%
k=3	61.00%	72.519%

Here are the results when we use Naïve Bayes with these two data sets:

	pimaSmall	pima
Bayes	72.000%	77.354%



Wow! So it looks like Naïve Bayes performs better than kNN!

The kappa score for the kNN where k=3 on the large data set was 0.35415, only fair performance. I wonder what kappa is for Naïve Bayes?

ORIGINAL Σ

219	44	263
45	85	130
Σ 264	Σ 129	393

Σ 67176 | 32824

RANDOM

176.675	86.327
87.329	42.671

$\Rightarrow P(r) = \frac{219,344}{393} = .558127$

$K = \frac{P(c) - P(r)}{1 - P(r)} = \frac{.77354 - .558127}{1 - .558127}$

$= \frac{.215412}{.441872} = \underline{\underline{.4875}}$

The kappa is 0.4875, moderate agreement!

So for this example, Naïve Bayes is better than k

Advantages of Bayes

- simple to implement (just counting things)
- need less training data than many other methods
- a good method to use if you want something that performs well and has good performance times.

Main disadvantage of Bayes:

It cannot learn interactions among features. For example, it cannot learn that I like foods with cheese and I like foods with rice but I do not like foods with both

Advantages of kNN

- simple to implement.
- does not assume the data has any particular structure—a good thing!
- large amount of memory needed to store the training set.

kNN

k Nearest Neighbors is a reasonable choice when the training set is large. kNN is extremely versatile and used in a large number of fields including recommendation systems, proteomics (the study of the entire protein set of an organism), the interaction among proteins, and image classification.



What enables us to multiple probabilities together is the fact that the events these probabilities represent are independent. For example, consider a game where we flip a coin and roll a die. These events are independent meaning what we roll on the die does not depend on whether we flip a heads or tails on the coin. And, as I just said, if events are independent we can determine their joint probability (the probability that they both occurred) by multiplying the individual probabilities together. So the probability of getting a heads and rolling a 6 is

$$P(\text{heads} \wedge 6) = P(\text{heads}) \times P(6) = 0.5 \times \frac{1}{6} = 0.08333$$

Let's say I alter a deck of cards keeping all the black cards (26 of them) but only retaining the face cards for the red suits (6 of them). That makes a 32 card deck. What is the probability of selecting a face card?

$$P(\text{facecard}) = \frac{12}{32} = 0.375$$

The probability of selecting a red card is

$$P(\text{red}) = \frac{6}{32} = 0.1875$$

What is the probability of selecting a single card that is both red and a face card? Here we do not multiply probabilities. We **do not** do

$$P(\text{red} \wedge \text{facecard}) = P(\text{red}) \times P(\text{facecard}) = 0.375 \times 0.185 = 0.0703$$

Here is what our common sense tells us. The chance of picking a red card is .1875. But if we pick a red card it is 100% likely it will be a face card. So it seems that the probability of picking a card that is both red and a face card is .1875.

Or we can start a different way. The probability of picking a face card is .375. The way the deck is arranged half the face cards are red. So the probability of picking a card that is both red and a face card is $.375 * .5 = .1875$.

Here we cannot multiply probabilities together because the attributes are not independent—if we pick red the probability of a face card changes—and vice versa.

In many if not most real world data mining problems there are attributes that are not independent.

Consider the athlete data. Here we had 2 attributes weight and height. Weight and height are not independent. The taller you get the more likely you will be heavier.

Suppose I have attributes zip code, income, and age. These are not independent. Certain zipcodes have big bucks houses others consist of trailer parks. Palo Alto zipcodes may be dominated by 20-somethings—Arizona zipcodes may be dominated by retirees.

Think about the music attributes—things like amount of distorted guitar (1-5 scale), amount of classical violin sound. Here many of these attributes are not independent. If I have a lot of distorted guitar sound, the probability of having a classical violin sound decreases.

Suppose I have a dataset consisting of blood test results. Many of these values are not independent. For example, there are multiple thyroid tests including free T4 and TSH. There is an inverse relationship between the values of these two tests.

Think about cases yourself. For example, consider attributes of cars. Are they independent? Attributes of a movie? Amazon purchases?

So, for Bayes to work we need to use attributes that are independent, but most real-world problems violate that condition. What we are going to do is just to assume that they are independent! We are using the magic wand of sweeping things under the rug™—and ignoring this problem. We call it *naïve Bayes* because we are naïvely assuming independence even though we know it is not. It turns out that naïve Bayes works really, really, well even with this naïve assumption.



code it

Can you run the naïve Bayes code on our other data sets? For example, our kNN algorithm was 53% accurate on the auto MPG data set. Does a Bayes approach produce better results?

```
tenfold("mpgData/mpgData", "class attr num num num num comment")
?????
```

Chapter 7: Naïve Bayes and Text

Classifying unstructured text

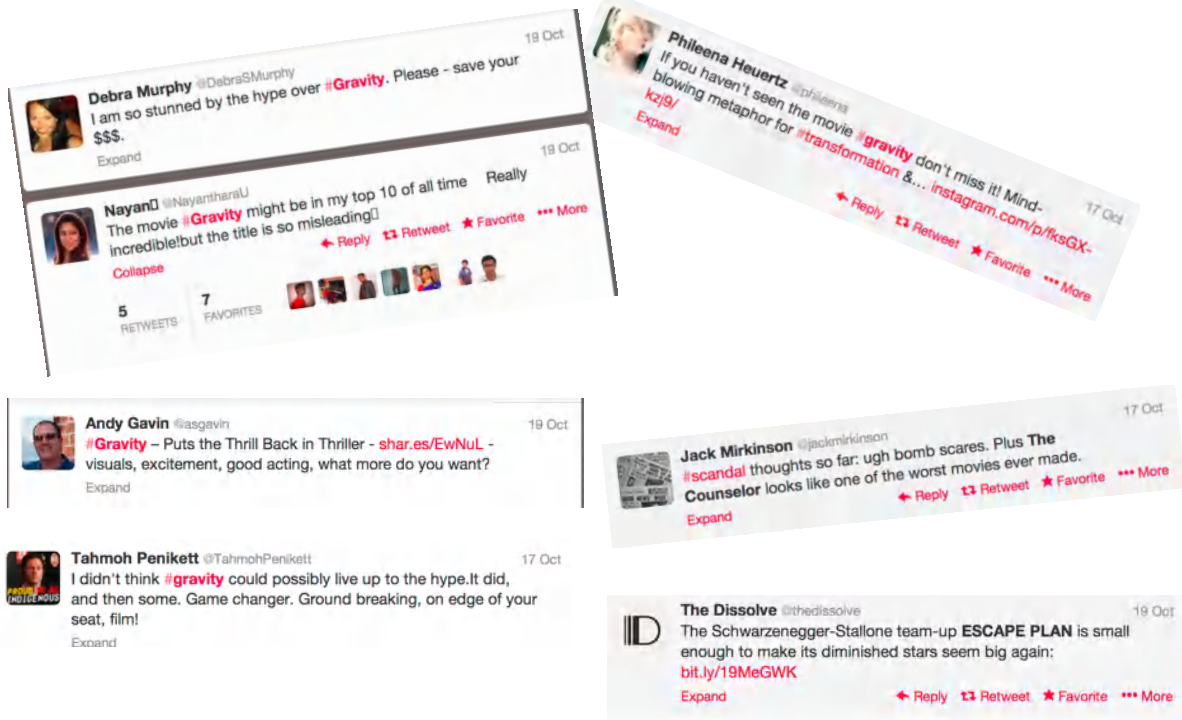
In previous chapters we've looked at recommendation systems that have people explicitly rate things with star systems (5 stars for Phoenix), thumbs-up/thumbs-down (Inception--thumbs-up!), and numerical scales. We've looked at implicit things like the behavior of people—did they buy the item, did they click on a link. We have also looked at classification systems that use attributes like height, weight, how people voted on a particular bill. In all these cases the information in the datasets can easily be represented in a table.

age	glucose level	blood pressure	diabetes?
26	78	50	1
56	111	72	1
23	81	78	0

mpg	cylinders	HP	sec. 0-60
30	4	68	19.5
45	4	48	21.7
20	8	130	12

This type of data is called “structured data”—data where instances (rows in the tables above) are described by a set of attributes (for example, a row in a table might describe a car by a set of attributes including miles per gallon, the number of cylinders and so on). Unstructured data includes things like email messages, twitter messages, blog posts, and newspaper articles. These types of things (at least at first glance) do not seem to be neatly represented in a table.

For example, suppose we are interested in determining whether various movies are good or not good and we want to analyze Twitter messages:



We, as speakers of English can see that Andy Gavin likes *Gravity*, since he said “puts the thrill back in thriller” and “good acting.” We know that Debra Murphy seems not so excited about the movie since she said “save your \$\$\$.” And if someone writes “I wanna see Gravity sooo bad, we should all go see it!!!” that person probably likes the movie even though they used the word *bad*.

Suppose I am at my local food co-op and see something called Chobani Greek Yogurt. It looks interesting but is it any good? I get out my iPhone, do a google search and find the following from the blog “Woman Does Not Live on Bread Alone”:

Chobani nonfat greek yogurt.

Have you ever had greek yogurt? If not, stop reading, gather your keys (and a coat if you live in New York) and get to your local grocery. Even when nonfat and plain, greek yogurt is so thick and creamy, I feel guilty whenever I eat it. It is definitely what yogurt is MEANT to be. The plain flavor is tart and fantastic. Those who can have it, try the honey version. There's no sugar, but a bit of honey for a taste of sweetness (or add your own local honey-- local honey is good for allergies!). I must admit, even though I'm not technically supposed to have honey, if I've had a bad day, and just desperately need sweetness, I add a teaspoon of honey to my yogurt, and it's SO worth it. The fruit flavors from Chobani all have sugar in them, but fruit is simply unnecessary with this delicious yogurt. If your grocery doesn't carry the Chobani brand, Fage (pronounced Fa-yeh) is a well known, and equally delicious brand.

Now, for Greek yogurt, you will pay about 50 cents to a dollar more, and there are about 20 more calories in each serving. But it's worth it, to me, to not feel deprived and saddened over an afternoon snack!

<http://womandoesnotliveonbreadalone.blogspot.com/2009/03/sugar-free-yogurt-reviews.html>

Is that a positive or negative review for Chobani? Even based on the second sentence: *If not, stop reading, gather your keys ... and get to your local grocery store*, it seems that this will be a positive review. She describes the flavor as *fantastic* and calls the yogurt *delicious*. It seems that I should buy it and check it out. I will be right back...



An automatic system for determining positive and negative texts.



John, that looks like a positive tweet for Gravity!

Let's imagine an automatic system that can read some text and decide whether it is a positive or negative report about a product. Why would we want such a system? Suppose there is a company that sells health monitors, they might want to know about what people are saying about their products. Are what people say mostly positive or negative? They release an ad campaign for a new product. Are people favorable about the product (*Man, I sooo want this!*) or negative (*looks like crap*). Apple has a press conference to talk about the iPhone problems. Is the resulting press coverage positive? A Senate candidate delivers a major policy speech—do the political bloggers view it favorably? So an automatic system does sound useful.



So how can I create an automatic text classification system?

Let's say I want to create a system that can tell whether a person likes or dislikes various food products. We might come up with an idea of having a list of words that would provide evidence that a person likes the product and another list of words that provides evidence that the person doesn't like the product.

'Like' words:
 delicious
 tasty
 good
 love
 smooth

'Dislike' words:
 awful
 bland
 bad
 hate
 gritty

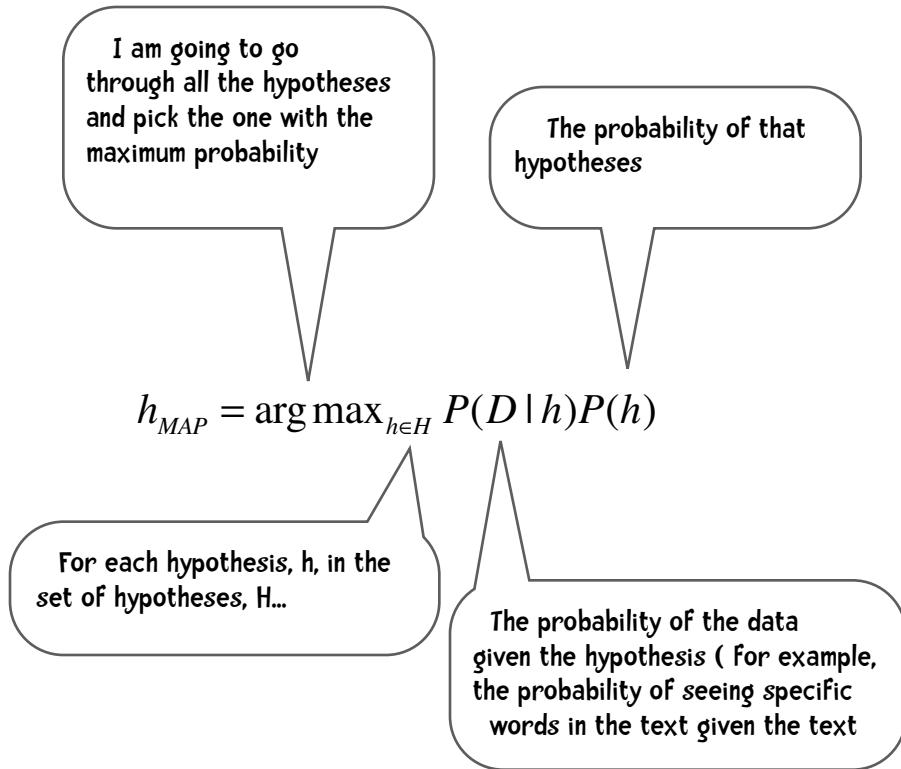
If we are trying to determine if a particular reviewer likes Chobani yogurt or not, we can just count the number of 'like' words and the number of 'dislike' words in their text. We will classify the text based on which number is higher. We can do this for other classification tasks. For example, if we want to decide whether someone is pro-choice or pro-life, we can base it on the words and phrases they use. If they use the phrase 'unborn child' then chances are they are pro-life; if they use fetus they are more likely to be pro-choice. It's not surprising that we can use the occurrence of words to classify text.



Rather than just using raw counts to classify text, let's use the naïve Bayes!!

$$h_{MAP} = \arg \max_{h \in H} P(D \mid h)P(h)$$

Let's dissect that formula!



We will use the naïve Bayes methods that were introduced in the previous chapter. We start with a training data set and, since we are now interested in unstructured text this data set is called **the training corpus**. Each entry in the corpus we will call a document even if it is a 140 character Twitter post. Each document is labeled with its class. So, for example, we might have a corpus of Twitter posts that rated movies. Each post is labeled in some way as a ‘favorable’ review or ‘unfavorable’ and we are going to train our classifier using this corpus of labeled documents. The $P(h)$ in the formula above is the probability of these labels. If we have 1,000 documents in our training corpus and 500 of them are favorable reviews and 500 unfavorable then

$$P(\text{favorable}) = 0.5$$

$$P(\text{unfavorable}) = 0.5$$



When we start with labeled training data it is called 'supervised learning.' Text classification is an example of supervised learning.

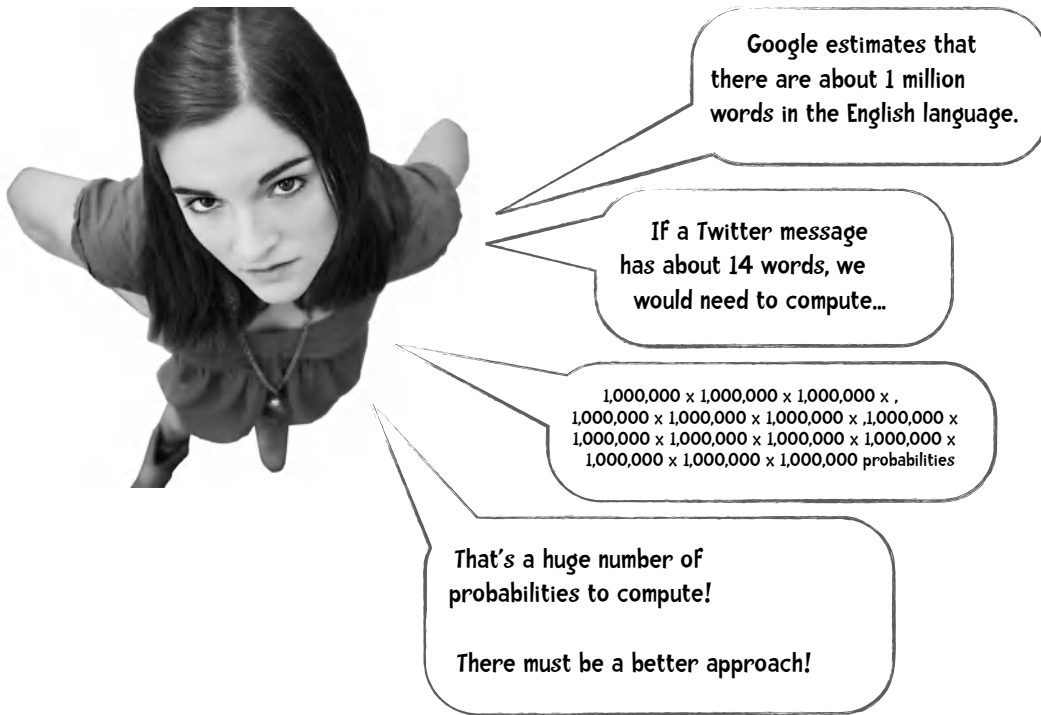
Learning from unlabeled text is called unsupervised learning. One example of unsupervised learning is clustering which we will cover in the next chapter.

There is also semi-supervised learning where the system learns from both labeled and unlabeled data. Often the system is bootstrapped using labeled data and then in subsequent learning makes use of unlabeled data.

Okay, back to

$$h_{MAP} = \arg \max_{h \in H} P(D | h)P(h)$$

Now let's examine the $P(D|h)$ part of the formula—the probability of seeing some evidence, some data D given the hypothesis h . The data D we are going to use is the words in the text. One approach would be to start with the first sentence of a document, for example, *Puts the Thrill back in Thriller*. And compute things like the probability that a 'like' document starts with the word *Puts*; what's the probability of a 'like' document having a second word of *the*; and the probability of the third word of a like document being *Thrill* and so on. And then compute the probability of a dislike document starting with the word *Puts*, the probability of the second word of a dislike document being *the* and so on.



Hmm. yeah. That is a huge number of probabilities which makes this approach unworkable. And, fortunately, there is a better approach. We are going to simplify things a bit by treating the documents as bags of unordered words. Instead of asking things like What's the probability that the third word is *thrill* given it is a 'like' document we will ask What's the probability that the word *thrill* occurs in a 'like' document. Here is how we are going to compute those probabilities.

Training Phase

First, we are going to determine the vocabulary—the unique words—of all the documents (both like and dislike documents). So, for example, even though *the* may occur thousands of times in our training corpus it only occurs once in our vocabulary. Let

$|Vocabulary|$

denote the number of words in the vocabulary. Next, for each word w_k in the vocabulary we are going to compute the probability of that word occurring given each hypothesis: $P(w_k | h_i)$.

We are going to compute this as follows. For each hypothesis (in this case 'like' and dislike')

1. combine the documents tagged with that hypothesis into one text file.
2. count how many word occurrences there are in the file. This time, if there are 500 occurrences of *the* we are going to count it 500 times. Let's call this n .
3. For each word in the vocabulary w_k , count how many times that word occurred in the text. Call this n_k
4. For each word in the vocabulary w_k , compute

$$P(w_k | h_i) = \frac{n_k + 1}{n + |\text{Vocabulary}|}$$

Naive Bayes Classification Phase

Once we have completed the training phase we can classify documents using the formula that was already presented:

$$h_{MAP} = \arg \max_{h \in H} P(D | h)P(h)$$



Let's say our training corpus consisted of 500 Twitter messages with positive reviews of movies and 500 negative. So

$$P(\textit{like}) = 0.5 \qquad P(\textit{dislike}) = 0.5$$

After training the probabilities are as follows:

word	P(word like)	P(word dislike)
am	0.007	0.009
by	0.012	0.012
good	0.002	0.0005
gravity	0.00001	0.00001
great	0.003	0.0007
hype	0.0007	0.002
I	0.01	0.01
over	0.005	0.0047
stunned	0.0009	0.002
the	0.047	0.0465

How should we classify:

I am stunned by the hype over gravity

We are going to compute

$$P(\textit{like}) \times P(I \mid \textit{like}) \times P(\textit{am} \mid \textit{like}) \times P(\textit{stunned} \mid \textit{like}) \times \dots$$

and

$$P(\textit{dislike}) \times P(I \mid \textit{dislike}) \times P(\textit{am} \mid \textit{dislike}) \times P(\textit{stunned} \mid \textit{dislike}) \times \dots$$

and chose the hypothesis associated with the highest probability.

word	P(word like)	P(word dislike)
	P(like) = 0.5	P(dislike) =0.05
I	0.01	0.01
am	0.007	0.009
stunned	0.0009	0.002
by	0.012	0.012
the	0.047	0.0465
hype	0.0007	0.002
over	0.005	0.0047
gravity	0.00001	0.00001
\prod	6.22E-22	4.72E-21

So the probabilities are

like 0.0000000000000000000000622

dislike 0.0000000000000000000004720

The probability of dislike is larger than that for like so we classify the tweet as a dislike.

Just a reminder:

That e notation means how many places to move the decimal point. If the number is positive we move the decimal to the right, negative means move it to the left. So

1.23e-1 = 0.123
 1.23e-2 = 0.0123
 1.23e-3 = 0.00123

and so on



Here's an illustration of the problem. Let's say we have a 100 word document and the average probability of each word is 0.001 (words like *tell*, *reported*, *average*, *morning*, and *am* have a probability of around 0.001). If I multiply those probabilities in Python we get zero:

```
>>> 0.0001**100
0.0
```

However, if we add the log of the probabilities we do get a non-zero value:

```
>>> import math
>>> p = 0
>>> for i in range(100):
    p += math.log(0.0001)

>>> p
-921.034037197617
```


in case you forgot ... $b^n = x$

The logarithm (or log) of a number (the x above) is the exponent (the n above) that you need to raise a base (b) to equal that number. For example, suppose the base is 10,

$\log_{10}(1000) = 3$ since 1000 equals 10^3

The base of the Python log function is the mathematical constant e. We don't really need to know about e. What is of interest to us is:

1. logs compress the scale of a number (with logs we can represent smaller numbers in Python)

For ex.,

$.0000001 \times .000005 = .000000000005$

the logs of those numbers are:

$-16.11809 + -9.90348 = -26.02157$

2. instead of multiplying the probabilities we are going to add the logs of the probabilities (as shown above).

Newsgroup Corpus

We will first investigate how this algorithm works by using a standard reference corpus of usenet newsgroup posts. The data consists of posts from 20 different newsgroups:

comp.graphics	misc.forsale	soc.religion.christian	alt.atheism
comp.os.ms-windows-misc	rec.autos	talk.politics.guns	sci.space
comp.sys.ibm.pc.hardware	rec.motorcycles	talk.politics.mideast	sci.crypt
comp.sys.mac.hardware	rec.sport.baseball	talk.politics.misc	sci.electronics
comp.windows.x	rec.sport.hockey	talk.religion.misc	sci.med

We would like to build a classifier that can correctly determine what group the post came from. For example, we would like to classify this post

From: essbaum@rchland.vnet.ibm.com
(Alexander Essbaum)
Subject: Re: Mail order response time
Disclaimer: This posting represents the poster's views, not necessarily those of IBM
Nntp-Posting-Host: relva.rchland.ibm.com
Organization: IBM Rochester
Lines: 18
> I have ordered many times from Competition
> accesories and ussually get 2-3 day delivery.

ordered 2 fork seals and 2 guide bushings from CA for my FZR. two weeks later get 2 fork seals and 1 guide bushing. call CA and ask for remaining *guide* bushing and order 2 *slide* bushings (explain on the phone which bushings are which; the guy seemed to understand). two weeks later get 2 guide bushings.

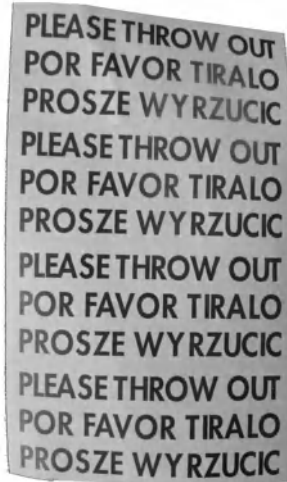
sigh

how much you wanna bet that once i get ALL the parts and take the fork apart that some parts won't fit?

as being from rec.motorcycles

Notice the misspellings (*accesories* and *ussually*). This might be challenging for a classifier!

The data is available at <http://qwone.com/~jason/20Newsgroups/> (we are using the 20news=bydate dataset) . It is also available on the website for the book, <http://guidetodatamining.com>. The data consists of 18,846 documents and is already sorted into training (60% of the data) and test sets. The training and test data are in separate directories. Within each directory are subdirectories representing each newsgroup. Within those are the separate documents representing posts to that newsgroup.



Throwing things out!

Before we start coding, let's think about this task in more detail.



Ladies and Gentlemen. On the main stage ... Just based on the words in the text, we are going to attempt to tell which newsgroup the post is from

For example, we would like to build a system that would classify the following post as being from rec.motorcycle:

I am looking at buying a Dual Sport type motorcycle. This is my first cycle as well. I am interested in any experiences people have with the following motorcycles, good or bad.

Honda XR250L
Suzuki DR350S
Suzuki DR250ES
Yamaha XT350

Most XXX vs. YYY articles I have seen in magazines pit the Honda XR650L against another cycle, and the 650 always comes out shining. Is it safe to assume that the 250 would be of equal quality ?

Let's consider which words might be helpful in the classification task:



I...	"I" is not helpful
am...	not helpful
looking...	not helpful
at...	not helpful
buying...	erm. probably helpful
a ...	not helpful
dual...	definitely helpful
sport ...	definitely
type...	probably not
motor-cycle	definitely!!!!

If we throw out the 200 most frequent words in English our document looks like this:

I am looking at buying a Dual Sport type motorcycle. This is my first cycle as well. I am interested in any experiences people have with the following motorcycles, good or bad.

Honda XR250L
Suzuki DR350S
Suzuki DR250ES
Yamaha XT350

Most XXX vs. YYY articles I have seen in magazines pit the Honda XR650L against another cycle, and the 650 always comes out shining. Is it safe to assume that the 250 would be of equal quality ?

Removing these words cuts down the size of our text by about half. Plus, it doesn't look like removing these words will have any impact on our ability to categorize texts. Indeed data miners have called such words *words without any content*, and *fluff words*. H.P. Luhn, in his seminal paper 'The automatic creation of literature abstracts' says of these words that they are “too common to have the type of significance being sought and would constitute 'noise' in the system.” That noise argument is interesting as it implies that removing these words will improve performance. These words that we remove are called 'stop words'. We have a list of such words, the 'stop word list', and remove these words from the text in a preprocessing step. We remove these words because 1) it cuts down on the amount of processing we need to do and 2) it does not negatively impact the performance of our system—as the noise argument suggests removing them might improve performance.

Common Words vs. Stop Words

While it is true that common words like 'the' and 'a' may not help us in our classification task, other common words such as 'work', 'write', and 'school' may help depending on our classification task. When we create a stop word list, we often omit common words that may be helpful. You can explore these differences by comparing stop word lists and frequent word lists found on the web.

The counter argument: the hazards of stop word removal



You whippersnapper. You shouldn't be throwing away those common words!

While removing stop words may be useful in some situations, you should not just automatically remove them without thinking. For example, it turns out just using the most frequent words and throwing out the rest (the reverse technique of the above) provides

sufficient information to identify where Arabic documents were written. (If you are curious about this check out the paper *Linguistic Dumpster Diving: Geographical Classification of Arabic Text* I co-wrote with some of my colleagues at New Mexico State University. It is available on my website <http://zacharski.org>). In looking at online chats, sexual predators use words like *I*, *me*, and *you*, much more frequently than non-predators. If your task is to identify sexual predators, removing frequent words would actually hurt your performance.



Don't blindly remove stop words.
Think First.

Coding it – Python Style

Let us first consider coding the training part of the Naïve Bayes Classifier. Recall that the training data is organized as follows:

```
20news-bydate-train
  alt.atheism
    text file 1 for alt.atheism
    text file 2
    ...
    text file n
  comp.graphics
    text file 1 for comp.graphics
    ...
```



So I have a directory (in this example called '20news-bydate-train'). Underneath this directory are subdirectories representing different classification categories (in this case alt.atheism, comp.graphics, etc). The names of these subdirectories match the category names. The test directory is organized in a similar way. So, in matching this structure, the Python code for training will need to know the training directory (for example, /Users/raz/Downloads/20news-bydate/20news-bydate-train/). The outline for the training code is as follows.

class BayesText

1. the init method:

- a. read in the words from the stoplist
- b. read the training directory to get the names of the subdirectories (in addition to being the names of the subdirectories, these are the names of the categories).
- c. For each of those subdirectories, call a method "train" that will count the occurrences of words in all the files of that subdirectory.
- d. compute the probabilities using

$$P(w_k | h_i) = \frac{n_k + 1}{n + |\text{Vocabulary}|}$$

Yet another reminder that all the code is available at
guidetodatamining.com

```

from __future__ import print_function
import os, codecs, math

class BayesText:

    def __init__(self, trainingdir, stopwordslist):
        """This class implements a naive Bayes approach to text
        classification
        trainingdir is the training data. Each subdirectory of
        trainingdir is titled with the name of the classification
        category -- those subdirectories in turn contain the text
        files for that category.
        The stopwordslist is a list of words (one per line) will be
        removed before any counting takes place.
        """
        self.vocabulary = {}
        self.prob = {}
        self.totals = {}
        self.stopwords = {}
        f = open(stopwordlist)
        for line in f:
            self.stopwords[line.strip()] = 1
        f.close()
        categories = os.listdir(trainingdir)
        #filter out files that are not directories
        self.categories = [filename for filename in categories
                           if os.path.isdir(trainingdir + filename)]
        print("Counting ...")
        for category in self.categories:
            print('    ' + category)
            (self.prob[category],
             self.totals[category]) = self.train(trainingdir, category)
        # I am going to eliminate any word in the vocabulary
        # that doesn't occur at least 3 times
        toDelete = []
        for word in self.vocabulary:
            if self.vocabulary[word] < 3:
                # mark word for deletion
                # can't delete now because you can't delete
                # from a list you are currently iterating over
                toDelete.append(word)

```



```

# now delete
for word in toDelete:
    del self.vocabulary[word]
# now compute probabilities
vocabLength = len(self.vocabulary)
print("Computing probabilities:")
for category in self.categories:
    print(' ' + category)
    denominator = self.totals[category] + vocabLength
    for word in self.vocabulary:
        if word in self.prob[category]:
            count = self.prob[category][word]
        else:
            count = 1
        self.prob[category][word] = (float(count + 1)
                                     / denominator)

print ("DONE TRAINING\n\n")

def train(self, trainingdir, category):
    """counts word occurrences for a particular category"""
    currentdir = trainingdir + category
    files = os.listdir(currentdir)
    counts = {}
    total = 0
    for file in files:
        #print(currentdir + '/' + file)
        f = codecs.open(currentdir + '/' + file, 'r', 'iso8859-1')
        for line in f:
            tokens = line.split()
            for token in tokens:
                # get rid of punctuation and lowercase token
                token = token.strip('\'. ,? :-')
                token = token.lower()
                if token != '' and not token in self.stopwords:
                    self.vocabulary.setdefault(token, 0)
                    self.vocabulary[token] += 1
                    counts.setdefault(token, 0)
                    counts[token] += 1
                    total += 1
            f.close()
    return(counts, total)

```

The results of the training phase are stored in a dictionary (hash table) called `prob`. The keys of the dictionary are the different classifications (`comp.graphics`, `rec.motorcycles`, `soc.religion.christian`, etc); the values are dictionaries. The keys of these subdictionaries are the words and the values are the probabilities of those words. Here is an example:

```
bT = BayesText(trainingDir, stoplistfile)
>>>bT.prob["rec.motorcycles"]["god"]
0.00013035445075435553
>>>bT.prob["soc.religion.christian"]["god"]
0.004258192391884386
>>>bT.prob["rec.motorcycles"]["the"]
0.028422937849264914
>>>bT.prob["soc.religion.christian"]["the"]
0.039953678998362795
```

So, for example, the probability of the word ‘god’ occurring in a text in the `rec.motorcycles` newsgroup is 0.00013 (or one occurrence of *god* in every 10,000 words). The probability of the word ‘god’ occurring in a text in `soc.religion.christian` is .00424 (one occurrence in every 250 words).

Training also results in a list called `categories`, which, as you might predict, is simply a list of all the categories:

```
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc',
'comp.sys.ibm.pc.hardware', ...]
```



So that is the training phase. Let us now turn to classifying a document.



code it

Can you code a method called `classify` that will predict the classification of a document? For example:

```
>>> bT.classify("20news-bydate-test/rec.motorcycles/104673")
'rec.motorcycles'
>>> bT.classify("20news-bydate-test/sci.med/59246")
'sci.med'
>>> bT.classify("20news-bydate-test/soc.religion.christian/21424")
'soc.religion.christian'
```

As you can see, the `classify` method takes a filename as an argument and returns a string denoting the classification.

A Python file you can use as a template, `bayesText-ClassifyTemplate.py`, is available on our website.



```
class BayesText:

    def __init__(self, trainingdir, stopwordlist):
        self.vocabulary = {}
        self.prob = {}
        self.totals = {}
        self.stopwords = {}
        f = open(stopwordlist)
        for line in f:
            self.stopwords[line.strip()] = 1
        f.close()
        categories = os.listdir(trainingdir)
        #filter out files that are not directories
        self.categories = [filename for filename in categories
                           if os.path.isdir(trainingdir +
filename)]
        print("Counting ...")
        for category in self.categories:
            print(' ' + category)
            (self.prob[category],
             self.totals[category]) = self.train(trainingdir,
category)
        # I am going to eliminate any word in the vocabulary
```



code it - one possible solution

```
def classify(self, filename):
    results = {}
    for category in self.categories:
        results[category] = 0
    f = codecs.open(filename, 'r', 'iso8859-1')
    for line in f:
        tokens = line.split()
        for token in tokens:
            token = token.strip('\",.,?;-').lower()
            if token in self.vocabulary:
                for category in self.categories:
                    if self.prob[category][token] == 0:
                        print("%s %s" % (category, token))
                        results[category] += math.log(
                            self.prob[category][token])
    f.close()
    results = list(results.items())
    results.sort(key=lambda tuple: tuple[1], reverse = True)
    # for debugging I can change this to give me the entire list
    return results[0][0]
```

Finally, let's have a method that classifies every document in the test directory and prints out the percent accuracy of this method.

```

def testCategory(self, directory, category):
    files = os.listdir(directory)
    total = 0
    correct = 0
    for file in files:
        total += 1
        result = self.classify(directory + file)
        if result == category:
            correct += 1
    return (correct, total)

def test(self, testdir):
    """Test all files in the test directory--that directory is
    organized into subdirectories--each subdir is a classification
    category"""
    categories = os.listdir(testdir)
    #filter out files that are not directories
    categories = [filename for filename in categories if
                   os.path.isdir(testdir + filename)]
    correct = 0
    total = 0
    for category in categories:
        (catCorrect, catTotal) = self.testCategory(
            testdir + category, category)
        correct += catCorrect
        total += catTotal
    print("Accuracy is %f%% (%i test instances)" %
          ((float(correct) / total) * 100, total))

```

When I run this code using an empty stoplist file I get:

DONE TRAINING

Running Test ...

.....

Accuracy is 77.774827% (7532 test instances)

71.77% accuracy is pretty good...
I wonder what the accuracy would be
if we used a stoplist?

Only one way to find out ...



code it

Can you run the classifier with a few stop word lists? Does performance improve? Which is most accurate? (You will need to search the web to find these lists)

stop list size	accuracy
0	71.774827
list 1	
list 2	



code it - some results

I found a 25 word stop word list at: <http://nlp.stanford.edu/IR-book/html/htmledition/dropping-common-terms-stop-words-1.html>

And a 174 word one at <http://www.ranks.nl/resources/stopwords.html>

(these word lists are available on our website)

Here are the results:

stop list size	accuracy
0	71.714827%
25 word list	78.757302%
174 word list	79.938927%

So in this case, it looks like having a 174 word stop word list improved performance about 2% over having no stop word list? Does this match your results?

Naïve Bayes and Sentiment Analysis

The goal of sentiment analysis is to determine the writer's attitude (or opinion).



One common type of sentiment analysis is to determine the **polarity** of a review or comment (positive or negative) and we can use a Naïve Bayes Classifier for this task. We can try this out by using the polarity movie review dataset first presented in Pang and Lee 2004¹. Their dataset consists of 1,000 positive and 1,000 negative reviews. Here are some examples:

the second serial-killer thriller of the month is just awful . oh , it starts deceptively okay , with a handful of intriguing characters and some solid location work

when i first heard that romeo & juliet had been " updated " i shuddered . i thought that yet another of shakespeare's classics had been destroyed . fortunately , i was wrong . baz luhrman has directed an " in your face " , and visually

¹ Pang, Bo and Lillian Lee. 2004. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. Proceedings of ACL.

You can download the original dataset from <http://www.cs.cornell.edu/People/pabo/movie-review-data/>. I have organized the data into 10 buckets (folds) with the following directory structure:

```

review_polarity_buckets
txt_sentoken
  neg
    0      files in fold 0
    1      files in fold 1
    ...
    9      files in fold 9
  pos
    0      files in fold 0
    ...

```

This re-organized dataset is available on our website.



code it

Can you modify the Naive Bayes Classifier code to do 10-fold cross validation of the classifier on this data set. The output should look something like:

```

Classified as:
      neg   pos
+-----+-----+
neg   |  1   |  2   |
pos   |  3   |  4   |
+-----+-----+

```

12.345 percent correct
total of 2000 instances

Also compute the kappa coefficient.

Obvious Disclaimer

You won't become proficient in data mining by reading this book anymore than reading a book about piano playing will make you proficient at piano playing. You need to practice!



Woman practicing Brahms



Practice makes the heart grow fonder!

Woman practicing Naïve Bayes



code it – my results

Here are the results I got:

		Classified as:	
		neg	pos
neg	845	155	
pos	222	778	

81.150 percent correct
total of 2000 instances

Also compute the kappa coefficient.

$$\kappa = \frac{P(c) - P(r)}{1 - P(r)} = \frac{.8115 - 0.5}{1 - 0.5} = \frac{.3115}{.5} = 0.623$$

So we have good performance of the algorithm on this data.

My code is on the following page!

Yet another reminder:
The code is available for download on the book's
website <http://guidetodatamining.com/>

```

from __future__ import print_function
import os, codecs, math

class BayesText:

    def __init__(self, trainingdir, stopwordlist, ignoreBucket):
        """This class implements a naive Bayes approach to text
        classification
        trainingdir is the training data. Each subdirectory of
        trainingdir is titled with the name of the classification
        category -- those subdirectories in turn contain the text
        files for that category.
        The stopwordlist is a list of words (one per line) will be
        removed before any counting takes place.
        """
        self.vocabulary = {}
        self.prob = {}
        self.totals = {}
        self.stopwords = {}
        f = open(stopwordlist)
        for line in f:
            self.stopwords[line.strip()] = 1
        f.close()
        categories = os.listdir(trainingdir)
        #filter out files that are not directories
        self.categories = [filename for filename in categories
                           if os.path.isdir(trainingdir + filename)]
        print("Counting ...")
        for category in self.categories:
            #print('    ' + category)
            (self.prob[category],
             self.totals[category]) = self.train(trainingdir, category,
                                                  ignoreBucket)
        # I am going to eliminate any word in the vocabulary
        # that doesn't occur at least 3 times
        toDelete = []

```

```

for word in self.vocabulary:
    if self.vocabulary[word] < 3:
        # mark word for deletion
        # can't delete now because you can't delete
        # from a list you are currently iterating over
        toDelete.append(word)
# now delete
for word in toDelete:
    del self.vocabulary[word]
# now compute probabilities
vocabLength = len(self.vocabulary)
#print("Computing probabilities:")
for category in self.categories:
    #print('    ' + category)
    denominator = self.totals[category] + vocabLength
    for word in self.vocabulary:
        if word in self.prob[category]:
            count = self.prob[category][word]
        else:
            count = 1
        self.prob[category][word] = (float(count + 1)
                                     / denominator)

#print ("DONE TRAINING\n\n")

def train(self, trainingdir, category, bucketNumberToIgnore):
    """counts word occurrences for a particular category"""
    ignore = "%i" % bucketNumberToIgnore
    currentdir = trainingdir + category
    directories = os.listdir(currentdir)
    counts = {}
    total = 0
    for directory in directories:
        if directory != ignore:
            currentBucket = trainingdir + category + "/" + \
                directory
            files = os.listdir(currentBucket)
            #print("    " + currentBucket)
            for file in files:

```

```

f = codecs.open(currentBucket + '/' + file, 'r',
                 'iso8859-1')
for line in f:
    tokens = line.split()
    for token in tokens:
        # get rid of punctuation
        # and lowercase token
        token = token.strip('\'. ,?:-')
        token = token.lower()
        if token != '' and not token in \
            self.stopwords:
            self.vocabulary.setdefault(token, 0)
            self.vocabulary[token] += 1
            counts.setdefault(token, 0)
            counts[token] += 1
            total += 1
    f.close()
return(counts, total)

```

```

def classify(self, filename):
    results = {}
    for category in self.categories:
        results[category] = 0
    f = codecs.open(filename, 'r', 'iso8859-1')
    for line in f:
        tokens = line.split()
        for token in tokens:
            #print(token)
            token = token.strip('\'. ,?:-').lower()
            if token in self.vocabulary:
                for category in self.categories:
                    if self.prob[category][token] == 0:
                        print("%s %s" % (category, token))
                        results[category] += math.log(
                            self.prob[category][token])
    f.close()
    results = list(results.items())
    results.sort(key=lambda tuple: tuple[1], reverse = True)

```

```

# for debugging I can change this to give me the entire list
return results[0][0]

def testCategory(self, direc, category, bucketNumber):
    results = {}
    directory = direc + ("%i/" % bucketNumber)
    #print("Testing " + directory)
    files = os.listdir(directory)
    total = 0
    correct = 0
    for file in files:
        total += 1
        result = self.classify(directory + file)
        results.setdefault(result, 0)
        results[result] += 1
        #if result == category:
        #    correct += 1
    return results

def test(self, testdir, bucketNumber):
    """Test all files in the test directory--that directory is
    organized into subdirectories--each subdir is a classification
    category"""
    results = {}
    categories = os.listdir(testdir)
    #filter out files that are not directories
    categories = [filename for filename in categories if
                  os.path.isdir(testdir + filename)]
    correct = 0
    total = 0
    for category in categories:
        #print(".", end="")
        results[category] = self.testCategory(
            testdir + category + '/', category, bucketNumber)
    return results

def tenfold(dataPrefix, stoplist):
    results = {}
    for i in range(0,10):

```

```

bT = BayesText(dataPrefix, stoplist, i)
r = bT.test(theDir, i)
for (key, value) in r.items():
    results.setdefault(key, {})
    for (ckey, cvalue) in value.items():
        results[key].setdefault(ckey, 0)
        results[key][ckey] += cvalue
    categories = list(results.keys())
categories.sort()
print( "\n      Classified as: ")
header = "      "
subheader = "      +"
for category in categories:
    header += "% 2s " % category
    subheader += "-----+"
print (header)
print (subheader)
total = 0.0
correct = 0.0
for category in categories:
    row = " %s |" % category
    for c2 in categories:
        if c2 in results[category]:
            count = results[category][c2]
        else:
            count = 0
        row += " %3i |" % count
        total += count
        if c2 == category:
            correct += count
    print(row)
print(subheader)
print("\n%5.3f percent correct" %((correct * 100) / total))
print("total of %i instances" % total)

```

change these to match your directory structure

```

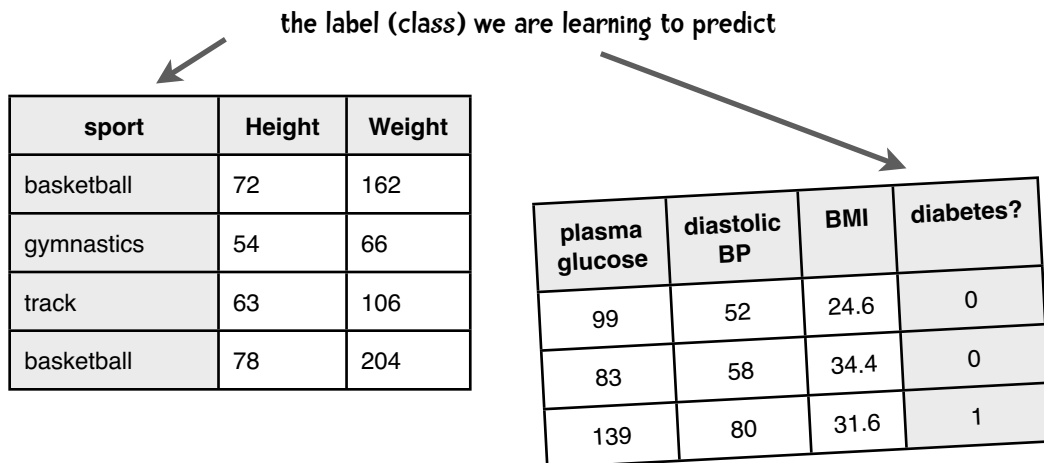
theDir = "/Users/raz/Downloads/review_polarity_buckets/txt_sentoken/"
stoplistfile = "/Users/raz/Downloads/20news-bydate/stopwords25.txt"
tenfold(theDir, stoplistfile)

```


Chapter 8: Clustering

Discovering Groups

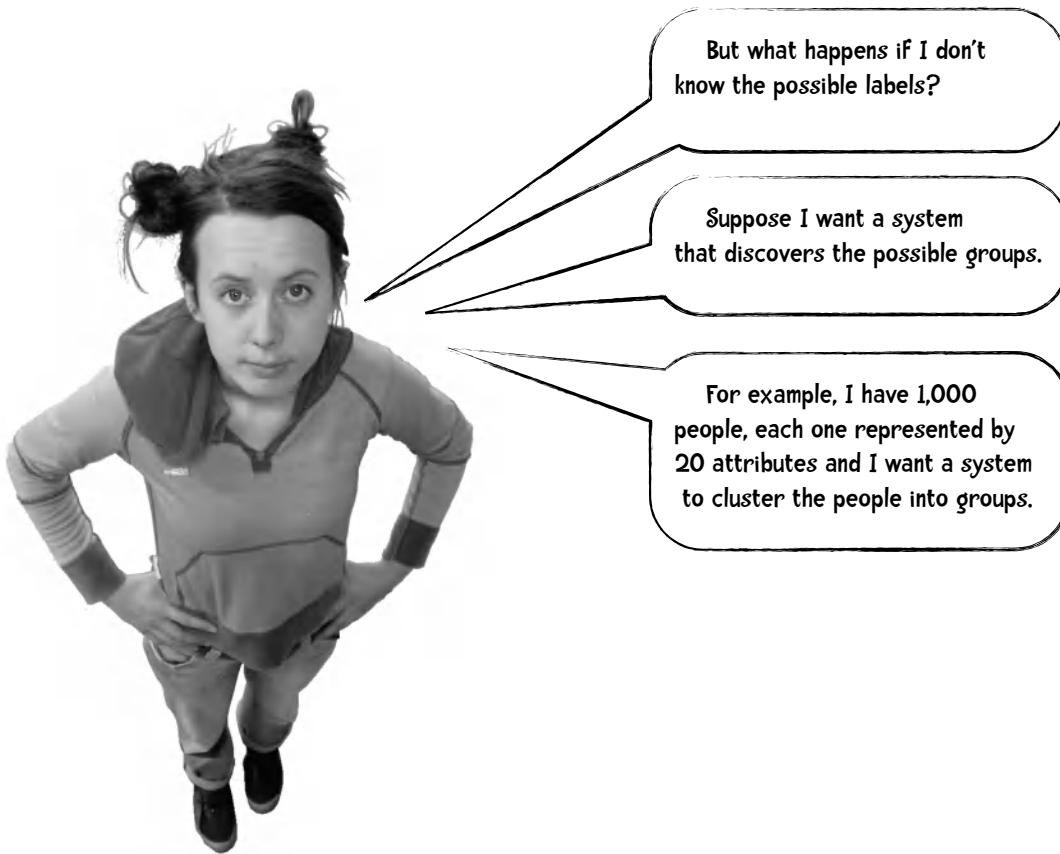
In previous chapters we have been developing classification systems. In these systems we train a classifier on a set of labeled examples.



After we train the classifier, we can use it to label new examples.

*This person looks like a basketball player. That one a gymnast.
That person is unlikely to get diabetes in 3 years.*

and so on. In other words, the classifier selects a label from a set of labels it acquired during the training phase—it knows the possible labels.



This task is called clustering. The system divides a set of instances into clusters or groups based on some measure of similarity. There are two main types of clustering algorithms.

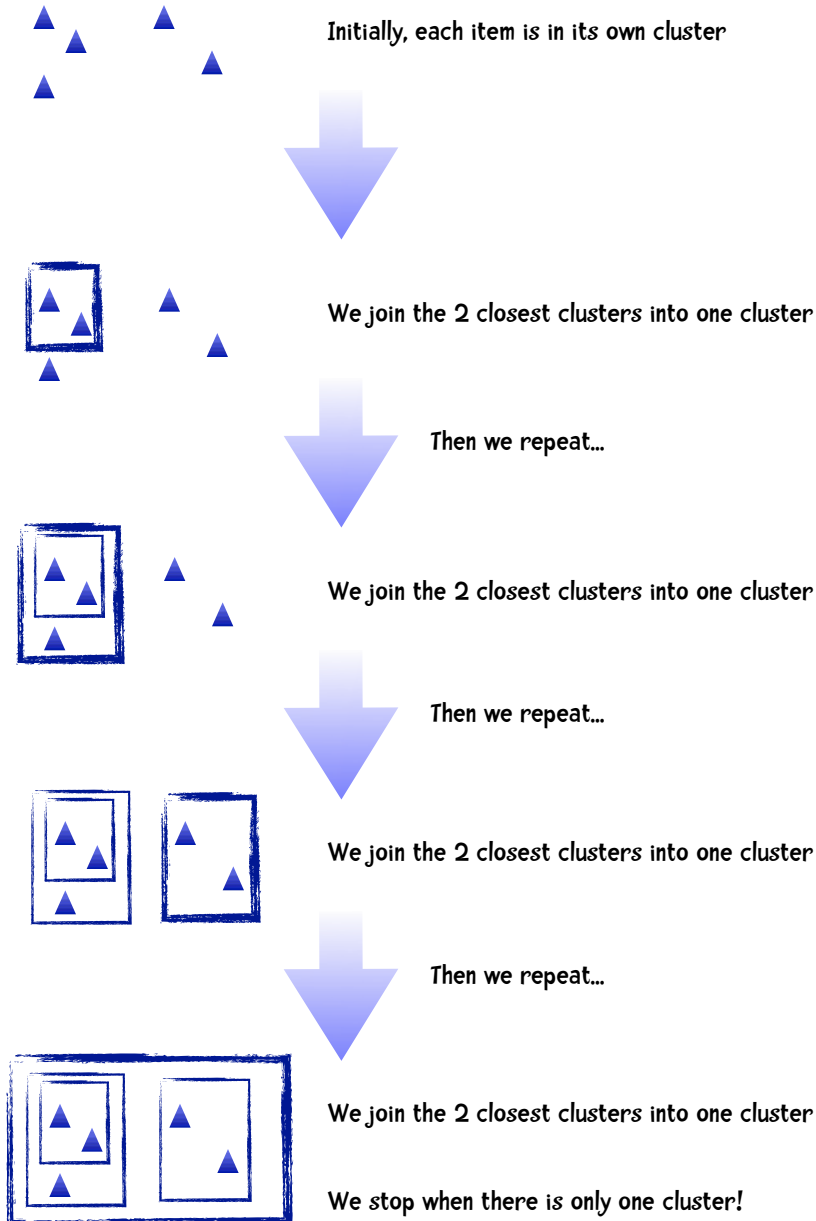
k-means clustering

For one type, we tell the algorithm how many clusters to make. *Please cluster these 1,000 people into 5 groups. Please classify these web pages into 15 groups.* These methods go by the name of k-means clustering algorithms and we will discuss those a bit later in the chapter.

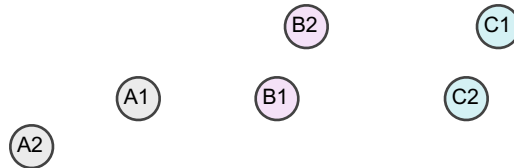
hierarchical clustering

For the other approach we don't specify how many clusters to make. Instead the algorithm starts with each instance in its own cluster. At each iteration of the algorithm it combines the two most similar clusters into one. It repeatedly does this until there is only one cluster. This

is called hierarchical clustering and its name makes sense. The running of the algorithm results in one cluster, which consists of two sub-clusters. Each of those two sub-clusters in turn, consist of 2 sub-sub clusters and so on.



Again, at each iteration of the algorithm we join the two closest clusters. To determine the 'closest clusters' we use a distance formula. But we have some choices in how we compute the distance between two clusters, which leads to different clustering methods. Consider the three clusters (A, B, and C) illustrated below each containing two members. Which pair of clusters should we join? Cluster A with B, or cluster C with B?



Single-linkage clustering

In single-linkage clustering we define the distance between two clusters as the **shortest** distance between any member of one cluster to any member of the other. With this definition, the distance between Cluster A and Cluster B is the distance between A1 and B1, since that is shorter than the distances between A1 and B2, A2 and B1, and A2 and B2. With single-linkage clustering, Cluster A is closer to Cluster B than C is to B, so we would combine A and B into a new cluster.

Complete-linkage clustering

In complete-linkage clustering we define the distance between two clusters as the **greatest** distance between any member of one cluster to any member of the other. With this definition, the distance between Cluster A and Cluster B is the distance between A2 and B2. With complete-linkage clustering, Cluster C is closer to Cluster B than A is to B, so we would combine B and C into a new cluster.

Average-linkage clustering

In average-linkage clustering we define the distance between two clusters as the **average** distance between any member of one cluster to any member of the other. In the diagram above, it appears that the average distance between Clusters C and B would be less than the average between A and B and we would combine B and C into a new cluster.

Hey! Let's work through an example of single-linkage clustering!



Good idea! Let's practice by clustering dog breeds based on height and weight!

breed	height (inches)	weight (pounds)
Border Collie	20	45
Boston Terrier	16	20
Brittany Spaniel	18	35
Bullmastiff	27	120
Chihuahua	8	8
German Shepherd	25	78
Golden Retriever	23	70
Great Dane	32	160
Portuguese Water Dog	21	50
Standard Poodle	19	65
Yorkshire Terrier	6	7

Psst! I think we are forgetting something. Isn't there something we should do before computing distance?



Normalization!

Let's
change those numbers to Modified
Standard Scores



Modified Standard Scores

breed	height	weight
Border Collie	0	-0.1455
Boston Terrier	-0.7213	-0.873
Brittany Spaniel	-0.3607	-0.4365
Bullmastiff	1.2623	2.03704
Chihuahua	-2.1639	-1.2222
German Shepherd	0.9016	0.81481
Golden Retriever	0.541	0.58201
Great Dane	2.16393	3.20106
Portuguese Water Dog	0.1803	0
Standard Poodle	-0.1803	0.43651
Yorkshire Terrier	-2.525	-1.25132

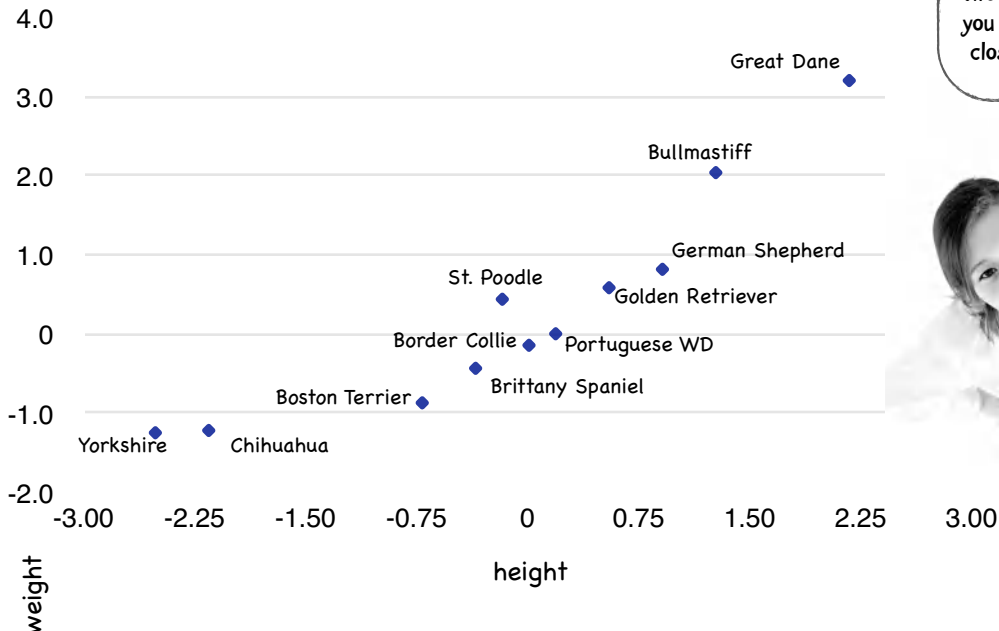


Next we are going to compute the
Euclidean distance between
breeds!

CLUSTERING

Euclidean Distances (a few of the shortest distances are highlighted):

	BT	BS	B	C	GS	GR	GD	PWD	SP	YT
Border Collie	1.024	0.463	2.521	2.417	1.317	0.907	3.985	0.232	0.609	2.756
Boston Terrier		0.566	3.522	1.484	2.342	1.926	4.992	1.255	1.417	1.843
Brittany Spaniel			2.959	1.967	1.777	1.360	4.428	0.695	0.891	2.312
Bullmastiff				4.729	1.274	1.624	1.472	2.307	2.155	5.015
Chihuahua					3.681	3.251	6.188	2.644	2.586	0.362
German Shphrd						0.429	2.700	1.088	1.146	4.001
Golden Retriever							3.081	0.685	0.736	3,572
Great Dane								3.766	3.625	6.466
Portuguese WD									0.566	2.980
Standard Poodle										2.889



Based on this chart, which two breeds do you think are the closest?

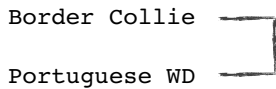


If you said Border Collie and Portuguese Water Dog, you would be correct!

The algorithm.

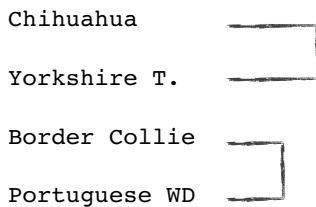
Step 1.

Initially, each breed is in its own cluster. We find the two closest clusters and combine them into one cluster. From the table on the preceding page we see that the closest clusters are the Border Collie and the Portuguese Water Dog (distance of 0.232) so we combine them.



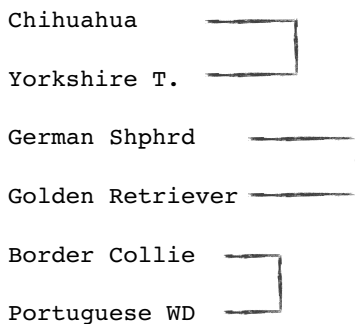
Step 2.

We find the two closest clusters and combine them into one cluster. From the table on the preceding page we see that these are the Chihuahua and the Yorkshire Terrier (distance of 0.362) so we combine them.



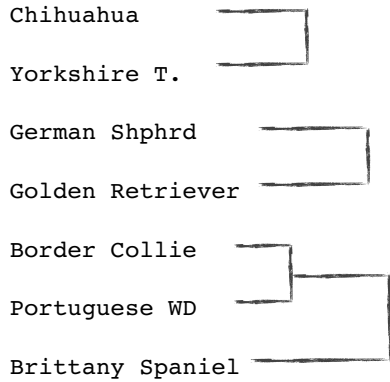
Step 3.

We repeat the process again. This time combining the German Shepherd and the Golden Retriever.



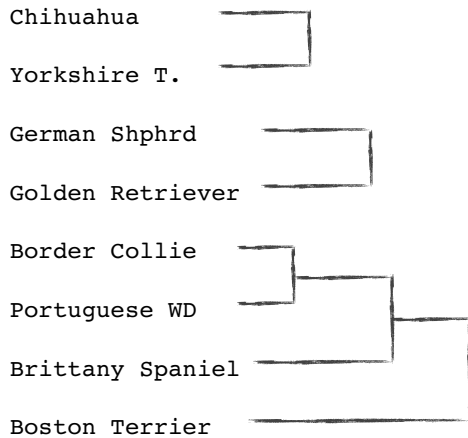
Step 4.

We repeat the process yet again. From the table we see that the next closest pair is the Border Collie and the Brittany Spaniel. The Border Collie is already in a cluster with the Portuguese Water Dog which we created in Step 1. So in this step we are going to combine that cluster with the Brittany Spaniel.



This type of diagram is called a dendrogram. It is basically a tree diagram that represents clusters.

And we continue:

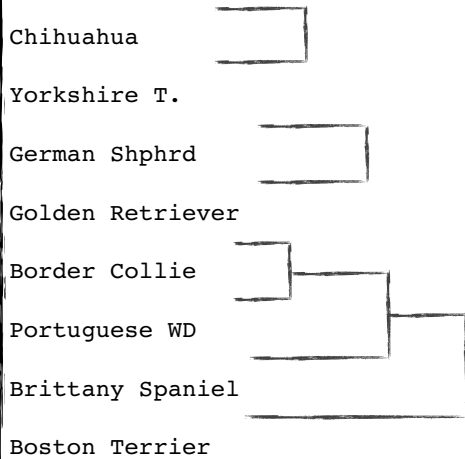




sharpen your pencil

Finish the clustering of the dog data!

To help you in this task, there is a sorted list of dog breed distances on this chapter's webpage (<https://raw.githubusercontent.com/zacharski/pg2dm-python/0684ec677a1a1baaebc47bc0f8f21ec121e83339/data/ch8/dogDistanceSorted.txt>).



coding a hierarchical clustering algorithm

For coding the clusterer we can use a priority queue!

Can you remind me what a priority queue is?

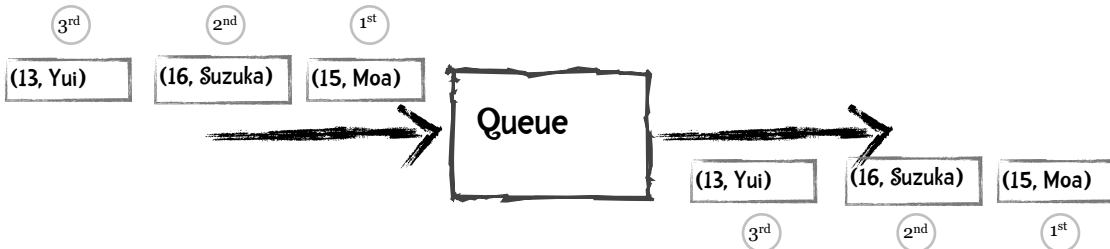


Sure!!

In a regular queue, the order in which you put the items in the queue is the order you get the items out of the queue...

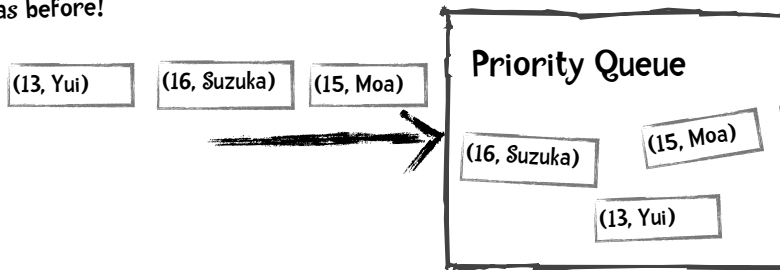


Suppose I put tuples representing a person's age and name into a queue. First the tuple for Moe is put into the queue, then the one for Suzuka and then for Yui. When I get an item from the queue, I first get the tuple for Moe since that was the first one put in the queue; then the one for Suzuka and then Yui!

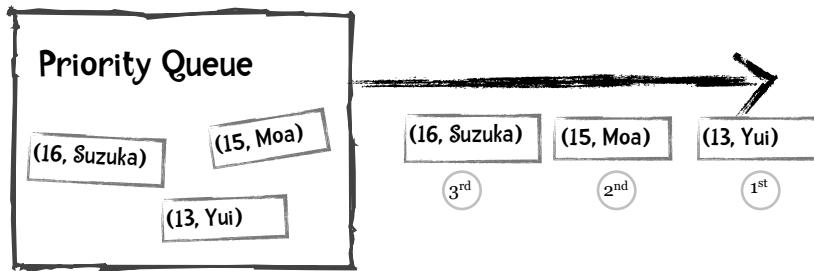


In a priority queue each item put into the queue has an associated priority. The order in which items are retrieved from the queue is based on this priority. Items with a higher priority are retrieved before items with a lower one. In our example data, suppose the younger a person is, the higher their priority.

We put the tuples into the queue in the same order as before!



The first item to be retrieved from the queue will be Yui because she is youngest and thus has the highest priority!



Let's see how this works in Python

```
>>> from queue import PriorityQueue           # load the PriorityQueue library
>>> singersQueue = PriorityQueue()         # create a PriorityQueue called
                                           # singersQueue

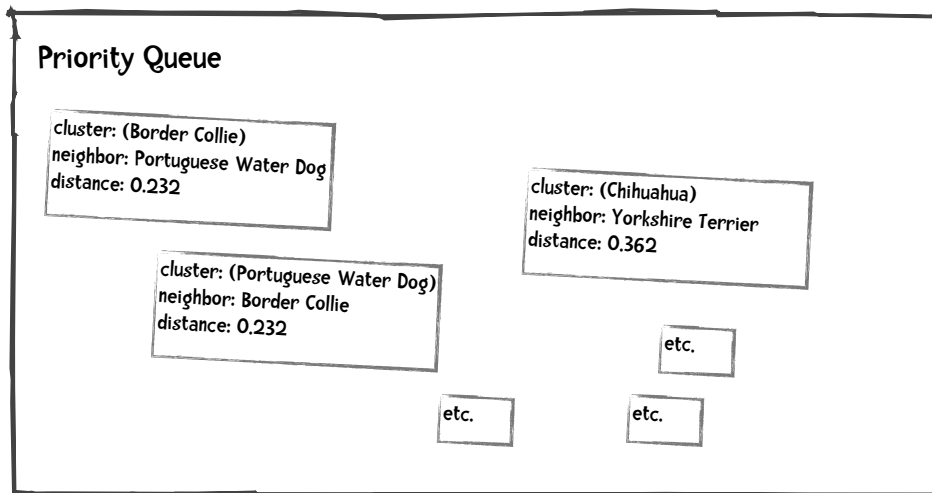
>>> singersQueue.put((16, 'Suzuka Nakamoto')) # put a few items in the queue
>>> singersQueue.put((15, 'Moa Kikuchi'))
>>> singersQueue.put((14, 'Yui Mizuno'))
```

```

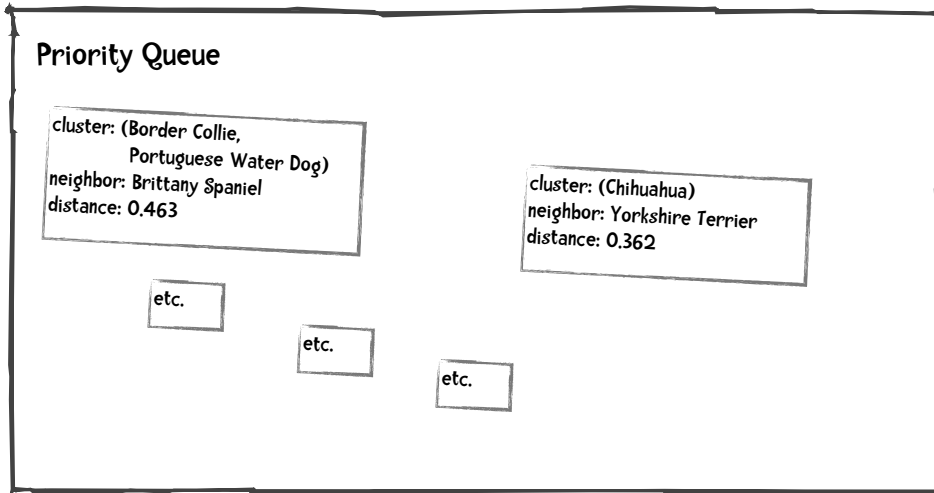
>>> singersQueue.put((17, 'Ayaka Sasaki'))
>>> singersQueue.get()                               # The first item retrieved
(14, 'Yui Mizuno')                                   # will be the youngest, Yui.
>>> singersQueue.get()
(15, 'Moa Kikuchi')
>>> singersQueue.get()
(16, 'Suzuka Nakamoto')
>>> singersQueue.get()
(17, 'Ayaka Sasaki')

```

For our task of building a hierarchical clusterer, we will put the clusters in a priority queue. The priority will be the shortest distance to a cluster's nearest neighbor. Using our dog breed example, we will put the Border Collie in our queue recording that it's nearest neighbor is the Portuguese Water Dog at a distance of 0.232. We put similar entries into the queue for the other breeds:



We will get the two entries with the shortest distance, making sure we have a matching pair. In this case we get the entries for Border Collie and Portuguese Water Dog. Next, we join the clusters into one cluster. In this case, we create a Border Collie - Portuguese Water Dog cluster. And put that cluster on the queue:



And repeat until there is only one cluster on the queue. The entries we will put on the queue need to be slightly more complex than those used in this example. So let's look at this example in more detail.

Reading the data from a file

The data will be in a CSV (comma separated values) file where the first column is the name of the instance and the rest of the columns are the values of various attributes. The first line of the file will be a header that describes these attributes:

```
breed,height (inches),weight (pounds)
Border Collie,20,45
Boston Terrier,16,20
Brittany Spaniel,18,35
Bullmastiff,27,120
Chihuahua,8,8
German Shepherd,25,78
Golden Retriever,23,70
Great Dane,32,160
Portuguese Water Dog,21,50
Standard Poodle,19,65
Yorkshire Terrier,6,7
```

The data in this file is read into a list called, not surprisingly, `data`. The list `data` saves the information by column. Thus, `data[0]` is a list containing the breed names (`data[0][0]` is the string 'Border Collie', `data[0][1]` is 'Boston Terrier' and so on). `data[1]` is a list

containing the height values, and `data[2]` is the weight list. All the data except that in the first column is converted into floats. For example, `data[1][0]` is the float 20.0 and `data[2][0]` is the float 45. Once the data is read in, it is normalized. Throughout the description of the algorithm I will use the term *index* to refer to the row number of the instance (for example, Border Collie is index 0, Boston Terrier is index 1, and Yorkshire Terrier is index 10).

Initializing the Priority Queue

At the start of the algorithm, we will put in the queue, entries for each breed. Let's consider the entry for the Border Collie. First, we calculate the distance of the Border Collie to all other breeds and put that information into a Python dictionary:

```
{1: ((0, 1), 1.0244),    the distance between the Border Collie (index 0) and the Boston Terrier
                        (index 1), is 1.0244
 2: ((0, 2), 0.463),    the distance between the Border Collie the Brittany Spaniel is 0.463
 ...
10: ((0, 10), 2.756)}  the Border Collie -- Yorkshire Terrier distance is 2.756
```

We will also keep track of the Border Collie's nearest neighbor and the distance to that nearest neighbor:

closest distance: 0.232 nearest pair: (0, 8)

The closest neighbor to the Border Collie (index 0) is the Portuguese Water Dog (index 8) and vice versa.

The problem of identical distances and what is with all those tuples.

You may have noticed that in the table on page 8-7, the distance between the Portuguese Water Dog and the Standard Poodle and the distance between the Boston Terrier and the Brittany Spaniel are the same—0.566. If we retrieve items from the priority queue based on distance there is a possibility that we will retrieve Standard Poodle and Boston Terrier and join them in a cluster, which would be an error. To prevent this error we will use a tuple containing the indices (based on the `data` list) of the two breeds that the distance represents. For example, Portuguese Water Dog is entry 8 in our data and the Standard

Poodle is entry 9, so the tuple will be (8,9). This tuple is added to the nearest neighbor list. The nearest neighbor for the poodle will be:

```
['Portuguese Water Dog', 0.566, (8,9)]
```

and the nearest neighbor for the Portuguese Water Dog will be:

```
['Standard Poodle', 0.566, (8,9)]
```

By using this tuple, when we retrieve items from the queue we can see if they are a matching pair.

Another thing to consider about identical distances.

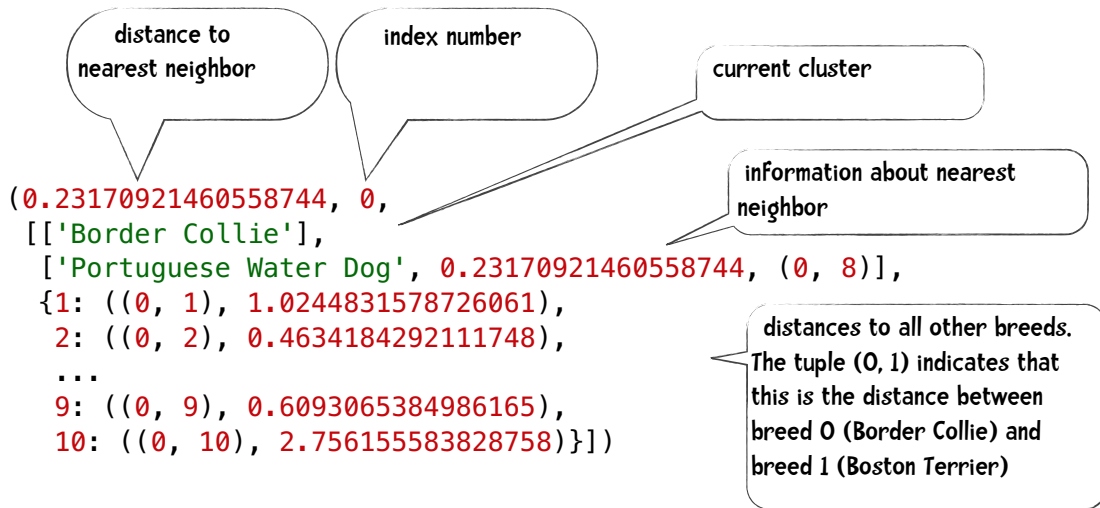
When I introduced Python Priority Queues a few pages ago, I inserted into the queue, tuples representing the ages and names of Japanese Idol performers. These entries were retrieved based on age. What happens if some of the entries have the same age (the same priority)?

Let's try:

```
>>> singersQueue.put((15, 'Suzuka Nakamoto'))
>>> singersQueue.put((15, 'Moa Kikuchi'))
>>> singersQueue.put((15, 'Yui Mizuno'))
>>> singersQueue.put((15, 'Avaka Sasaki'))
>>> singersQueue.put((12, 'Megumi Okada'))
>>> singersQueue.get()
(12, 'Megumi Okada')
>>> singersQueue.get()
(15, 'Avaka Sasaki')
>>> singersQueue.get()
(15, 'Moa Kikuchi')
>>> singersQueue.get()
(15, 'Suzuka Nakamoto')
>>> singersQueue.get()
(15, 'Yui Mizuno')
>>>
```

You can see that if the first items in the tuples match, Python uses the next item to break the tie. In the case of all those 15 year olds, the entries are retrieved based on the next item, the person's name. And since these are strings, they are ordered alphabetically. Thus the entry for Avaka Sasaki is retrieved before Moa Kikuchi and Moa is retrieved before Suzuka, which is retrieved before Yui.

In our case of hierarchical clustering, We use the distance between breeds as the primary priority. To resolve ties we will use an index number. The first element we put on the queue will have an index of 0, the second element an index of 1, the third , 2, and so on. Our complete entry we add to the queue will be of the form:



We initialize the priority queue by placing on the queue, an entry like this for each breed.

Repeat the following until there is only one cluster.

We get two items from the queue, merge them into one cluster and put that entry on the queue. In our dog breed example, we get the entry for Border Collie and the entry for Portuguese Water Dog. We create the queue

```
['Border Collie', 'Portuguese Water Dog']
```

Next we compute the distance of this new cluster to all the other dog breeds except those in the new cluster. We do this by merging the distance dictionaries of the two initial clusters in the following way. Let's call the distance dictionary of the first item we get from the queue `distanceDict1`, the distance dictionary of the second item we get from the queue `distanceDict2`, and the distance dictionary we are constructing for the new cluster `newDistanceDict`.

```

Initialize newDistanceDict to an empty dictionary
for each key, value pair in distanceDict1:
    if there is an entry in distanceDict2 with that key:
        if the distance for that entry in distanceDict1 is
           shorter than that in distanceDict2:
            place the distanceDict1 entry in newDistanceDict
        else:
            place the distanceDict1 entry in newDistanceDict

```

key	value in the Border Collie Distance List	value in the Portuguese Water Dog Distance List	value in the Distance List For the new cluster
0	–	((0, 8), 0.2317092146055)	–
1	((0, 1), 1.02448315787260)	((1, 8), 1.25503395239308)	((0, 1), 1.02448315787260)
2	((0, 2), 0.46341842921117)	((2, 8), 0.69512764381676)	(0, 2), 0.46341842921117)
3	((0, 3), 2.52128307411504)	((3, 8), 2.3065500082408)	((3, 8), 2.3065500082408)
4	((0, 4), 2.41700998092941)	((4, 8), 2.643745991701)	((0, 4), 2.41700998092941)
5	((0, 5), 1.3172590972761)	((5, 8), 1.088215707936)	((5, 8), 1.088215707936)
6	((0, 6), 0.90660838225252)	((6, 8), 0.684696194462)	((6, 8), 0.684696194462)
7	((0, 7), 3.98523295438990)	((7, 8), 3.765829069545)	((7, 8), 3.765829069545)
8	((0, 8), 0.23170921460558)	–	–
9	((0, 9), 0.60930653849861)	((8, 9), 0.566225873458)	((8, 9), 0.566225873458)
10	((0, 10), 2.7561555838287)	((8, 10), 2.980333906137)	((0, 10), 2.7561555838287)

The complete entry that will be placed on the queue as a result of merging the Border Collie and the Portuguese Water Dog will be

```

(0.4634184292111748, 11, [('Border Collie', 'Portuguese Water Dog'),
 [2, 0.4634184292111748, (0, 2)],
 {1: ((0, 1), 1.0244831578726061), 2: ((0, 2), 0.4634184292111748),
 3: ((3, 8), 2.306550008240866), 4: ((0, 4), 2.4170099809294157),
 5: ((5, 8), 1.0882157079364436), 6: ((6, 8), 0.6846961944627522),
 7: ((7, 8), 3.7658290695451373), 9: ((8, 9), 0.5662258734585477),
 10: ((0, 10), 2.756155583828758)}]])

```



Code It

Can you implement the algorithm presented above in Python?

To help you in this task, there is a Python file on the book's website, `hierarchicalClustererTemplate.py` (<http://guidetodatamining.com/guide/pg2dm-python/ch8/hierarchicalClustererTemplate.py>) that gives you a starting point. You need to:

1. Finish the `init` method.

For each entry in the data:

1. compute the Euclidean Distance from that entry to all other entries and create a Python Dictionary as described above.
2. Find the nearest neighbor
3. Place the info for this entry on the queue.

2. Write a `cluster` method. This method should repeatedly:

1. retrieve the top 2 entries on the queue
2. merge them
3. place the new cluster on the queue

until there is only one cluster on the queue.





Code It - solution

Remember:
This is only my solution and not necessarily the best solution. You might have come up with a better one!

```

from queue import PriorityQueue
import math

"""
Example code for hierarchical clustering
"""

def getMedian(alist):
    """get median value of list alist"""
    tmp = list(alist)
    tmp.sort()
    alen = len(tmp)
    if (alen % 2) == 1:
        return tmp[alen // 2]
    else:
        return (tmp[alen // 2] + tmp[(alen // 2) - 1]) / 2

def normalizeColumn(column):
    """Normalize column using Modified Standard Score"""
    median = getMedian(column)
    asd = sum([abs(x - median) for x in column]) / len(column)
    result = [(x - median) / asd for x in column]
    return result

class hClusterer:
    """ this clusterer assumes that the first column of the data is a label
    not used in the clustering. The other columns contain numeric data"""

    def __init__(self, filename):
        file = open(filename)
        self.data = {}
        self.counter = 0
        self.queue = PriorityQueue()
        lines = file.readlines()

```

```

file.close()
header = lines[0].split(',')
self.cols = len(header)
self.data = [[] for i in range(len(header))]
for line in lines[1:]:
    cells = line.split(',')
    toggle = 0
    for cell in range(self.cols):
        if toggle == 0:
            self.data[cell].append(cells[cell])
            toggle = 1
        else:
            self.data[cell].append(float(cells[cell]))
# now normalize number columns (that is, skip the first column)
for i in range(1, self.cols):
    self.data[i] = normalizeColumn(self.data[i])

###
### I have read in the data and normalized the
### columns. Now for each element i in the data, I am going to
### 1. compute the Euclidean Distance from element i to all the
### other elements. This data will be placed in neighbors,
### which is a Python dictionary. Let's say i = 1, and I am
### computing the distance to the neighbor j and let's say j
### is 2. The neighbors dictionary for i will look like
### {2: ((1,2), 1.23), 3: ((1, 3), 2.3)... }
###
### 2. find the closest neighbor
###
### 3. place the element on a priority queue, called simply queue,
### based on the distance to the nearest neighbor (and a counter
### used to break ties.

# now push distances on queue
rows = len(self.data[0])

for i in range(rows):
    minDistance = 99999
    nearestNeighbor = 0
    neighbors = {}
    for j in range(rows):
        if i != j:
            dist = self.distance(i, j)
            if i < j:
                pair = (i,j)
            else:
                pair = (j,i)
            neighbors[j] = (pair, dist)

```

```

        if dist < minDistance:
            minDistance = dist
            nearestNeighbor = j
            nearestNum = j
    # create nearest Pair
    if i < nearestNeighbor:
        nearestPair = (i, nearestNeighbor)
    else:
        nearestPair = (nearestNeighbor, i)

    # put instance on priority queue
    self.queue.put((minDistance, self.counter,
                   [[self.data[0][i]], nearestPair, neighbors]))
    self.counter += 1

def distance(self, i, j):
    sumSquares = 0
    for k in range(1, self.cols):
        sumSquares += (self.data[k][i] - self.data[k][j])**2
    return math.sqrt(sumSquares)

def cluster(self):
    done = False
    while not done:
        topOne = self.queue.get()
        nearestPair = topOne[2][1]
        if not self.queue.empty():
            nextOne = self.queue.get()
            nearPair = nextOne[2][1]
            tmp = []
            ##
            ## I have just popped two elements off the queue,
            ## topOne and nextOne. I need to check whether nextOne
            ## is topOne's nearest neighbor and vice versa.
            ## If not, I will pop another element off the queue
            ## until I find topOne's nearest neighbor. That is what
            ## this while loop does.
            ##
            while nearPair != nearestPair:
                tmp.append((nextOne[0], self.counter, nextOne[2]))
                self.counter += 1
                nextOne = self.queue.get()
                nearPair = nextOne[2][1]
            ##
            ## this for loop pushes the elements I popped off in the
            ## above while loop.
            ##

```



```

for item in tmp:
    self.queue.put(item)

if len(topOne[2][0]) == 1:
    item1 = topOne[2][0][0]
else:
    item1 = topOne[2][0]
if len(nextOne[2][0]) == 1:
    item2 = nextOne[2][0][0]
else:
    item2 = nextOne[2][0]
## curCluster is, perhaps obviously, the new cluster
## which combines cluster item1 with cluster item2.
curCluster = (item1, item2)

## Now I am doing two things. First, finding the nearest
## neighbor to this new cluster. Second, building a new
## neighbors list by merging the neighbors lists of item1
## and item2. If the distance between item1 and element 23
## is 2 and the distance between item2 and element 23 is 4
## the distance between element 23 and the new cluster will
## be 2 (i.e., the shortest distance).
##

minDistance = 99999
nearestPair = ()
nearestNeighbor = ''
merged = {}
nNeighbors = nextOne[2][2]
for (key, value) in topOne[2][2].items():
    if key in nNeighbors:
        if nNeighbors[key][1] < value[1]:
            dist = nNeighbors[key]
        else:
            dist = value
        if dist[1] < minDistance:
            minDistance = dist[1]
            nearestPair = dist[0]
            nearestNeighbor = key
        merged[key] = dist

if merged == {}:
    return curCluster
else:
    self.queue.put( (minDistance, self.counter,
                    [curCluster, nearestPair, merged]))
    self.counter += 1

```

```

def printDendrogram(T, sep=3):
    """Print dendrogram of a binary tree. Each tree node is represented by a
    length-2 tuple. printDendrogram is written and provided by David Eppstein
    2002. Accessed on 14 April 2014:
    http://code.activestate.com/recipes/139422-dendrogram-drawing/ """

    def isPair(T):
        return type(T) == tuple and len(T) == 2

    def maxHeight(T):
        if isPair(T):
            h = max(maxHeight(T[0]), maxHeight(T[1]))
        else:
            h = len(str(T))
        return h + sep

    activeLevels = {}

    def traverse(T, h, isFirst):
        if isPair(T):
            traverse(T[0], h-sep, 1)
            s = [' ']*(h-sep)
            s.append('|')
        else:
            s = list(str(T))
            s.append(' ')

        while len(s) < h:
            s.append('-')

        if (isFirst >= 0):
            s.append('+')
            if isFirst:
                activeLevels[h] = 1
            else:
                del activeLevels[h]

    A = list(activeLevels)
    A.sort()
    for L in A:
        if len(s) < L:
            while len(s) < L:
                s.append(' ')
            s.append('|')

    print (''.join(s))

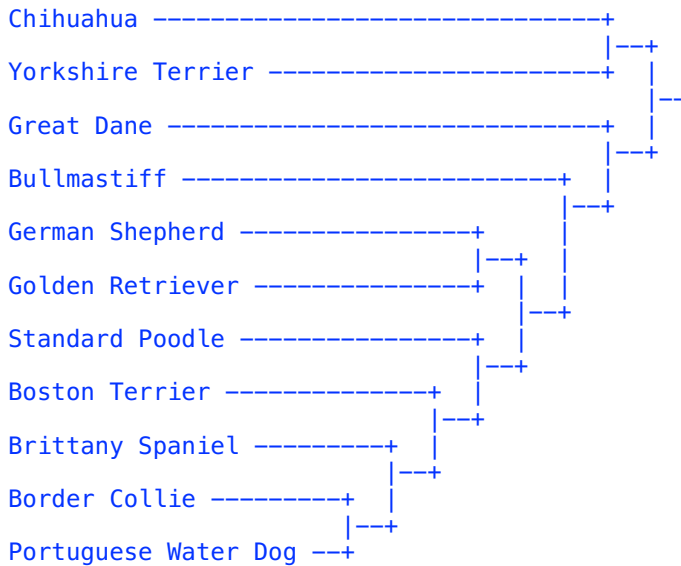
    if isPair(T):

```

```
    traverse(T[1], h-sep, 0)
traverse(T, maxHeight(T), -1)
```

```
filename = '//Users/raz/Dropbox/guide/pg2dm-python/ch8/dogs.csv'
n
hg = hClusterer(filename)
cluster = hg.cluster()
printDendrogram(cluster)
```

When I run this code I get the following results:



which match the results we computed by hand. That's encouraging.



you try!

Breakfast Cereals



On the book's website, there is a file containing nutritional information about 77 breakfast cereals including

- cereal name
- calories per serving
- protein (in grams)
- fat (in grams)
- sodium (in mg)
- fiber (grams)
- carbohydrates (grams)
- sugars (grams)
- potassium (mg)
- vitamins (% of RDA)

Nutrition Facts	
Serving Size: 1/2 cup (27g)	
Servings Per Container: about 12	
Amount Per Serving	
	with 1% cheerios
Calories	100 140
Calories from Fat 15 15	
% Daily Value*	
Total Fat 1.5g	2% 2%
Saturated Fat 0g	0% 0%
Trans Fat 0g	
Polysaturated Fat 0.5g	
Monounsaturated Fat 0.5g	
Cholesterol 0mg	0% 1%
Sodium 150mg	6% 8%
Potassium 60mg	2% 7%
Total Carbohydrate 23g	8% 10%
Dietary Fiber 2g	6% 6%
Sugars 9g	
Other Carbohydrate 12g	
Protein 1g	

Can you perform hierarchical clustering of this data?

Which cereal is most similar to Trix?

To Muesli Raisins & Almonds?

This data set is from Carnegie Mellon University:
<http://lib.stat.cmu.edu/DASL/DataFiles/Cereals.html>



you try - results

To run the clusterer on this dataset we only needed to change the filename from `dogs.csv` to `cereal.csv`. Here is an abbreviated version of the results:

```
Mueslix Crispy Blend -----+
Muesli Raisins & Almonds -----+ |---+
Muesli Peaches & Pecans -----+ |---+
...
Lucky Charms -----+
Fruity Pebbles ---+ |---+
Trix -----+ |---+
Cocoa Puffs -----+ |---+
Count Chocula ---+ |---+
```

Trix, is most similar to Fruity Pebbles. (I recommend you confirm this by running out right now and buying a box of each.) Perhaps not surprisingly, Muesli Raisins & Almonds is closest to Muesli Peaches & Pecans.



That's it for hierarchical clustering!

That was pretty easy!

Introducing ...

k-means clustering

With k-means clustering we specify how many clusters to make. This is the 'k'. If we want to make 2 groups $k = 2$, if we want to make 100, $k=100$.

k-means clustering is The Most Popular clustering algorithm!

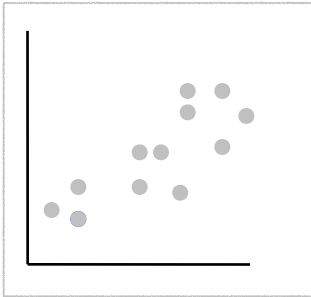
K-means is cool!



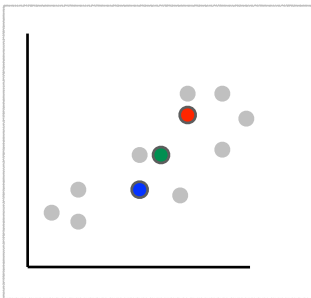
The algorithm is over 50 years old! It was first proposed by Dr. Stuart Lloyd of Bell Labs in 1957.

Here is what you need to know about k-means



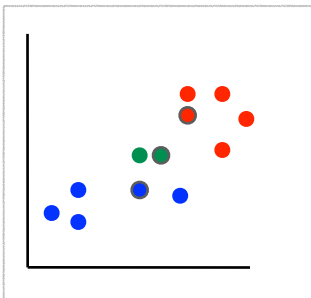


Here are some instances we want to cluster into 3 groups ($k=3$). Suppose they are dog breeds as mentioned earlier and the dimensions are height and weight.

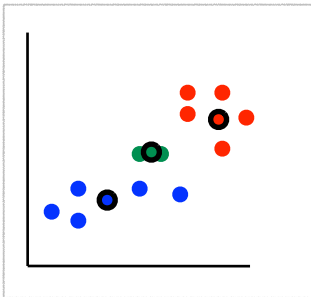


Because $k=3$, we pick 3 random points as the initial centroids of each cluster ('initial centroid' means the initial center or mean of the cluster).

Right then. We've indicated these initial centroids as red, green, and blue circles.



Okay. Next, we are going to assign each instance to the nearest centroid. The points assigned to each centroid are a cluster. So we have created k initial clusters!!



Now, for each cluster, we compute the mean (average) point of that cluster. This will be our updated centroid.



And repeat (assign each instance to the centroid & recompute centroids) until the centroids don't move much or we have reached some maximum number of iterations.

The basic k-means algorithm is:

1. select k random instances to be the initial centroids
2. REPEAT
3. assign each instance to the nearest centroid. (forming k clusters)
4. update centroids by computing mean of each cluster
5. UNTIL centroids don't change (much).

Let's go through an example. Consider the following points (x and y coordinates):

(1, 2)
(1, 4)
(2, 2)
(2, 3)
(4, 2)
(4, 4)
(5, 1)
(5, 3)

Say we want to cluster these into 2 groups.

step 1 of above algorithm: select k random instances to be initial centroids.

Suppose we randomly select (1, 4) as centroid 1 and (4, 2) as centroid 2.

step 3: assign each instance to the nearest centroid

To assign each instance to the nearest centroid we can use any of the distance measures we have previously discussed. To keep things simple, for this example let's use Manhattan Distance.

point	distance from centroid 1 (1, 4)	distance from centroid 2 (4, 2)
(1, 2)	2	3
(1,4)	0	5
(2, 2)	3	2
(2, 3)	2	3
(4, 2)	5	0
(4, 4)	3	2
(5, 1)	7	2
(5, 3)	5	2

Based on these distances we assign the points to the following clusters:

<p>CLUSTER 1 (1, 2) (1, 4) (2, 3)</p>
--

<p>CLUSTER 2 (2, 2) (4, 2) (4, 4) (5, 1) (5, 3)</p>
--

step 4: update centroids

We compute the new centroids by computing the mean of each cluster. The mean x coordinate of cluster 1 is:

$$(1 + 1 + 2) / 3 = 4/3 = 1.33$$

and the mean y is

$$(2 + 4 + 3) / 3 = 9/3 = 3$$

So the new cluster 1 centroid is (1.33, 3).

The new centroid for cluster 2 is (4, 2.4)

step 5: until centroids don't change

The old centroids were (1, 4) and (4, 2) and the new ones are (1.33, 3) and (4, 2.4). The centroids changed so we repeat.

step 3: assign each instance to the nearest centroid

Again we compute Manhattan Distance.

point	distance from centroid 1 (1.33, 3)	distance from centroid 2 (4, 2.4)
(1, 2)	1.33	3.4
(1, 4)	1.33	4.6
(2, 2)	1.67	2.4
(2, 3)	0.67	2.6
(4, 2)	3.67	0.4
(4, 4)	3.67	1.6
(5, 1)	5.67	2.4
(5, 3)	3.67	1.6

and based on these distances assign the points to clusters:

<p>CLUSTER 1 (1, 2) (1, 4) (2, 2) (2, 3)</p>
--

<p>CLUSTER 2 (4, 2) (4, 4) (5, 1) (5, 3)</p>
--

step 4: update centroids

We compute the new centroids by computing the mean of each cluster.

Cluster 1 centroid: (1.5, 2.75)

Cluster 2 centroid: (4.5, 2.5)

step 5: until centroids don't change

The centroids changed so we repeat.

step 3: assign each instance to the nearest centroid

Again we compute Manhattan Distance.

point	distance from centroid 1 (1.5, 2.75)	distance from centroid 2 (4.5, 2.5)
(1, 2)	1.25	4.0
(1, 4)	1.75	5.0
(2, 2)	1.25	3.0
(2, 3)	0.75	3.0
(4, 2)	3.25	1.0
(4, 4)	3.75	2.0
(5, 1)	5.25	2.0
(5, 3)	3.75	1.0

and based on these distances assign the points to clusters:

CLUSTER 1 (1, 2) (1, 4) (2, 2) (2, 3)
--

CLUSTER 2 (4, 2) (4, 4) (5, 1) (5, 3)
--

step 4: update centroids

We compute the new centroids by computing the mean of each cluster.

Cluster 1 centroid: (1.5, 2.75)

Cluster 2 centroid: (4.5, 2.5)

step 5: until centroids don't change

The updated centroids are identical to the previous ones so the algorithm converged on a solution and we can stop. The final clusters are

CLUSTER 1
(1, 2)
(1, 4)
(2, 2)
(2, 3)

CLUSTER 2
(4, 2)
(4, 4)
(5, 1)
(5, 3)

We stop when the centroids don't change. This is the same condition as saying no point are shifting from one cluster to another. This is what we mean when we say the algorithm 'converges'.


During the execution of the algorithm, the centroids shift from their initial position to some final position. The vast majority of this shift occurs during the first few iterations. Often, the centroids barely move during the final iterations.

This means that the k-means algorithm produces good clusters early on and later iterations are likely to produce only minor refinements.



Because of this behavior of the algorithm, we can dramatically reduce its execution time by relaxing our criteria of “no points are shifting from one cluster to another” to “fewer than 1% of the points are shifting from one cluster to another.”
This is a common approach!



 K-means is simple!



For you computer science geeks:

K-means is an instance of the Expectation Maximization (EM) Algorithm, which is an iterative method that alternates between two phases. We start with an initial estimate of some parameter. In the K-means case we start with an estimate of the centroids. In the expectation (E) phase, we use this estimate to place points into their **expected** cluster. In the Maximization (M) phase we use these expected values to adjust the estimate of the centroids. If you are interested in learning more about the EM algorithm the wikipedia page http://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm is a good place to start.

Hill Climbing

I would like to briefly interrupt our discussion of K-means clustering to talk about hill climbing algorithms. Suppose our goal is to reach the peak of some mountain and we come up with the following algorithm:




start at some random location on the mountain.

REPEAT

take a step in the direction that will take you higher.

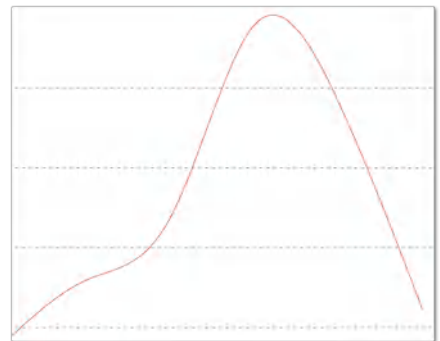
UNTIL there is no direction that will take you higher.

This seems like a reasonable algorithm.

Consider using it with the mountain shown here 

You can see that regardless of where we are plopped down on the mountain, we will reach the peak if we follow the algorithm.

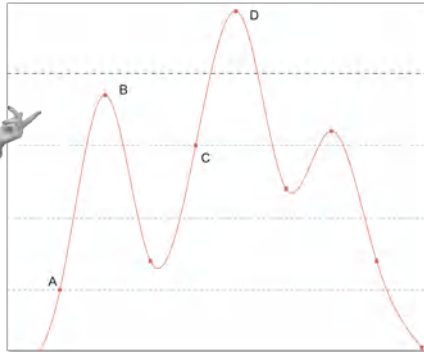
And if we think of this as a graph, we will reach the peak value regardless of where we start on the graph.



Now let's consider using the algorithm with the graph on the following page

Here, things don't work out as expected. If we start at 'A' on the graph...

We will reach the peak 'B' but not reach the highest peak 'D'. Or, to put it another way, we reach a local maximum, B, but not the global maximum, D.



Thus, this simple version of the hill-climbing algorithm is not guaranteed to reach the optimal solution.

The k-means clustering algorithm is like this. There is no guarantee that it will find the optimal division of the data into clusters. Why?

Because at the start of the algorithm we select an initial set of centroids randomly, which is much like picking a random spot like point 'A' on the graph above. Then, based on this initial set, we optimize the clusters finding the local optimum (similar to point 'B' on the graph).

The final clusters are heavily dependent on the selection of the initial centroids.

Even so, the k-means algorithm generates decent clusters.





How do we know whether one set of clusters (division of the data into clusters) is better than another?

SSE or Scatter

To determine the quality of a set of clusters we can use the **sum of the squared error** (SSE). This is also called **scatter**. Here is how to compute it: for each point we will square the distance from that point to its centroid, then add all those squared distances together. More formally,

$$SSE = \sum_{i=1}^k \sum_{x \in C_i} \text{dist}(c_i, x)^2$$

Let's dissect that. In the first summation sign we are iterating over the clusters. So initially i equals cluster 1, then i equals cluster 2, up to i equals cluster k . The next summation sign iterates over the points in that cluster—something like, for each point x in cluster i . *Dist* is whatever distance formula we are using (for example, Manhattan, or Euclidean). So we compute the distance between that point, x , and the centroid for the cluster c_i , square that distance and add it to our total.

Let's say we run our k -means algorithm twice on the same data and for each run we pick a different set of random initial centroids. Is the set of clusters that were computed during the first run worse or better than the set computed during the second run? To answer that question we compute the SSE for both sets of clusters. The set with the smaller SSE is the better of the two.

Time to start coding!

Here's the code for basic k-means



```
import math
import random
```

```
def getMedian(alist):
    """get median of list"""
    tmp = list(alist)
    tmp.sort()
    alen = len(tmp)
    if (alen % 2) == 1:
        return tmp[alen // 2]
    else:
        return (tmp[alen // 2] + tmp[(alen // 2) - 1]) / 2
```

```
def normalizeColumn(column):
    """normalize the values of a column using Modified Standard Score
    that is (each value - median) / (absolute standard deviation)"""
    median = getMedian(column)
    asd = sum([abs(x - median) for x in column]) / len(column)
    result = [(x - median) / asd for x in column]
    return result
```

```
class kClusterer:
    """ Implementation of kMeans Clustering
    This clusterer assumes that the first column of the data is a label
    not used in the clustering. The other columns contain numeric data
    """
```

```
def __init__(self, filename, k):
    """ k is the number of clusters to make
    This init method:
    1. reads the data from the file named filename
    2. stores that data by column in self.data
    3. normalizes the data using Modified Standard Score
```

```

    4. randomly selects the initial centroids
    5. assigns points to clusters associated with those centroids
"""
file = open(filename)
self.data = {}
self.k = k
self.counter = 0
self.iterationNumber = 0
# used to keep track of % of points that change cluster membership
# in an iteration
self.pointsChanged = 0
# Sum of Squared Error
self.sse = 0
#
# read data from file
#
lines = file.readlines()
file.close()
header = lines[0].split(',')
self.cols = len(header)
self.data = [[] for i in range(len(header))]
# we are storing the data by column.
# For example, self.data[0] is the data from column 0.
# self.data[0][10] is the column 0 value of item 10.
for line in lines[1:]:
    cells = line.split(',')
    toggle = 0
    for cell in range(self.cols):
        if toggle == 0:
            self.data[cell].append(cells[cell])
            toggle = 1
        else:
            self.data[cell].append(float(cells[cell]))

self.datasize = len(self.data[1])
self.memberOf = [-1 for x in range(len(self.data[1]))]
#
# now normalize number columns
#
for i in range(1, self.cols):
    self.data[i] = normalizeColumn(self.data[i])

# select random centroids from existing points
random.seed()
self.centroids = [[self.data[i][r] for i in range(1, len(self.data))],
                  for r in random.sample(range(len(self.data[0])),
                                         self.k)]

self.assignPointsToCluster()

```

```

def updateCentroids(self):
    """Using the points in the clusters, determine the centroid
    (mean point) of each cluster"""
    members = [self.memberOf.count(i) in range(len(self.centroids))]
    self.centroids = [[sum([self.data[k][i]
        for i in range(len(self.data[0]))
        if self.memberOf[i] == centroid])/members[centroid]
        for k in range(1, len(self.data))]
        for centroid in range(len(self.centroids))]

def assignPointToCluster(self, i):
    """ assign point to cluster based on distance from centroids"""
    min = 999999
    clusterNum = -1
    for centroid in range(self.k):
        dist = self.euclideanDistance(i, centroid)
        if dist < min:
            min = dist
            clusterNum = centroid
    # here is where I will keep track of changing points
    if clusterNum != self.memberOf[i]:
        self.pointsChanged += 1
    # add square of distance to running sum of squared error
    self.sse += min**2
    return clusterNum

def assignPointsToCluster(self):
    """ assign each data point to a cluster"""
    self.pointsChanged = 0
    self.sse = 0
    self.memberOf = [self.assignPointToCluster(i)
        for i in range(len(self.data[1])]

def euclideanDistance(self, i, j):
    """ compute distance of point i from centroid j"""
    sumSquares = 0
    for k in range(1, self.cols):
        sumSquares += (self.data[k][i] - self.centroids[j][k-1])**2
    return math.sqrt(sumSquares)

def kCluster(self):
    """the method that actually performs the clustering
    As you can see this method repeatedly
    updates the centroids by computing the mean point of each cluster
    re-assign the points to clusters based on these new centroids

```

```

until the number of points that change cluster membership
is less than 1%.
"""
done = False

while not done:
    self.iterationNumber += 1
    self.updateCentroids()
    self.assignPointsToCluster()
    #
    # we are done if fewer than 1% of the points change clusters
    #
    if float(self.pointsChanged) / len(self.memberOf) < 0.01:
        done = True
print("Final SSE: %f" % self.sse)

def showMembers(self):
    """Display the results"""
    for centroid in range(len(self.centroids)):
        print ("\n\nClass %i\n=====" % centroid)
        for name in [self.data[0][i] for i in range(len(self.data[0]))]:
            if self.memberOf[i] == centroid:
                print (name)

##
## RUN THE K-MEANS CLUSTERER ON THE DOG DATA USING K = 3
###
km = kClusterer('dogs2.csv', 3)
km.kCluster()
km.showMembers()

```



Let's dissect
that code a bit!

As with our code for the hierarchical clusterer, we are storing the data by column. Consider our dog breed data. If we represent the data in spreadsheet form, it would likely look like this (the height and weight are normalized):

breed	height	weight
Border Collie	0	-0.1455
Boston Terrier	-0.7213	-0.873
Brittany Spaniel	-0.3607	-0.4365
Bullmastiff	1.2623	2.03704
German Shepherd	0.9016	0.81481
...

And if we were to transfer this data to Python we would likely make a list that looks like the following:

```
data = [ data for the Border Collie,  
         data for the Boston Terrier,  
         ... ]
```

So to fully specify the data format:

```
data = [ ['Border Collie', 0, -0.1455],  
         ['Boston Terrier', -0.7213, -0.873],  
         ... ]
```

So we are storing the data by row. This seems like the common sense approach and the one we have been using throughout the book. Alternatively, we can store the data column first:

```
data = [ column 1 data,
        column 2 data,
        column 3 data ]
```

So for our dog example:

```
data = [ ['Border Collie', 'Boston Terrier', 'Brittany Spaniel', ...],
        [ 0, -0.7213, -0.3607, ...],
        [-0.1455, -0.7213, -0.4365, ...],
        ... ]
```

This is what we did for the hierarchical clusterer and what we are doing here for k-means. The benefit of this approach is that it makes implementing many of the math functions easier. We can see this in the first two procedures in the code above, `getMedian` and `normalizeColumn`. Because we stored the data by column, these procedures take simple lists as arguments.

```
>>> normalizeColumn([8, 6, 4, 2])
[1.5, 0.5, -0.5, -1.5]
```

The constructor method, `__init__` takes as arguments, the filename of the data file and `k`, the number of clusters to construct. It reads the data from the file and stores the data by column. It normalizes the data using the `normalizeColumn` procedure, which implements the Modified Standard Score method. Finally, it selects `k` elements from this data as the initial centroids and assigns each point to a cluster depending on that point's distance to the initial centroids. It does this assignment using the method `assignPointsToCluster`.

The method, `kCluster` actually performs the clustering by repeatedly calling `updateCentroids`, which computes the mean of each cluster and `assignPointsToCluster` until fewer than 1% of the points change clusters. The method `showMembers` simply displays the results.

Running the code on the dog breed data yields the following results:

```
Final SSE: 5.243159
```

```
Class 0
=====
Bullmastiff
Great Dane
```

Class 1

=====

Boston Terrier
Chihuahua
Yorkshire Terrier

Class 2

=====

Border Collie
Brittany Spaniel
German Shepherd
Golden Retriever
Portuguese Water Dog
Standard Poodle

Wow! For this small dataset the clusterer does extremely well.



You try

How well does the kmeans clusterer work with the cereal dataset with $k = 4$

- Do the sweet cereals cluster together (Cap'n'Crunch, Cocoa Puffs, Froot Loops, Lucky Charms?)
- Do the bran cereals cluster together (100% Bran, All-Bran, All-Bran with Extra Fiber, Bran Chex?)
- What does Cheerios cluster with?

Try the clusterer with the auto mpg dataset with different values for $k=8$?

Does this follow your expectations of how these cars should be grouped?



You try - my results

How well does the kmeans clusterer work with the cereal dataset with $k = 4$.

Your results may vary from mine but here is what I found out.

- Do the sweet cereals cluster together (Cap'n'Crunch, Cocoa Puffs, Froot Loops, Lucky Charms? Yes, all these sweet cereals (plus Count Chocula, Fruity Pebbles, and others) are in the same sweet cluster.
- Do the bran cereals cluster together (100% Bran, All-Bran, All-Bran with Extra Fiber, Bran Chex? Again, yes! Included in this cluster are also Raisin Bran and Fruitful Bran.
- What does Cheerios cluster with?
Cheerios always seems to be in the same cluster as Special K

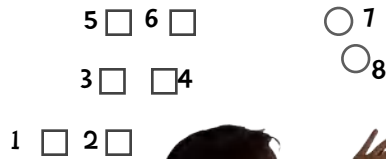
Try the clusterer with the auto mpg dataset with different values for $k=8$?

Does this follow your expectations of how these cars should be grouped?

The clusterer seems to do a reasonable job on this dataset but on rare occasions you will notice one or more of the clusters are empty.



OMG! I told the clusterer to make 8 groups but 1 of them is empty. There must be something wrong with the code!

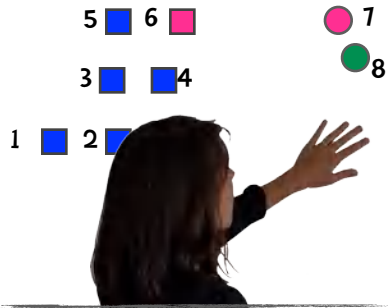


Consider clustering these points with $k = 3$. We randomly pick points 1, 7 & 8 as the initial centroids.¹

Nothing wrong with the code. Let's look at an example to see how this happens.

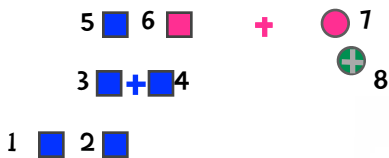


1. This example from Tolga Can http://www.ceng.metu.edu.tr/~tcan/ceng465_F1314/Schedule/KMeansEmpty.html

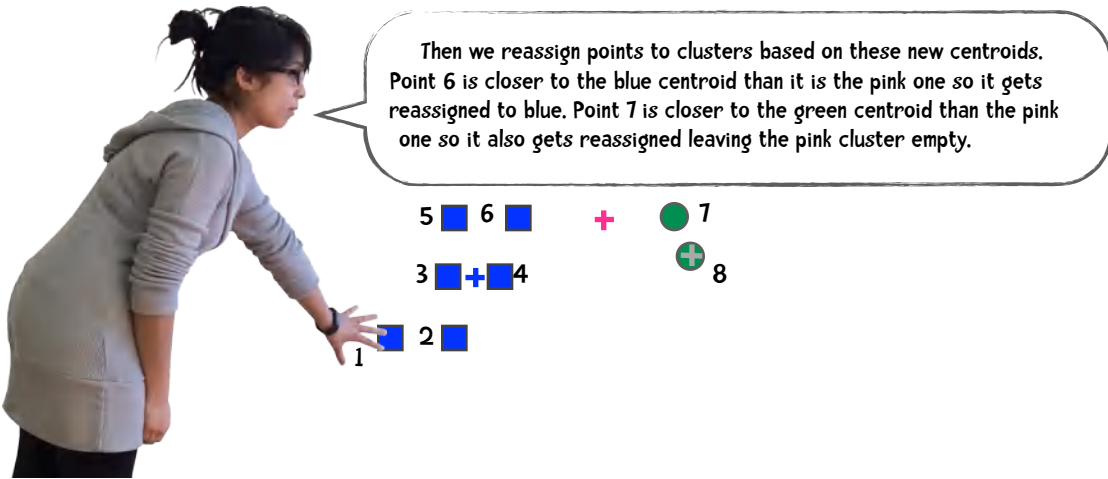


Here we assign the points to clusters. Point 6 is closer to point 7 than it is to point 1 so we assign it to the pink cluster.¹

Next we update the centroids (shown by the '+')



1. For those of you who are not looking at this in color, the pink cluster now contains points 6 and 7.



In sum, just because we specify how many groups to make does not mean that the k-means clusterer will produce that many non-empty groups. This may be a good thing. Just looking at the data above, it appears to be naturally clustered into two groups and our attempt to cluster the data into three failed. Suppose we have 1,000 instances we would like to cluster into 10 groups and when we run the clusterer two of the groups are empty. This result may indicate something about the underlying structure of the data. Perhaps the data does not naturally divide into ten groups and we can explore other groupings (trying to cluster into eight groups, for example).

On the other hand, sometimes when we specify 10 clusters we actually want 10 non-empty clusters. If that is the case, we need to alter the algorithm so it detects an empty cluster. Once one is detected the algorithm changes that cluster's centroid to a different point. One possibility is to change it to the instance that is furthest from its corresponding centroid. (In the example above, once we detect the pink cluster is empty, we re-assign the pink centroid to point 1, since point 1 is the furthest point to its corresponding centroid. That is, I compute the distances from

- 1 to its centroid
- 2 to its centroid
- 3 to its centroid
- 4 to its centroid
- 5 to its centroid
- 6 to its centroid
- 7 to its centroid
- 8 to its centroid

and pick the point that is furthest from its centroid as the new centroid of the empty cluster.



(sigh) Wouldn't it be dreamy if we could make *k*-means faster and more accurate.

With a simple change to *k*-means we can! The new algorithm is called ***k*-means++**

Even the name makes it sound newer, better, faster, and more accurate—a turbocharged *k*-means!



***k*-means++**

In the previous section we examined the *k*-means algorithm in its original form as it was developed in the late 50s. As we have seen, it is easy to implement and performs well. It is still the most widely used clustering algorithm on the planet. But it is not without its flaws. A major weakness in *k*-means is in the first step where it **randomly** picks *k* of the datapoints to be the initial centroids. As you can probably tell by my bolding and embiggening the word ‘random’, it is the random part that is the problem. Because it is random, sometimes the initial centroids are a great pick and lead to near optimal clustering. Other times the initial centroids are a reasonable pick and lead to good clustering. But sometimes—again, because we pick randomly—sometimes the initial centroids are poor leading to non-optimal clustering. The *k*-means++ algorithm fixes this defect by changing the way we pick the initial centroids. Everything else about *k*-means remains the same.

embiggen: verb. To make larger, to make the size increase.

k-means++ -- selecting the initial set of centroids

1. Initially, the set of initial centroids is empty.
2. Select the first centroid randomly from the data points as before.
3. Until we have k initial centroids:
 - a. Compute the distance, D , between each datapoint (dp) and its closest centroid. This distance is $D(dp)$.
 - b. In a probability proportional to $D(dp)$ select one datapoint at random to be a new centroid and add it to the set of centroids.
 - c. REPEAT



Let's dissect the meaning of "In a probability proportional to $D(dp)$ select one datapoint to be a new centroid." To do this, I will present a simple example. Suppose we are in the middle of this process. We have already selected two initial centroids and are in the process of selecting another one. So we are on step 3a of the above algorithm. Let's say we have 5 remaining centroids and their distances to the 2 centroids (c_1 and c_2) are as follows:

	Dc1	Dc2
dp1	5	7
dp2	9	8
dp3	2	5
dp4	3	7
dp5	5	2

Dc1 means "distance to centroid 1 and Dc2 means "distance to centroid 2." dp1 represents datapoint 1.

Step 3a says we pick the closest distance so we get:

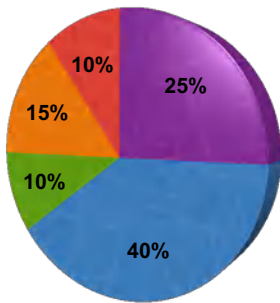
	closest
dp1	5
dp2	8
dp3	2
dp4	3
dp5	2

Now we are going to convert those numbers to a decimals whose sum equals 1 (I'll call this the weight). To do that we sum the original numbers. In this case the sum equals 20. Now we divide each number by the sum. The result is shown here



	weight
dp1	0.25
dp2	0.40
dp3	0.10
dp4	0.15
dp5	0.10
sum	1.00

I like to think of this as a roulette wheel that looks like this:



We are going to spin a ball on that wheel, see where it lands, and pick that as the new centroid. This is what we mean by “In a probability proportional to $D(dp)$ select one datapoint to be a new centroid.”

Let us rough out this idea in Python. Say we have a list tuples containing a datapoint and its weight

```
data = [("dp1", 0.25), ("dp2", 0.4), ("dp3", 0.1),  
        ("dp4", 0.15), ("dp5", 0.1)]
```

The function roulette will now select a datapoint in a probability proportional to its weight:

```
import random
random.seed()

def roulette(datalist):
    i = 0
    soFar = datalist[0][1]
    ball = random.random()
    while soFar < ball:
        i += 1
        soFar += datalist[i][1]
    return datalist[i][0]
```

If the function did pick with this proportion, we would predict that if we picked 100 times, 25 of them would be dp1; 40 of them would be dp2; 10 of them dp3; 15 dp4; and 10, dp5. Let's see if that is true:

```
import collections
results = collections.defaultdict(int)
for i in range(100):
    results[roulette(data)] += 1
print results
```

```
{'dp5': 11, 'dp4': 15, 'dp3': 10, 'dp2': 38, 'dp1': 26}
```

Great! Our function does return datapoints in roughly the correct proportion.

The idea in k-means++ clustering is that, while we still pick the initial centroids randomly, we prefer centroids that are far away from one another.





Code It

Can you implement k-means++ in Python?

Again, the only difference between our previous implementation of k-means and this code is in how we select the initial centroids. Make a copy of our original k-means code and modify it. Our original code created the initial centroids in this line:

```
self.centroids = [[self.data[i][r] for i in range(1, len(self.data))]
                  for r in random.sample(range(len(self.data[0])),
                  self.k)]
```

Let us replace that line with:

```
self.selectInitialCentroids()
```

Your job is to write that method!

Good luck!

Throughout the book, the author has been putting pictures of hip people like us using laptops in hopes of influencing you the reader to put down the book and do some coding.

If he has been successful, please let him know at ron.zacharski@gmail.com

a reader coding (and apparently enjoying it!)

the book put down





Code It -solution

Here is my version of `selectInitialCentroids`:

```
def distanceToClosestCentroid(self, point, centroidList):
    result = self.eDistance(point, centroidList[0])
    for centroid in centroidList[1:]:
        distance = self.eDistance(point, centroid)
        if distance < result:
            result = distance
    return result

def selectInitialCentroids(self):
    """implement the k-means++ method of selecting
    the set of initial centroids"""
    centroids = []
    total = 0
    # first step is to select a random first centroid
    current = random.choice(range(len(self.data[0])))
    centroids.append(current)
    # loop to select the rest of the centroids, one at a time
    for i in range(0, self.k - 1):
        # for every point in the data find its distance to
        # the closest centroid
        weights = [self.distanceToClosestCentroid(x, centroids)
                   for x in range(len(self.data[0]))]
        total = sum(weights)
        # instead of raw distances, convert so sum of weight = 1
        weights = [x / total for x in weights]
        #
        # now roll virtual die
        num = random.random()
        total = 0
        x = -1
        # the roulette wheel simulation
        while total < num:
            x += 1
            total += weights[x]
        centroids.append(x)
    self.centroids = [[self.data[i][r] for i in range(1, len(self.data))]
                       for r in centroids]
```

The Python code for the entire k-means++ classifier is on the book's website:
<http://guidetodatamining.com>

Summary

Clustering is all about discovery. However, the simple examples we have been using in this chapter may obscure this fundamental idea. After all, we know how to cluster breakfast cereals without a computer's help—sugary cereals, healthy cereals. And we know how to cluster car models—a Ford F150 goes in the truck category, a Mazda Miata in the sports car category, and a Honda Civic in the fuel efficient category. But consider a task where discovery IS important.

When we do a web search we are presented with a long list of results. For example, when I just did a Google search on “carbon sequestration” I get over 2.8 million results. A number of researchers have examined the benefits of clustering these results. Instead of that long list of carbon sequestration results we might also see categories like “carbon sequestration in freshwater wetlands” and “carbon sequestration in forests.”

Josh Gotbaum's team conducted extensive interviews with 3,000 people asking them questions about their values. Using these interviews they clustered the people into five groups. When they examined the clusters they gave them the descriptions:

1. extending opportunity to others
2. working within a community
3. achieving independence
4. focusing on family
5. defending righteousness

They then crafted targeted campaign ads to each group.

from *The Numerati* by Stephen Baker

We just learned two clustering techniques, hierarchical clustering and k-means. When should we use one over the other?

Got it! What about hierarchical clustering?

Brilliant! Maybe I should practice by trying it out on some new data.

Good question!

The benefits of K-means is that it is simple and has fast execution time. It is a great choice in general. It is also good choice for your first steps in exploring your data even if you eventually move to another clustering technique. However, it does not handle outliers well. Although, we can remedy this by identifying and removing the outliers.

The obvious use of hierarchical clustering is when we want to create a taxonomy or hierarchy from our data. This hierarchy may be more informative about the data than a flat set of clusters. It is also not as efficient in terms of execution speed and memory requirements.

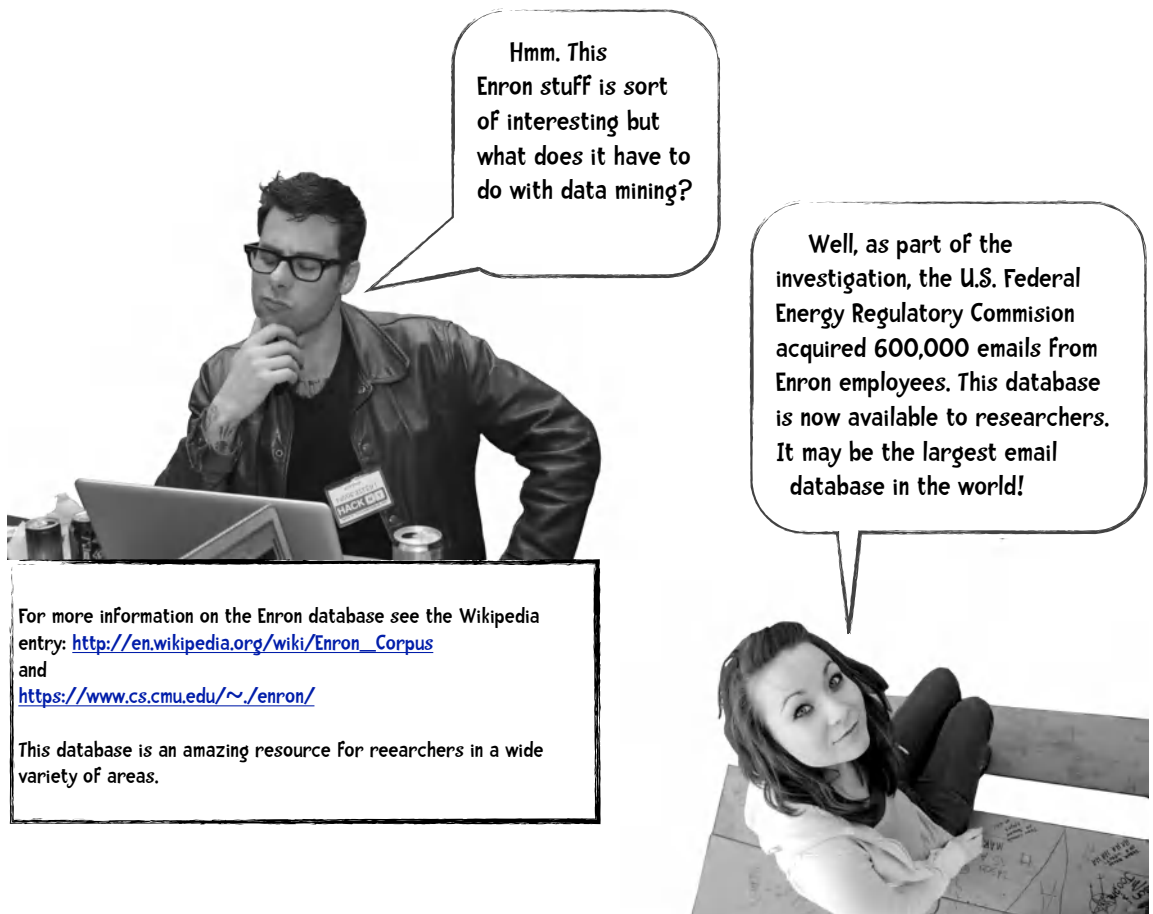


Enron

Perhaps you remember Enron and the Enron Scandal. In its heyday Enron was a mega-huge energy company with revenues over \$100 billion and over 20,000 employees (Microsoft's revenue then was only \$22 billion). Due to systemic sleaziness and corruption including creating an artificial energy shortage that resulted in electricity blackouts in California, Enron went bankrupt and a bunch of people went to jail. For a documentary about this see *Enron: The Smartest Guys in the Room*, which is available for streaming from Netflix and Amazon Prime.



Now you might be thinking “Hey, this Enron stuff is sort of interesting but what does it have to do with data mining?”



We are going to try to cluster a small part of the Enron corpus. For our simple test corpus, I have extracted the information of who sent email to whom and represented it in table form as shown here:

	Kay	Chris	Sara	Tana	Steven	Mark
Kay	0	53	37	6	0	12
Chris	53	0	1	0	2	0
Sara	37	1	0	1144	0	962
Tana	6	0	1144	0	0	1201
Steven	0	0	2	0	0	0
Mark	12	0	962	1201	0	0

In the dataset provided on our website, I've extracted this information for 90 individuals.

Suppose I am interested in clustering people to discover the relationships among these individuals.

Link analysis

There is an entire subfield of data mining called link analysis devoted to this type of problem (evaluating relationships among entities) and there are specialized algorithms devoted to this task.



You try

Can you perform hierarchical clustering on the Enron email dataset?

You can download the data from our website. (<http://www.guidetodatamining.com>). You may need to alter the code slightly to better match the problem.

Good luck!



You try - solution

In the dataset provided on our website, I've extracted this information for 90 individuals.

We are clustering the people based on similarity of email correspondence. If most of my email correspondence is with Ann, Ben and Clara, and most of yours is with these people as well, that provides evidence that we are in the same group. The idea is something like this:

between ->	Ann	Ben	Clara	Dongmei	Emily	Frank
my emails	127	25	119	5	1	6
your emails	172	35	123	7	3	5

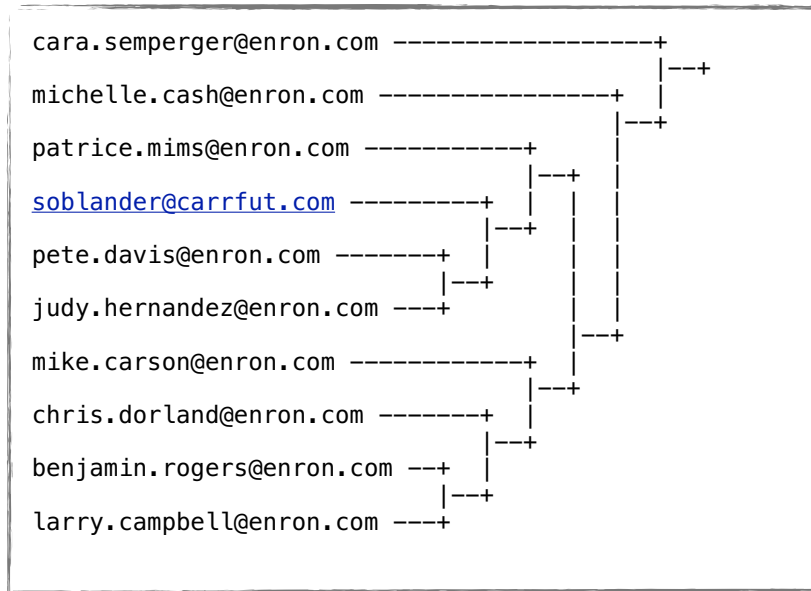
Because our rows are similar, we cluster together. A problem arises when we add in our columns:

between ->	me	you	Ann	Ben	Clara	Dongmei	Emily	Frank
my emails	2	190	127	25	119	5	1	6
your emails	190	3	172	35	123	7	3	5

In looking at the 'me' column, you corresponded with me 190 times but I only sent myself email twice. The 'you' column is similar. Now when we compare our rows they don't look so similar. Before I included the 'me' and 'you' columns the Euclidean distance was 46 and after I included them it was 269! To avoid this problem when I compute the Euclidean distance between two people I eliminate the columns for those two people. This required a slight change to the distance formula:

```
def distance(self, i, j):
    #enron specific distance formula
    sumSquares = 0
    for k in range(1, self.cols):
        if (k != i) and (k != j) :
            sumSquares += (self.data[k][i] - self.data[k][j])**2
    return math.sqrt(sumSquares)
```

Here is a subtree of the results:



I also performed k-means++ on the data, with $k = 8$. Here are some of the groups it discovered:

```

Class 5
=====
chris.germany@enron.com
scott.neal@enron.com
marie.heard@enron.com
leslie.hansen@enron.com
mike.carson@enron.com

Class 6
=====
sara.shackleton@enron.com
mark.taylor@enron.com
susan.scott@enron.com

Class 7
=====
tana.jones@enron.com
louise.kitchen@enron.com
mike.grigsby@enron.com
david.forster@enron.com
m.presto@enron.com

```

These results are interesting. Class 5 contains a number of traders. Chris Germany and Leslie Hansen are traders. Scott Neal is a vice president of trading. Marie Heard is a lawyer. Mike Carson is a manager of South East trading. The members of Class 7 are also interesting. All I know about Tana Jones is that she is an 'executive'. Louise Kitchen is President of online trading. Mike Grigsby was Vice President of Natural Gas. David Forster was a Vice President of trading. Kevin Presto (m.presto) was also a Vice President and a senior trader.

