O'REILLY®

Data Science from Scratch

FIRST PRINCIPLES WITH PYTHON

Joel Grus

Data Science from Scratch

by Joel Grus

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editor: Marie Beaugureau
- Production Editor: Melanie Yarbrough
- Copyeditor: Nan Reinhardt
- Proofreader: Eileen Cohen
- Indexer: Ellen Troutman-Zaig
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- April 2015: First Edition

Chapter 1. Introduction

“Data! Data! Data!” he cried impatiently. “I can’t make bricks without clay.”

Arthur Conan Doyle

The Ascendance of Data

We live in a world that's drowning in data. Websites track every user's every click. Your smartphone is building up a record of your location and speed every second of every day. "Quantified selfers" wear pedometers-on-steroids that are ever recording their heart rates, movement habits, diet, and sleep patterns. Smart cars collect driving habits, smart homes collect living habits, and smart marketers collect purchasing habits. The Internet itself represents a huge graph of knowledge that contains (among other things) an enormous cross-referenced encyclopedia; domain-specific databases about movies, music, sports results, pinball machines, memes, and cocktails; and too many government statistics (some of them nearly true!) from too many governments to wrap your head around.

Buried in these data are answers to countless questions that no one's ever thought to ask. In this book, we'll learn how to find them.

What Is Data Science?

There's a joke that says a data scientist is someone who knows more statistics than a computer scientist and more computer science than a statistician. (I didn't say it was a good joke.) In fact, some data scientists are — for all practical purposes — statisticians, while others are pretty much indistinguishable from software engineers. Some are machine-learning experts, while others couldn't machine-learn their way out of kindergarten. Some are PhDs with impressive publication records, while others have never read an academic paper (shame on them, though). In short, pretty much no matter how you define data science, you'll find practitioners for whom the definition is totally, absolutely wrong.

Nonetheless, we won't let that stop us from trying. We'll say that a data scientist is someone who extracts insights from messy data. Today's world is full of people trying to turn data into insight.

For instance, the dating site OkCupid asks its members to answer thousands of questions in order to find the most appropriate matches for them. But it also analyzes these results to figure out innocuous-sounding questions you can ask someone to find out how likely someone is to sleep with you on the first date.

Facebook asks you to list your hometown and your current location, ostensibly to make it easier for your friends to find and connect with you. But it also analyzes these locations to identify global migration patterns and where the fanbases of different football teams live.

As a large retailer, Target tracks your purchases and interactions, both online and in-store. And it uses the data to predictively model which of its customers are pregnant, to better market baby-related purchases to them.

In 2012, the Obama campaign employed dozens of data scientists who data-mined and experimented their way to identifying voters who needed extra attention, choosing optimal donor-specific fundraising appeals and programs, and focusing get-out-the-vote efforts where they were most likely to be useful. It is generally agreed that these efforts played an important role in the president's re-election, which means it is a safe bet that political campaigns of the future will become more and more data-driven, resulting in a never-ending arms race of data science and data collection.

Now, before you start feeling too jaded: some data scientists also occasionally use their skills for good — using data to make government more effective, to help the homeless, and to improve public health. But it certainly won't hurt your career if you like figuring out the best way to get people to click on advertisements.

Motivating Hypothetical: DataSciencester

Congratulations! You've just been hired to lead the data science efforts at DataSciencester, *the* social network for data scientists.

Despite being *for* data scientists, DataSciencester has never actually invested in building its own data science practice. (In fairness, DataSciencester has never really invested in building its product either.) That will be your job! Throughout the book, we'll be learning about data science concepts by solving problems that you encounter at work. Sometimes we'll look at data explicitly supplied by users, sometimes we'll look at data generated through their interactions with the site, and sometimes we'll even look at data from experiments that we'll design.

And because DataSciencester has a strong "not-invented-here" mentality, we'll be building our own tools from scratch. At the end, you'll have a pretty solid understanding of the fundamentals of data science. And you'll be ready to apply your skills at a company with a less shaky premise, or to any other problems that happen to interest you.

Welcome aboard, and good luck! (You're allowed to wear jeans on Fridays, and the bathroom is down the hall on the right.)

Chapter 2. A Crash Course in Python

> People are still crazy about Python after twenty-five years, which I find hard to believe.
>
> Michael Palin

All new employees at DataSciencester are required to go through new employee orientation, the most interesting part of which is a crash course in Python.

This is not a comprehensive Python tutorial but instead is intended to highlight the parts of the language that will be most important to us (some of which are often not the focus of Python tutorials).

Getting Python

You can download Python from python.org. But if you don't already have Python, I recommend instead installing the Anaconda distribution, which already includes most of the libraries that you need to do data science.

As I write this, the latest version of Python is 3.4. At DataSciencester, however, we use old, reliable Python 2.7. Python 3 is not backward-compatible with Python 2, and many important libraries only work well with 2.7. The data science community is still firmly stuck on 2.7, which means we will be, too. Make sure to get that version.

If you don't get Anaconda, make sure to install pip, which is a Python package manager that allows you to easily install third-party packages (some of which we'll need). It's also worth getting IPython, which is a much nicer Python shell to work with.

(If you installed Anaconda then it should have come with pip and IPython.)

Just run:

```
pip install ipython
```

and then search the Internet for solutions to whatever cryptic error messages that causes.

The Zen of Python

Python has a somewhat Zen description of its design principles, which you can also find inside the Python interpreter itself by typing `import this`.

One of the most discussed of these is:

> There should be one — and preferably only one — obvious way to do it.

Code written in accordance with this "obvious" way (which may not be obvious at all to a newcomer) is often described as "Pythonic." Although this is not a book about Python, we will occasionally contrast Pythonic and non-Pythonic ways of accomplishing the same things, and we will generally favor Pythonic solutions to our problems.

Whitespace Formatting

Many languages use curly braces to delimit blocks of code. Python uses indentation:

```
for i in [1, 2, 3, 4, 5]:
    print i                     # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print j                 # first line in "for j" block
        print i + j             # last line in "for j" block
    print i                     # last line in "for i" block
print "done looping"
```

This makes Python code very readable, but it also means that you have to be very careful with your formatting. Whitespace is ignored inside parentheses and brackets, which can be helpful for long-winded computations:

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 +
                           13 + 14 + 15 + 16 + 17 + 18 + 19 + 20)
```

and for making code easier to read:

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

easier_to_read_list_of_lists = [ [1, 2, 3],
                                 [4, 5, 6],
                                 [7, 8, 9] ]
```

You can also use a backslash to indicate that a statement continues onto the next line, although we'll rarely do this:

```
two_plus_three = 2 + \
                 3
```

One consequence of whitespace formatting is that it can be hard to copy and paste code into the Python shell. For example, if you tried to paste the code:

```
for i in [1, 2, 3, 4, 5]:

    # notice the blank line
    print i
```

into the ordinary Python shell, you would get a:

```
IndentationError: expected an indented block
```

because the interpreter thinks the blank line signals the end of the `for` loop's block.

IPython has a magic function `%paste`, which correctly pastes whatever is on your clipboard, whitespace and all. This alone is a good reason to use IPython.

Chapter 3. Visualizing Data

> I believe that visualization is one of the most powerful means of achieving personal goals.
>
> Harvey Mackay

A fundamental part of the data scientist's toolkit is data visualization. Although it is very easy to create visualizations, it's much harder to produce *good* ones.

There are two primary uses for data visualization:

- To *explore* data
- To *communicate* data

In this chapter, we will concentrate on building the skills that you'll need to start exploring your own data and to produce the visualizations we'll be using throughout the rest of the book. Like most of our chapter topics, data visualization is a rich field of study that deserves its own book. Nonetheless, we'll try to give you a sense of what makes for a good visualization and what doesn't.

matplotlib

A wide variety of tools exists for visualizing data. We will be using the `matplotlib` library, which is widely used (although sort of showing its age). If you are interested in producing elaborate interactive visualizations for the Web, it is likely not the right choice, but for simple bar charts, line charts, and scatterplots, it works pretty well.

In particular, we will be using the `matplotlib.pyplot` module. In its simplest use, `pyplot` maintains an internal state in which you build up a visualization step by step. Once you're done, you can save it (with `savefig()`) or display it (with `show()`).

For example, making simple plots (like Figure 3-1) is pretty simple:

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# create a line chart, years on x-axis, gdp on y-axis
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# add a title
plt.title("Nominal GDP")

# add a label to the y-axis
plt.ylabel("Billions of $")
plt.show()
```

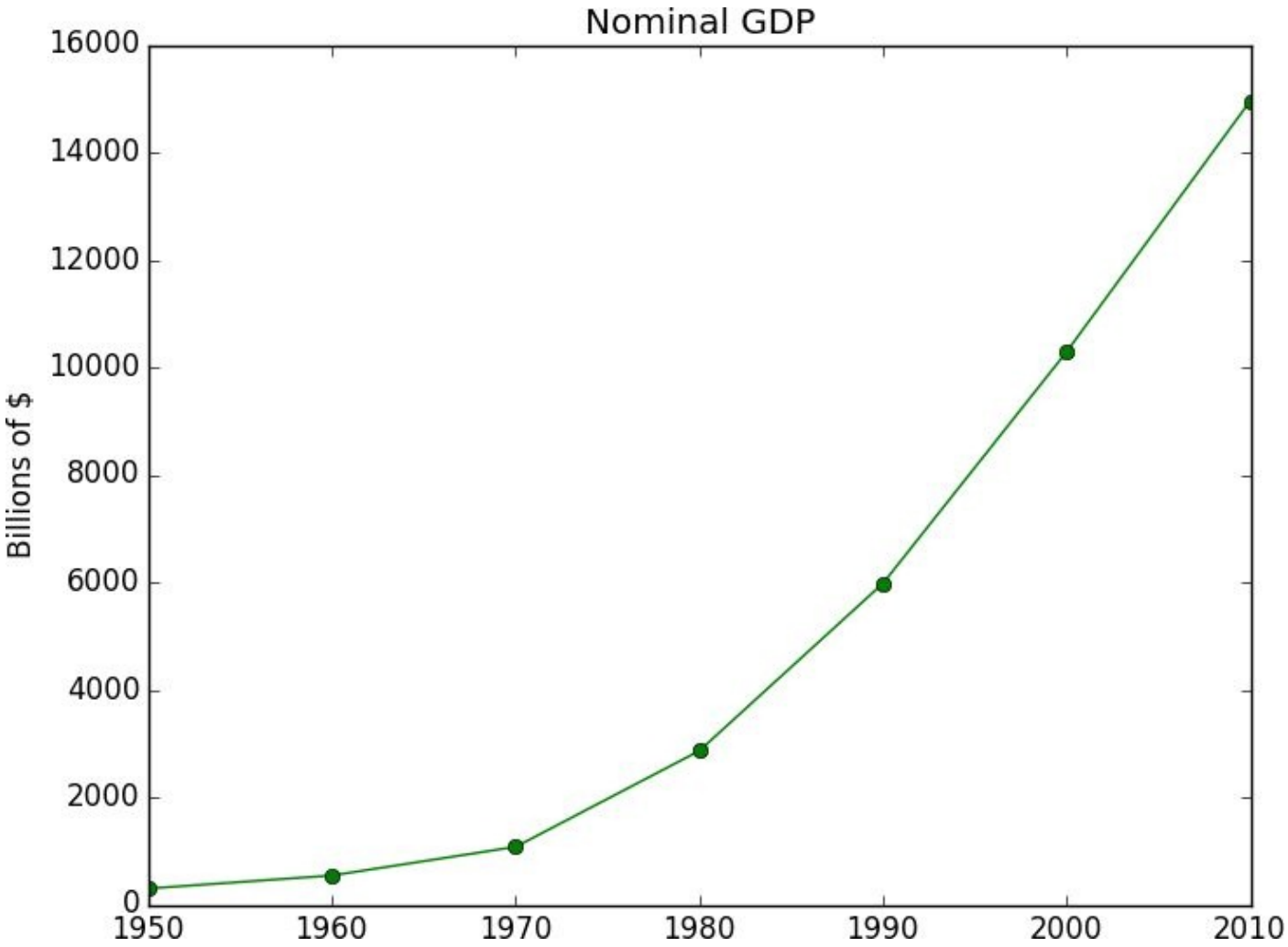

Figure 3-1. A simple line chart

Making plots that look publication-quality good is more complicated and beyond the scope of this chapter. There are many ways you can customize your charts with (for example) axis labels, line styles, and point markers. Rather than attempt a comprehensive treatment of these options, we'll just use (and call attention to) some of them in our examples.

> **NOTE**
>
> Although we won't be using much of this functionality, `matplotlib` is capable of producing complicated plots within plots, sophisticated formatting, and interactive visualizations. Check out its documentation if you want to go deeper than we do in this book.

Bar Charts

A bar chart is a good choice when you want to show how some quantity varies among some *discrete* set of items. For instance, Figure 3-2 shows how many Academy Awards were won by each of a variety of movies:

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Story"]
num_oscars = [5, 11, 3, 8, 10]

# bars are by default width 0.8, so we'll add 0.1 to the left coordinates
# so that each bar is centered
xs = [i + 0.1 for i, _ in enumerate(movies)]

# plot bars with left x-coordinates [xs], heights [num_oscars]
plt.bar(xs, num_oscars)

plt.ylabel("# of Academy Awards")
plt.title("My Favorite Movies")

# label x-axis with movie names at bar centers
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)

plt.show()
```

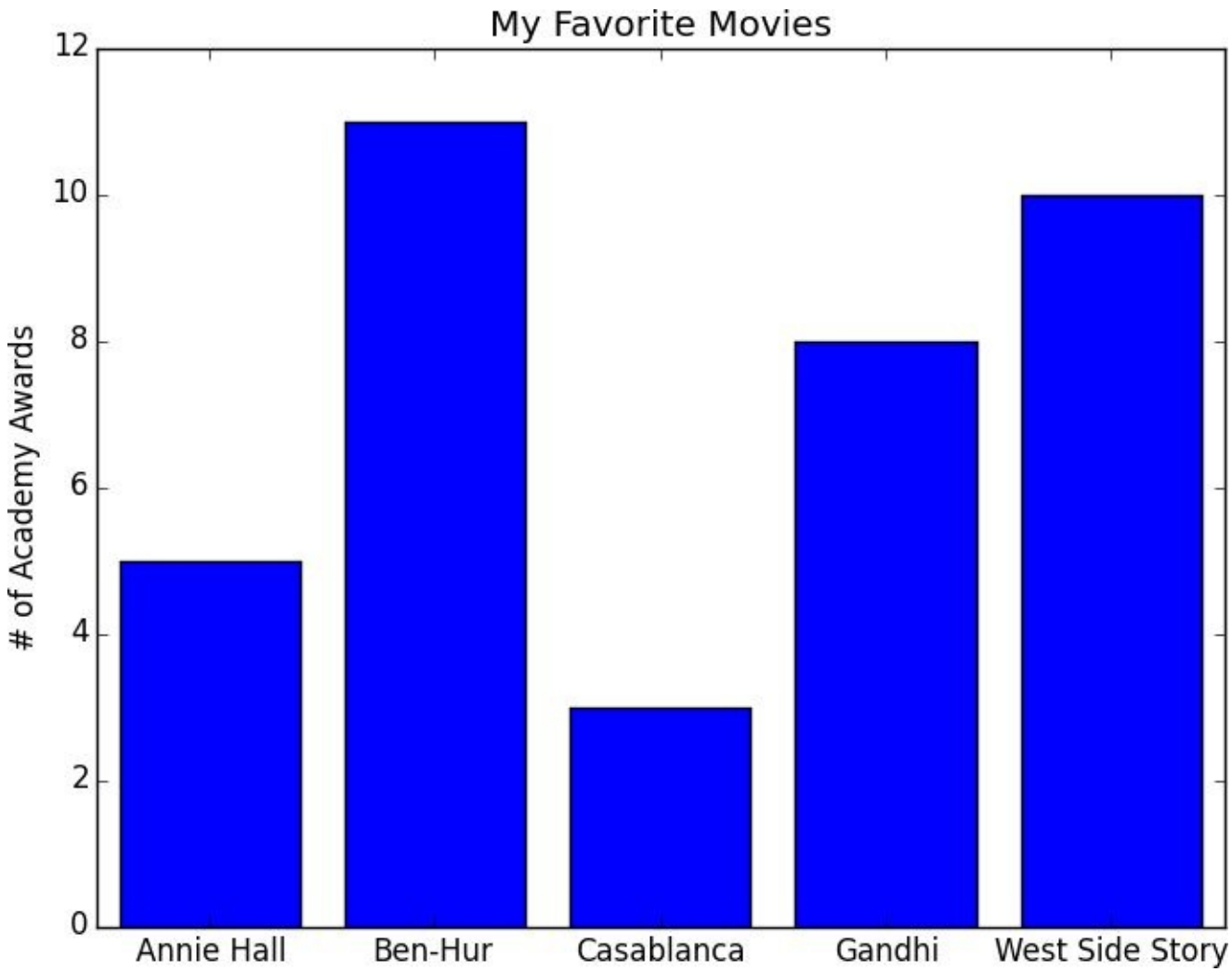

Figure 3-2. A simple bar chart

A bar chart can also be a good choice for plotting histograms of bucketed numeric values, in order to visually explore how the values are *distributed*, as in Figure 3-3:

Chapter 5. Statistics

> Facts are stubborn, but statistics are more pliable.
>
> Mark Twain

Statistics refers to the mathematics and techniques with which we understand data. It is a rich, enormous field, more suited to a shelf (or room) in a library rather than a chapter in a book, and so our discussion will necessarily not be a deep one. Instead, I'll try to teach you just enough to be dangerous, and pique your interest just enough that you'll go off and learn more.

Describing a Single Set of Data

Through a combination of word-of-mouth and luck, DataSciencester has grown to dozens of members, and the VP of Fundraising asks you for some sort of description of how many friends your members have that he can include in his elevator pitches.

Using techniques from Chapter 1, you are easily able to produce this data. But now you are faced with the problem of how to *describe* it.

One obvious description of any data set is simply the data itself:

```
num_friends = [100, 49, 41, 40, 25,
               # ... and lots more
              ]
```

For a small enough data set this might even be the best description. But for a larger data set, this is unwieldy and probably opaque. (Imagine staring at a list of 1 million numbers.) For that reason we use statistics to distill and communicate relevant features of our data.

As a first approach you put the friend counts into a histogram using `Counter` and `plt.bar()` (Figure 5-1):

```
friend_counts = Counter(num_friends)
xs = range(101)                         # largest value is 100
ys = [friend_counts[x] for x in xs]     # height is just # of friends
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
plt.title("Histogram of Friend Counts")
plt.xlabel("# of friends")
plt.ylabel("# of people")
plt.show()
```

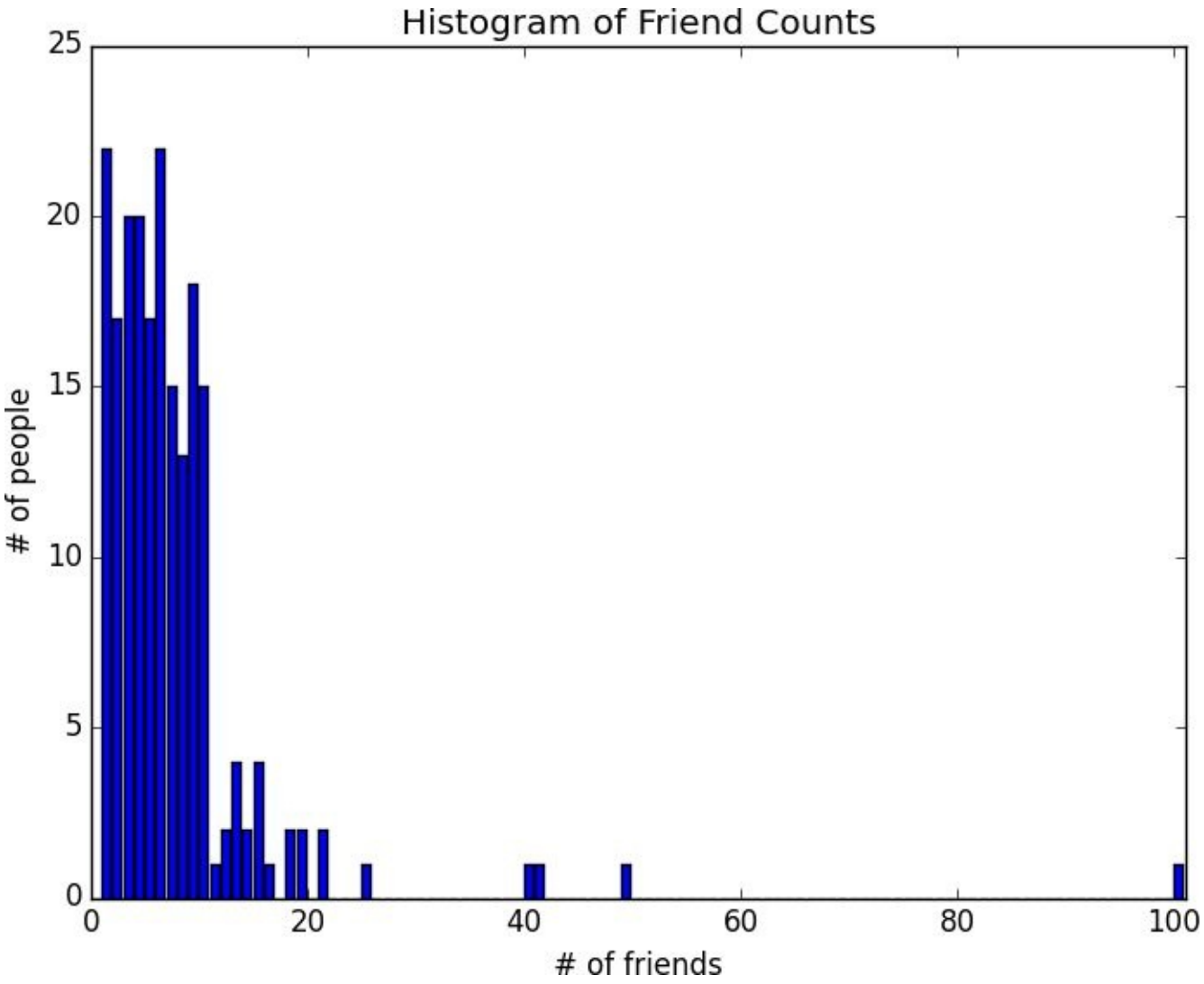

Figure 5-1. A histogram of friend counts

Unfortunately, this chart is still too difficult to slip into conversations. So you start generating some statistics. Probably the simplest statistic is simply the number of data points:

```
num_points = len(num_friends)              # 204
```

You're probably also interested in the largest and smallest values:

```
largest_value = max(num_friends)           # 100
smallest_value = min(num_friends)          # 1
```

which are just special cases of wanting to know the values in specific positions:

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]          # 1
second_smallest_value = sorted_values[1]   # 1
second_largest_value = sorted_values[-2]   # 49
```

But we're only getting started.

Central Tendencies

Usually, we'll want some notion of where our data is centered. Most commonly we'll use the *mean* (or average), which is just the sum of the data divided by its count:

```
# this isn't right if you don't from __future__ import division
def mean(x):
    return sum(x) / len(x)

mean(num_friends)   # 7.333333
```

If you have two data points, the mean is simply the point halfway between them. As you add more points, the mean shifts around, but it always depends on the value of every point.

We'll also sometimes be interested in the *median*, which is the middle-most value (if the number of data points is odd) or the average of the two middle-most values (if the number of data points is even).

For instance, if we have five data points in a sorted vector `x`, the median is `x[5 // 2]` or `x[2]`. If we have six data points, we want the average of `x[2]` (the third point) and `x[3]` (the fourth point).

Notice that — unlike the mean — the median doesn't depend on every value in your data. For example, if you make the largest point larger (or the smallest point smaller), the middle points remain unchanged, which means so does the median.

The `median` function is slightly more complicated than you might expect, mostly because of the "even" case:

```
def median(v):
    """finds the 'middle-most' value of v"""
    n = len(v)
    sorted_v = sorted(v)
    midpoint = n // 2

    if n % 2 == 1:
        # if odd, return the middle value
        return sorted_v[midpoint]
    else:
        # if even, return the average of the middle values
        lo = midpoint - 1
        hi = midpoint
        return (sorted_v[lo] + sorted_v[hi]) / 2

median(num_friends) # 6.0
```

Clearly, the mean is simpler to compute, and it varies smoothly as our data changes. If we have *n* data points and one of them increases by some small amount *e*, then necessarily the mean will increase by *e* / *n*. (This makes the mean amenable to all sorts of calculus tricks.) Whereas in order to find the median, we have to sort our data. And changing one of our data points by a small amount *e* might increase the median by *e*, by some number less than *e*, or not at all (depending on the rest of the data).

Chapter 6. Probability

> The laws of probability, so true in general, so fallacious in particular.
>
> Edward Gibbon

It is hard to do data science without some sort of understanding of *probability* and its mathematics. As with our treatment of statistics in Chapter 5, we'll wave our hands a lot and elide many of the technicalities.

For our purposes you should think of probability as a way of quantifying the uncertainty associated with *events* chosen from a some *universe* of events. Rather than getting technical about what these terms mean, think of rolling a die. The universe consists of all possible outcomes. And any subset of these outcomes is an event; for example, "the die rolls a one" or "the die rolls an even number."

Notationally, we write $P(E)$ to mean "the probability of the event E."

We'll use probability theory to build models. We'll use probability theory to evaluate models. We'll use probability theory all over the place.

One could, were one so inclined, get really deep into the philosophy of what probability theory *means*. (This is best done over beers.) We won't be doing that.

Dependence and Independence

Roughly speaking, we say that two events E and F are *dependent* if knowing something about whether E happens gives us information about whether F happens (and vice versa). Otherwise they are *independent*.

For instance, if we flip a fair coin twice, knowing whether the first flip is Heads gives us no information about whether the second flip is Heads. These events are independent. On the other hand, knowing whether the first flip is Heads certainly gives us information about whether both flips are Tails. (If the first flip is Heads, then definitely it's not the case that both flips are Tails.) These two events are dependent.

Mathematically, we say that two events E and F are independent if the probability that they both happen is the product of the probabilities that each one happens:

$$P(E, F) = P(E)P(F)$$

In the example above, the probability of "first flip Heads" is 1/2, and the probability of "both flips Tails" is 1/4, but the probability of "first flip Heads *and* both flips Tails" is 0.

Conditional Probability

When two events E and F are independent, then by definition we have:

$$P(E, F) = P(E)P(F)$$

If they are not necessarily independent (and if the probability of F is not zero), then we define the probability of E "conditional on F" as:

$$P(E \mid F) = P(E, F) / P(F)$$

You should think of this as the probability that E happens, given that we know that F happens.

We often rewrite this as:

$$P(E, F) = P(E \mid F)P(F)$$

When E and F are independent, you can check that this gives:

$$P(E \mid F) = P(E)$$

which is the mathematical way of expressing that knowing F occurred gives us no additional information about whether E occurred.

One common tricky example involves a family with two (unknown) children.

If we assume that:

1. Each child is equally likely to be a boy or a girl
2. The gender of the second child is independent of the gender of the first child

then the event "no girls" has probability 1/4, the event "one girl, one boy" has probability 1/2, and the event "two girls" has probability 1/4.

Now we can ask what is the probability of the event "both children are girls" (B) conditional on the event "the older child is a girl" (G)? Using the definition of conditional probability:

$$P(B \mid G) = P(B, G) / P(G) = P(B) / P(G) = 1 / 2$$

since the event B and G ("both children are girls *and* the older child is a girl") is just the event B. (Once you know that both children are girls, it's necessarily true that the older child is a girl.)

Most likely this result accords with your intuition.

We could also ask about the probability of the event “both children are girls” conditional on the event “at least one of the children is a girl” (L). Surprisingly, the answer is different from before!

As before, the event B and L (“both children are girls *and* at least one of the children is a girl”) is just the event B. This means we have:

$$P(B \mid L) = P(B, L) / P(L) = P(B) / P(L) = 1 / 3$$

How can this be the case? Well, if all you know is that at least one of the children is a girl, then it is twice as likely that the family has one boy and one girl than that it has both girls.

We can check this by “generating” a lot of families:

```
def random_kid():
    return random.choice(["boy", "girl"])

both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)
for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == "girl":
        older_girl += 1
    if older == "girl" and younger == "girl":
        both_girls += 1
    if older == "girl" or younger == "girl":
        either_girl += 1

print "P(both | older):", both_girls / older_girl      # 0.514 ~ 1/2
print "P(both | either): ", both_girls / either_girl   # 0.342 ~ 1/3
```

Chapter 7. Hypothesis and Inference

> It is the mark of a truly intelligent person to be moved by statistics.
>
> George Bernard Shaw

What will we do with all this statistics and probability theory? The *science* part of data science frequently involves forming and testing *hypotheses* about our data and the processes that generate it.

Statistical Hypothesis Testing

Often, as data scientists, we'll want to test whether a certain hypothesis is likely to be true. For our purposes, hypotheses are assertions like "this coin is fair" or "data scientists prefer Python to R" or "people are more likely to navigate away from the page without ever reading the content if we pop up an irritating interstitial advertisement with a tiny, hard-to-find close button" that can be translated into statistics about data. Under various assumptions, those statistics can be thought of as observations of random variables from known distributions, which allows us to make statements about how likely those assumptions are to hold.

In the classical setup, we have a *null hypothesis* H_0 that represents some default position, and some alternative hypothesis H_1 that we'd like to compare it with. We use statistics to decide whether we can reject H_0 as false or not. This will probably make more sense with an example.

Example: Flipping a Coin

Imagine we have a coin and we want to test whether it's fair. We'll make the assumption that the coin has some probability *p* of landing heads, and so our null hypothesis is that the coin is fair — that is, that $p = 0.5$. We'll test this against the alternative hypothesis $p \neq 0.5$.

In particular, our test will involve flipping the coin some number *n* times and counting the number of heads *X*. Each coin flip is a Bernoulli trial, which means that *X* is a Binomial(n,p) random variable, which (as we saw in Chapter 6) we can approximate using the normal distribution:

```
def normal_approximation_to_binomial(n, p):
    """finds mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

Whenever a random variable follows a normal distribution, we can use `normal_cdf` to figure out the probability that its realized value lies within (or outside) a particular interval:

```
# the normal cdf _is_ the probability the variable is below a threshold
normal_probability_below = normal_cdf

# it's above the threshold if it's not below the threshold
def normal_probability_above(lo, mu=0, sigma=1):
    return 1 - normal_cdf(lo, mu, sigma)

# it's between if it's less than hi, but not less than lo
def normal_probability_between(lo, hi, mu=0, sigma=1):
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# it's outside if it's not between
def normal_probability_outside(lo, hi, mu=0, sigma=1):
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

We can also do the reverse — find either the nontail region or the (symmetric) interval around the mean that accounts for a certain level of likelihood. For example, if we want to find an interval centered at the mean and containing 60% probability, then we find the cutoffs where the upper and lower tails each contain 20% of the probability (leaving 60%):

```
def normal_upper_bound(probability, mu=0, sigma=1):
    """returns the z for which P(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability, mu=0, sigma=1):
    """returns the z for which P(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability, mu=0, sigma=1):
    """returns the symmetric (about the mean) bounds
    that contain the specified probability"""
    tail_probability = (1 - probability) / 2

    # upper bound should have tail_probability above it
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)
```

```
    # lower bound should have tail_probability below it
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)

    return lower_bound, upper_bound
```

In particular, let's say that we choose to flip the coin $n = 1000$ times. If our hypothesis of fairness is true, *X* should be distributed approximately normally with mean 50 and standard deviation 15.8:

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

We need to make a decision about *significance* — how willing we are to make a *type 1 error* ("false positive"), in which we reject H_0 even though it's true. For reasons lost to the annals of history, this willingness is often set at 5% or 1%. Let's choose 5%.

Consider the test that rejects H_0 if *X* falls outside the bounds given by:

```
normal_two_sided_bounds(0.95, mu_0, sigma_0)   # (469, 531)
```

Assuming *p* really equals 0.5 (i.e., H_0 is true), there is just a 5% chance we observe an *X* that lies outside this interval, which is the exact significance we wanted. Said differently, if H_0 is true, then, approximately 19 times out of 20, this test will give the correct result.

We are also often interested in the *power* of a test, which is the probability of not making a *type 2 error*, in which we fail to reject H_0 even though it's false. In order to measure this, we have to specify what exactly H_0 being false *means*. (Knowing merely that *p* is *not* 0.5 doesn't give you a ton of information about the distribution of *X*.) In particular, let's check what happens if *p* is really 0.55, so that the coin is slightly biased toward heads.

In that case, we can calculate the power of the test with:

```
# 95% bounds based on assumption p is 0.5
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# actual mu and sigma based on p = 0.55
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)

# a type 2 error means we fail to reject the null hypothesis
# which will happen when X is still in our original interval
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.887
```

Imagine instead that our null hypothesis was that the coin is not biased toward heads, or that $p \leq 0.5$. In that case we want a *one-sided test* that rejects the null hypothesis when *X* is much larger than 50 but not when *X* is smaller than 50. So a 5%-significance test involves using `normal_probability_below` to find the cutoff below which 95% of the probability lies:

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# is 526 (< 531, since we need more probability in the upper tail)

type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
```

Chapter 8. Gradient Descent

> Those who boast of their descent, brag on what they owe to others.
>
> Seneca

Frequently when doing data science, we'll be trying to the find the best model for a certain situation. And usually "best" will mean something like "minimizes the error of the model" or "maximizes the likelihood of the data." In other words, it will represent the solution to some sort of optimization problem.

This means we'll need to solve a number of optimization problems. And in particular, we'll need to solve them from scratch. Our approach will be a technique called *gradient descent*, which lends itself pretty well to a from-scratch treatment. You might not find it super exciting in and of itself, but it will enable us to do exciting things throughout the book, so bear with me.

The Idea Behind Gradient Descent

Suppose we have some function `f` that takes as input a vector of real numbers and outputs a single real number. One simple such function is:

```
def sum_of_squares(v):
    """computes the sum of squared elements in v"""
    return sum(v_i ** 2 for v_i in v)
```

We'll frequently need to maximize (or minimize) such functions. That is, we need to find the input `v` that produces the largest (or smallest) possible value.

For functions like ours, the *gradient* (if you remember your calculus, this is the vector of partial derivatives) gives the input direction in which the function most quickly increases. (If you don't remember your calculus, take my word for it or look it up on the Internet.)

Accordingly, one approach to maximizing a function is to pick a random starting point, compute the gradient, take a small step in the direction of the gradient (i.e., the direction that causes the function to increase the most), and repeat with the new starting point. Similarly, you can try to minimize a function by taking small steps in the *opposite* direction, as shown in Figure 8-1.

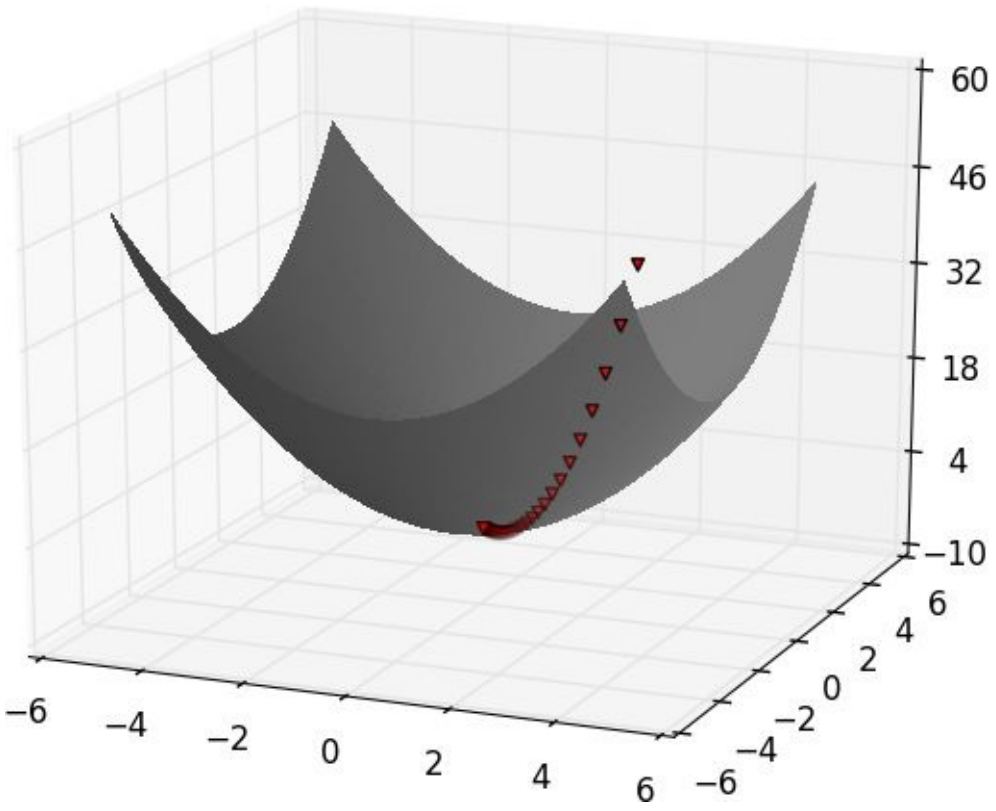

Figure 8-1. Finding a minimum using gradient descent

NOTE

If a function has a unique global minimum, this procedure is likely to find it. If a function has multiple (local) minima, this procedure might "find" the wrong one of them, in which case you might re-run the procedure from a variety of starting points. If a function has no minimum, then it's possible the procedure might go on forever.

Estimating the Gradient

If `f` is a function of one variable, its derivative at a point `x` measures how `f(x)` changes when we make a very small change to `x`. It is defined as the limit of the difference quotients:

```
def difference_quotient(f, x, h):
    return (f(x + h) - f(x)) / h
```

as `h` approaches zero.

(Many a would-be calculus student has been stymied by the mathematical definition of limit. Here we'll cheat and simply say that it means what you think it means.)

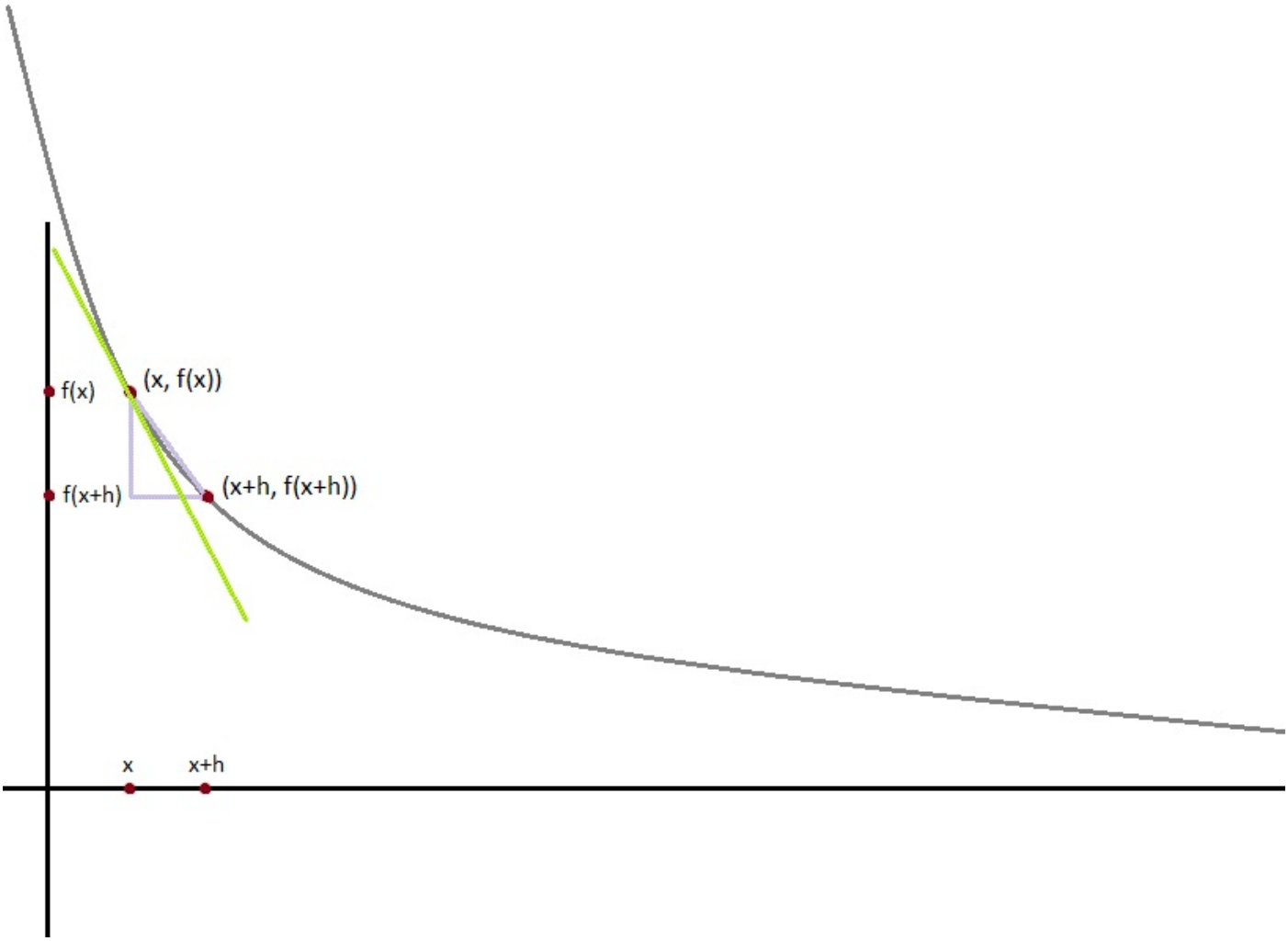

Figure 8-2. Approximating a derivative with a difference quotient

The derivative is the slope of the tangent line at $(x, f(x))$, while the difference quotient is the slope of the not-quite-tangent line that runs through $(x + h, f(x + h))$. As *h* gets smaller and smaller, the not-quite-tangent line gets closer and closer to the tangent line (Figure 8-2).

For many functions it's easy to exactly calculate derivatives. For example, the `square` function:

```
def square(x):
    return x * x
```

has the derivative:

```
def derivative(x):
    return 2 * x
```

which you can check — if you are so inclined — by explicitly computing the difference quotient and taking the limit.

What if you couldn't (or didn't want to) find the gradient? Although we can't take limits in Python, we can estimate derivatives by evaluating the difference quotient for a very small e. Figure 8-3 shows the results of one such estimation:

```
derivative_estimate = partial(difference_quotient, square, h=0.00001)

# plot to show they're basically the same
import matplotlib.pyplot as plt
x = range(-10,10)
plt.title("Actual Derivatives vs. Estimates")
plt.plot(x, map(derivative, x), 'rx', label='Actual')            # red  x
plt.plot(x, map(derivative_estimate, x), 'b+', label='Estimate')  # blue +
plt.legend(loc=9)
plt.show()
```

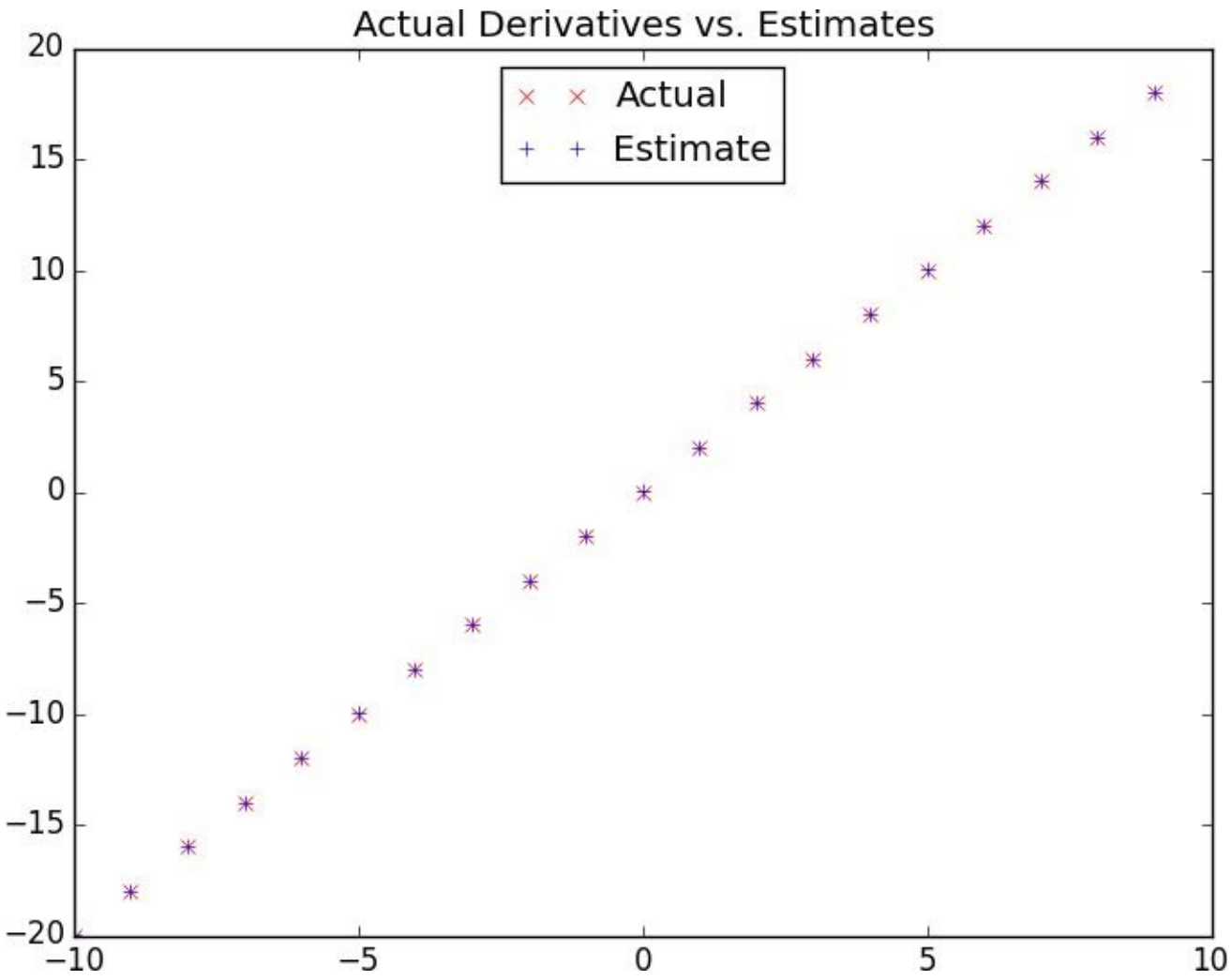

Figure 8-3. Goodness of difference quotient approximation

Chapter 9. Getting Data

> To write it, it took three months; to conceive it, three minutes; to collect the data in it, all my life.
>
> F. Scott Fitzgerald

In order to be a data scientist you need data. In fact, as a data scientist you will spend an embarrassingly large fraction of your time acquiring, cleaning, and transforming data. In a pinch, you can always type the data in yourself (or if you have minions, make them do it), but usually this is not a good use of your time. In this chapter, we'll look at different ways of getting data into Python and into the right formats.

stdin and stdout

If you run your Python scripts at the command line, you can *pipe* data through them using `sys.stdin` and `sys.stdout`. For example, here is a script that reads in lines of text and spits back out the ones that match a regular expression:

```
# egrep.py
import sys, re

# sys.argv is the list of command-line arguments
# sys.argv[0] is the name of the program itself
# sys.argv[1] will be the regex specified at the command line
regex = sys.argv[1]

# for every line passed into the script
for line in sys.stdin:
    # if it matches the regex, write it to stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

And here's one that counts the lines it receives and then writes out the count:

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1

# print goes to sys.stdout
print count
```

You could then use these to count how many lines of a file contain numbers. In Windows, you'd use:

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

whereas in a Unix system you'd use:

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

The | is the pipe character, which means "use the output of the left command as the input of the right command." You can build pretty elaborate data-processing pipelines this way.

> **NOTE**
>
> If you are using Windows, you can probably leave out the `python` part of this command:
>
> ```
> type SomeFile.txt | egrep.py "[0-9]" | line_count.py
> ```
>
> If you are on a Unix system, doing so might require a little more work.

Similarly, here's a script that counts the words in its input and writes out the most common ones:

```
# most_common_words.py
```

```
import sys
from collections import Counter

# pass in number of words as first argument
try:
    num_words = int(sys.argv[1])
except:
    print "usage: most_common_words.py num_words"
    sys.exit(1)   # non-zero exit code indicates error

counter = Counter(word.lower()                      # lowercase words
                  for line in sys.stdin             #
                  for word in line.strip().split()  # split on spaces
                  if word)                          # skip empty 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

after which you could do something like:

```
C:\DataScience>type the_bible.txt | python most_common_words.py 10
64193   the
51380   and
34753   of
13643   to
12799   that
12560   in
10263   he
9840    shall
8987    unto
8836    for
```

> **NOTE**
>
> If you are a seasoned Unix programmer, you are probably familiar with a wide variety of command-line tools (for example, `egrep`) that are built into your operating system and that are probably preferable to building your own from scratch. Still, it's good to know you can if you need to.

Reading Files

You can also explicitly read from and write to files directly in your code. Python makes working with files pretty simple.

Chapter 10. Working with Data

> Experts often possess more data than judgment.
>
> Colin Powell

Working with data is both an art and a science. We've mostly been talking about the science part, but in this chapter we'll look at some of the art.

Exploring Your Data

After you've identified the questions you're trying to answer and have gotten your hands on some data, you might be tempted to dive in and immediately start building models and getting answers. But you should resist this urge. Your first step should be to *explore* your data.

Exploring One-Dimensional Data

The simplest case is when you have a one-dimensional data set, which is just a collection of numbers. For example, these could be the daily average number of minutes each user spends on your site, the number of times each of a collection of data science tutorial videos was watched, or the number of pages of each of the data science books in your data science library.

An obvious first step is to compute a few summary statistics. You'd like to know how many data points you have, the smallest, the largest, the mean, and the standard deviation.

But even these don't necessarily give you a great understanding. A good next step is to create a histogram, in which you group your data into discrete *buckets* and count how many points fall into each bucket:

```
def bucketize(point, bucket_size):
    """floor the point to the next lower multiple of bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points, bucket_size):
    """buckets the points and counts how many in each bucket"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points, bucket_size, title=""):
    histogram = make_histogram(points, bucket_size)
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
    plt.title(title)
    plt.show()
```

For example, consider the two following sets of data:

```
random.seed(0)

# uniform between -100 and 100
uniform = [200 * random.random() - 100 for _ in range(10000)]

# normal distribution with mean 0, standard deviation 57
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(10000)]
```

Both have means close to 0 and standard deviations close to 58. However, they have very different distributions. Figure 10-1 shows the distribution of `uniform`:

```
plot_histogram(uniform, 10, "Uniform Histogram")
```

while Figure 10-2 shows the distribution of `normal`:

```
plot_histogram(normal, 10, "Normal Histogram")
```

In this case, both distributions had pretty different `max` and `min`, but even knowing that wouldn't have been sufficient to understand *how* they differed.

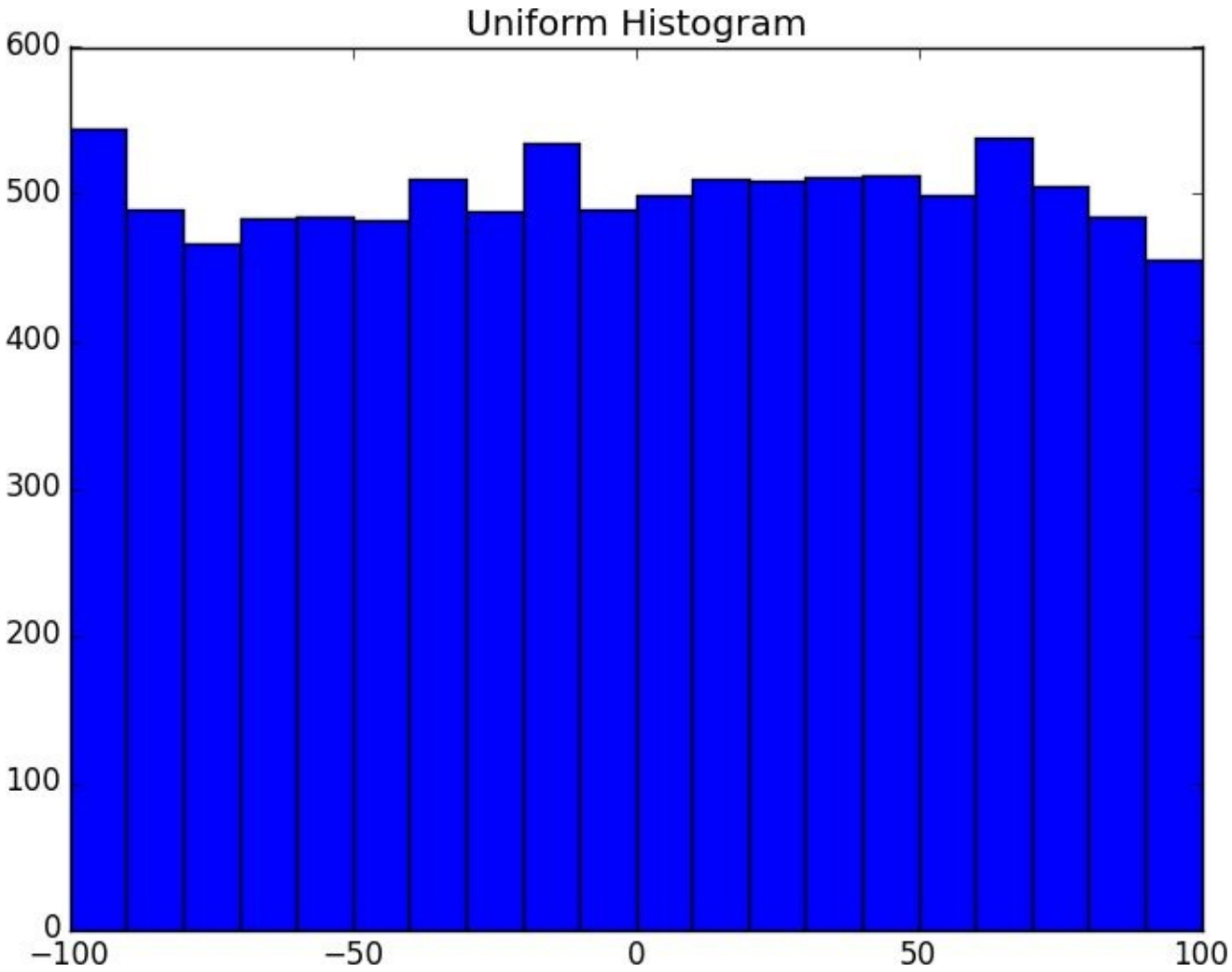

Figure 10-1. Histogram of `uniform`

Chapter 11. Machine Learning

> I am always ready to learn although I do not always like being taught.
>
> Winston Churchill

Many people imagine that data science is mostly machine learning and that data scientists mostly build and train and tweak machine-learning models all day long. (Then again, many of those people don't actually know what machine learning *is*.) In fact, data science is mostly turning business problems into data problems and collecting data and understanding data and cleaning data and formatting data, after which machine learning is almost an afterthought. Even so, it's an interesting and essential afterthought that you pretty much have to know about in order to do data science.

Modeling

Before we can talk about machine learning we need to talk about *models*.

What is a model? It's simply a specification of a mathematical (or probabilistic) relationship that exists between different variables.

For instance, if you're trying to raise money for your social networking site, you might build a *business model* (likely in a spreadsheet) that takes inputs like "number of users" and "ad revenue per user" and "number of employees" and outputs your annual profit for the next several years. A cookbook recipe entails a model that relates inputs like "number of eaters" and "hungriness" to quantities of ingredients needed. And if you've ever watched poker on television, you know that they estimate each player's "win probability" in real time based on a model that takes into account the cards that have been revealed so far and the distribution of cards in the deck.

The business model is probably based on simple mathematical relationships: profit is revenue minus expenses, revenue is units sold times average price, and so on. The recipe model is probably based on trial and error — someone went in a kitchen and tried different combinations of ingredients until they found one they liked. And the poker model is based on probability theory, the rules of poker, and some reasonably innocuous assumptions about the random process by which cards are dealt.

What Is Machine Learning?

Everyone has her own exact definition, but we'll use *machine learning* to refer to creating and using models that are *learned from data*. In other contexts this might be called *predictive modeling* or *data mining*, but we will stick with machine learning. Typically, our goal will be to use existing data to develop models that we can use to *predict* various outcomes for new data, such as:

- Predicting whether an email message is spam or not
- Predicting whether a credit card transaction is fraudulent
- Predicting which advertisement a shopper is most likely to click on
- Predicting which football team is going to win the Super Bowl

We'll look at both *supervised* models (in which there is a set of data labeled with the correct answers to learn from), and *unsupervised* models (in which there are no such labels). There are various other types like *semisupervised* (in which only some of the data are labeled) and *online* (in which the model needs to continuously adjust to newly arriving data) that we won't cover in this book.

Now, in even the simplest situation there are entire universes of models that might describe the relationship we're interested in. In most cases we will ourselves choose a *parameterized* family of models and then use data to learn parameters that are in some way optimal.

For instance, we might assume that a person's height is (roughly) a linear function of his weight and then use data to learn what that linear function is. Or we might assume that a decision tree is a good way to diagnose what diseases our patients have and then use data to learn the "optimal" such tree. Throughout the rest of the book we'll be investigating different families of models that we can learn.

But before we can do that, we need to better understand the fundamentals of machine learning. For the rest of the chapter, we'll discuss some of those basic concepts, before we move on to the models themselves.

Overfitting and Underfitting

A common danger in machine learning is *overfitting* — producing a model that performs well on the data you train it on but that generalizes poorly to any new data. This could involve learning *noise* in the data. Or it could involve learning to identify specific inputs rather than whatever factors are actually predictive for the desired output.

The other side of this is *underfitting*, producing a model that doesn't perform well even on the training data, although typically when this happens you decide your model isn't good enough and keep looking for a better one.

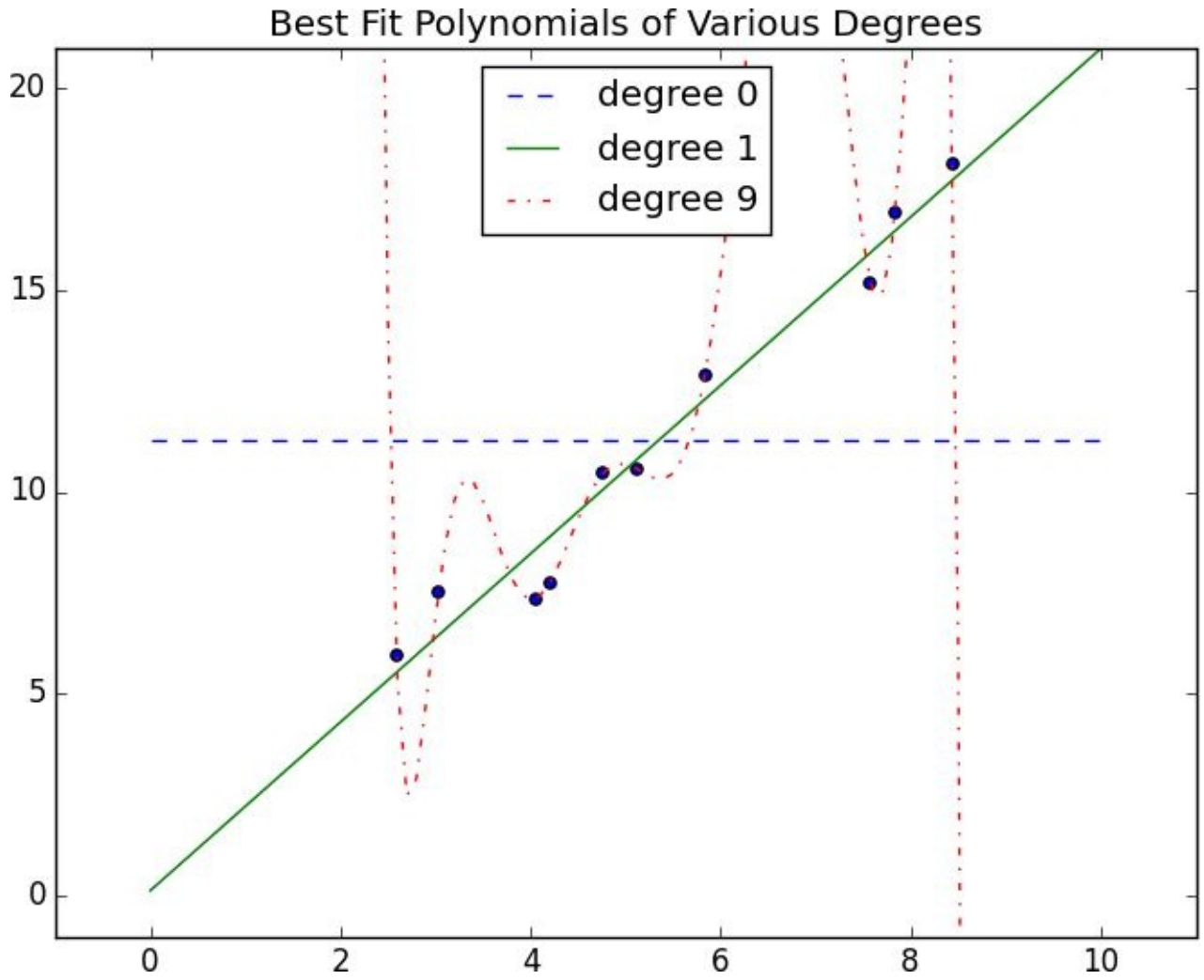

Figure 11-1. Overfitting and underfitting

In Figure 11-1, I've fit three polynomials to a sample of data. (Don't worry about how; we'll get to that in later chapters.)

The horizontal line shows the best fit degree 0 (i.e., constant) polynomial. It severely *underfits* the training data. The best fit degree 9 (i.e., 10-parameter) polynomial goes through every training data point exactly, but it very severely *overfits* — if we were to pick a few more data points it would quite likely miss them by a lot. And the degree 1 line strikes a nice balance — it's pretty close to every point, and (if these data are representative) the line will likely be close to new data points as well.

Clearly models that are too complex lead to overfitting and don't generalize well beyond

Chapter 12. k-Nearest Neighbors

> If you want to annoy your neighbors, tell the truth about them.
>
> Pietro Aretino

Imagine that you're trying to predict how I'm going to vote in the next presidential election. If you know nothing else about me (and if you have the data), one sensible approach is to look at how my *neighbors* are planning to vote. Living in downtown Seattle, as I do, my neighbors are invariably planning to vote for the Democratic candidate, which suggests that "Democratic candidate" is a good guess for me as well.

Now imagine you know more about me than just geography — perhaps you know my age, my income, how many kids I have, and so on. To the extent my behavior is influenced (or characterized) by those things, looking just at my neighbors who are close to me among all those dimensions seems likely to be an even better predictor than looking at all my neighbors. This is the idea behind *nearest neighbors classification*.

The Model

Nearest neighbors is one of the simplest predictive models there is. It makes no mathematical assumptions, and it doesn't require any sort of heavy machinery. The only things it requires are:

- Some notion of distance
- An assumption that points that are close to one another are similar

Most of the techniques we'll look at in this book look at the data set as a whole in order to learn patterns in the data. Nearest neighbors, on the other hand, quite consciously neglects a lot of information, since the prediction for each new point depends only on the handful of points closest to it.

What's more, nearest neighbors is probably not going to help you understand the drivers of whatever phenomenon you're looking at. Predicting my votes based on my neighbors' votes doesn't tell you much about what causes me to vote the way I do, whereas some alternative model that predicted my vote based on (say) my income and marital status very well might.

In the general situation, we have some data points and we have a corresponding set of labels. The labels could be `True` and `False`, indicating whether each input satisfies some condition like "is spam?" or "is poisonous?" or "would be enjoyable to watch?" Or they could be categories, like movie ratings (G, PG, PG-13, R, NC-17). Or they could be the names of presidential candidates. Or they could be favorite programming languages.

In our case, the data points will be vectors, which means that we can use the `distance` function from Chapter 4.

Let's say we've picked a number *k* like 3 or 5. Then when we want to classify some new data point, we find the `k` nearest labeled points and let them vote on the new output.

To do this, we'll need a function that counts votes. One possibility is:

```
def raw_majority_vote(labels):
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner
```

But this doesn't do anything intelligent with ties. For example, imagine we're rating movies and the five nearest movies are rated G, G, PG, PG, and R. Then G has two votes and PG also has two votes. In that case, we have several options:

- Pick one of the winners at random.
- Weight the votes by distance and pick the weighted winner.
- Reduce *k* until we find a unique winner.

We'll implement the third:

```
def majority_vote(labels):
    """assumes that labels are ordered from nearest to farthest"""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])

    if num_winners == 1:
        return winner                     # unique winner, so return it
    else:
        return majority_vote(labels[:-1]) # try again without the farthest
```

This approach is sure to work eventually, since in the worst case we go all the way down to just one label, at which point that one label wins.

With this function it's easy to create a classifier:

```
def knn_classify(k, labeled_points, new_point):
    """each labeled point should be a pair (point, label)"""

    # order the labeled points from nearest to farthest
    by_distance = sorted(labeled_points,
                         key=lambda (point, _): distance(point, new_point))

    # find the labels for the k closest
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # and let them vote
    return majority_vote(k_nearest_labels)
```

Let's take a look at how this works.

Example: Favorite Languages

The results of the first DataSciencester user survey are back, and we've found the preferred programming languages of our users in a number of large cities:

```
# each entry is ([longitude, latitude], favorite_language)

cities = [([-122.3 , 47.53], "Python"),  # Seattle
          ([ -96.85, 32.85], "Java"),    # Austin
          ([ -89.33, 43.13], "R"),       # Madison
          # ... and so on
]
```

The VP of Community Engagement wants to know if we can use these results to predict the favorite programming languages for places that weren't part of our survey.

As usual, a good first step is plotting the data (Figure 12-1):

```
# key is language, value is pair (longitudes, latitudes)
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

# we want each language to have a different marker and color
markers = { "Java" : "o", "Python" : "s", "R" : "^" }
colors  = { "Java" : "r", "Python" : "b", "R" : "g" }

for (longitude, latitude), language in cities:
    plots[language][0].append(longitude)
    plots[language][1].append(latitude)

# create a scatter series for each language
for language, (x, y) in plots.iteritems():
    plt.scatter(x, y, color=colors[language], marker=markers[language],
                      label=language, zorder=10)

plot_state_borders(plt)      # pretend we have a function that does this

plt.legend(loc=0)            # let matplotlib choose the location
plt.axis([-130,-60,20,55])   # set the axes

plt.title("Favorite Programming Languages")
plt.show()
```

Chapter 13. Naive Bayes

> It is well for the heart to be naive and for the mind not to be.
>
> Anatole France

A social network isn't much good if people can't network. Accordingly, DataSciencester has a popular feature that allows members to send messages to other members. And while most of your members are responsible citizens who send only well-received "how's it going?" messages, a few miscreants persistently spam other members about get-rich schemes, no-prescription-required pharmaceuticals, and for-profit data science credentialing programs. Your users have begun to complain, and so the VP of Messaging has asked you to use data science to figure out a way to filter out these spam messages.

A Really Dumb Spam Filter

Imagine a "universe" that consists of receiving a message chosen randomly from all possible messages. Let *S* be the event "the message is spam" and *V* be the event "the message contains the word *viagra*." Then Bayes's Theorem tells us that the probability that the message is spam conditional on containing the word *viagra* is:

$$P(S \mid V) = [P(V \mid S)P(S)] / [P(V \mid S)P(S) + P(V \mid \neg S)P(\neg S)]$$

The numerator is the probability that a message is spam *and* contains *viagra*, while the denominator is just the probability that a message contains *viagra*. Hence you can think of this calculation as simply representing the proportion of *viagra* messages that are spam.

If we have a large collection of messages we know are spam, and a large collection of messages we know are not spam, then we can easily estimate $P(V \mid S)$ and $P(V \mid \neg S)$. If we further assume that any message is equally likely to be spam or not-spam (so that $P(S) = P(\neg S) = 0.5$), then:

$$P(S \mid V) = P(V \mid S) / [P(V \mid S) + P(V \mid \neg S)]$$

For example, if 50% of spam messages have the word *viagra*, but only 1% of nonspam messages do, then the probability that any given *viagra*-containing email is spam is:

$$0.5 / (0.5 + 0.01) = 98\ \%$$

A More Sophisticated Spam Filter

Imagine now that we have a vocabulary of many words $w_1, \ldots, w_n$. To move this into the realm of probability theory, we'll write X_i for the event "a message contains the word w_i." Also imagine that (through some unspecified-at-this-point process) we've come up with an estimate $P(X_i \mid S)$ for the probability that a spam message contains the *i*th word, and a similar estimate $P(X_i \mid \neg S)$ for the probability that a nonspam message contains the *i*th word.

The key to Naive Bayes is making the (big) assumption that the presences (or absences) of each word are independent of one another, conditional on a message being spam or not. Intuitively, this assumption means that knowing whether a certain spam message contains the word "viagra" gives you no information about whether that same message contains the word "rolex." In math terms, this means that:

$$P(X_1 = x_1, \ldots, X_n = x_n \mid S) = P(X_1 = x_1 \mid S) \times \cdots \times P(X_n = x_n \mid S)$$

This is an extreme assumption. (There's a reason the technique has "naive" in its name.) Imagine that our vocabulary consists *only* of the words "viagra" and "rolex," and that half of all spam messages are for "cheap viagra" and that the other half are for "authentic rolex." In this case, the Naive Bayes estimate that a spam message contains both "viagra" and "rolex" is:

$$P(X_1 = 1, X_2 = 1 \mid S) = P(X_1 = 1 \mid S)P(X_2 = 1 \mid S) = .5 \times .5 = .25$$

since we've assumed away the knowledge that "viagra" and "rolex" actually never occur together. Despite the unrealisticness of this assumption, this model often performs well and is used in actual spam filters.

The same Bayes's Theorem reasoning we used for our "viagra-only" spam filter tells us that we can calculate the probability a message is spam using the equation:

$$P(S \mid X = x) = P(X = x \mid S) / [P(X = x \mid S) + P(X = x \mid \neg S)]$$

The Naive Bayes assumption allows us to compute each of the probabilities on the right simply by multiplying together the individual probability estimates for each vocabulary word.

In practice, you usually want to avoid multiplying lots of probabilities together, to avoid a problem called *underflow*, in which computers don't deal well with floating-point numbers that are too close to zero. Recalling from algebra that $\log(ab) = \log a + \log b$ and that $\exp(\log x) = x$, we usually compute $p_1 * \cdots * p_n$ as the equivalent (but floating-point-friendlier):

$$\exp(\log(p_1) + \cdots + \log(p_n))$$

The only challenge left is coming up with estimates for $P(X_i \mid S)$ and $P(X_i \mid \neg S)$, the probabilities that a spam message (or nonspam message) contains the word w_i. If we have a fair number of "training" messages labeled as spam and not-spam, an obvious first try is to estimate $P(X_i \mid S)$ simply as the fraction of spam messages containing word w_i.

This causes a big problem, though. Imagine that in our training set the vocabulary word "data" only occurs in nonspam messages. Then we'd estimate $P(\text{"data"} \mid S) = 0$. The result is that our Naive Bayes classifier would always assign spam probability 0 to *any* message containing the word "data," even a message like "data on cheap viagra and authentic rolex watches." To avoid this problem, we usually use some kind of smoothing.

In particular, we'll choose a *pseudocount* — *k* — and estimate the probability of seeing the *i*th word in a spam as:

$$P(X_i \mid S) = (k + \text{number of spams containing } w_i) / (2k + \text{number of spams})$$

Similarly for $P(X_i \mid \neg S)$. That is, when computing the spam probabilities for the *i*th word, we assume we also saw *k* additional spams containing the word and *k* additional spams not containing the word.

For example, if "data" occurs in 0/98 spam documents, and if *k* is 1, we estimate $P(\text{"data"} \mid S)$ as 1/100 = 0.01, which allows our classifier to still assign some nonzero spam probability to messages that contain the word "data."

Chapter 14. Simple Linear Regression

> Art, like morality, consists in drawing the line somewhere.
>
> G. K. Chesterton

In Chapter 5, we used the `correlation` function to measure the strength of the linear relationship between two variables. For most applications, knowing that such a linear relationship exists isn't enough. We'll want to be able to understand the nature of the relationship. This is where we'll use simple linear regression.

The Model

Recall that we were investigating the relationship between a DataSciencester user's number of friends and the amount of time he spent on the site each day. Let's assume that you've convinced yourself that having more friends *causes* people to spend more time on the site, rather than one of the alternative explanations we discussed.

The VP of Engagement asks you to build a model describing this relationship. Since you found a pretty strong linear relationship, a natural place to start is a linear model.

In particular, you hypothesize that there are constants α (alpha) and β (beta) such that:

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

where y_i is the number of minutes user *i* spends on the site daily, x_i is the number of friends user *i* has, and ε_i is a (hopefully small) error term representing the fact that there are other factors not accounted for by this simple model.

Assuming we've determined such an `alpha` and `beta`, then we make predictions simply with:

```
def predict(alpha, beta, x_i):
    return beta * x_i + alpha
```

How do we choose `alpha` and `beta`? Well, any choice of `alpha` and `beta` gives us a predicted output for each input `x_i`. Since we know the actual output `y_i` we can compute the error for each pair:

```
def error(alpha, beta, x_i, y_i):
    """the error from predicting beta * x_i + alpha
    when the actual value is y_i"""
    return y_i - predict(alpha, beta, x_i)
```

What we'd really like to know is the total error over the entire data set. But we don't want to just add the errors — if the prediction for `x_1` is too high and the prediction for `x_2` is too low, the errors may just cancel out.

So instead we add up the *squared* errors:

```
def sum_of_squared_errors(alpha, beta, x, y):
    return sum(error(alpha, beta, x_i, y_i) ** 2
               for x_i, y_i in zip(x, y))
```

The *least squares solution* is to choose the `alpha` and `beta` that make `sum_of_squared_errors` as small as possible.

Using calculus (or tedious algebra), the error-minimizing alpha and beta are given by:

```
def least_squares_fit(x, y):
```

```
    """given training values for x and y,
    find the least-squares values of alpha and beta"""
    beta = correlation(x, y) * standard_deviation(y) / standard_deviation(x)
    alpha = mean(y) - beta * mean(x)
    return alpha, beta
```

Without going through the exact mathematics, let's think about why this might be a reasonable solution. The choice of `alpha` simply says that when we see the average value of the independent variable `x`, we predict the average value of the dependent variable `y`.

The choice of `beta` means that when the input value increases by `standard_deviation(x)`, the prediction increases by `correlation(x, y) * standard_deviation(y)`. In the case when `x` and `y` are perfectly correlated, a one standard deviation increase in `x` results in a one-standard-deviation-of-`y` increase in the prediction. When they're perfectly anticorrelated, the increase in `x` results in a *decrease* in the prediction. And when the correlation is zero, `beta` is zero, which means that changes in `x` don't affect the prediction at all.

It's easy to apply this to the outlierless data from Chapter 5:

```
alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)
```

This gives values of alpha = 22.95 and beta = 0.903. So our model says that we expect a user with `n` friends to spend `22.95 + n * 0.903` minutes on the site each day. That is, we predict that a user with no friends on DataSciencester would still spend about 23 minutes a day on the site. And for each additional friend, we expect a user to spend almost a minute more on the site each day.

In Figure 14-1, we plot the prediction line to get a sense of how well the model fits the observed data.

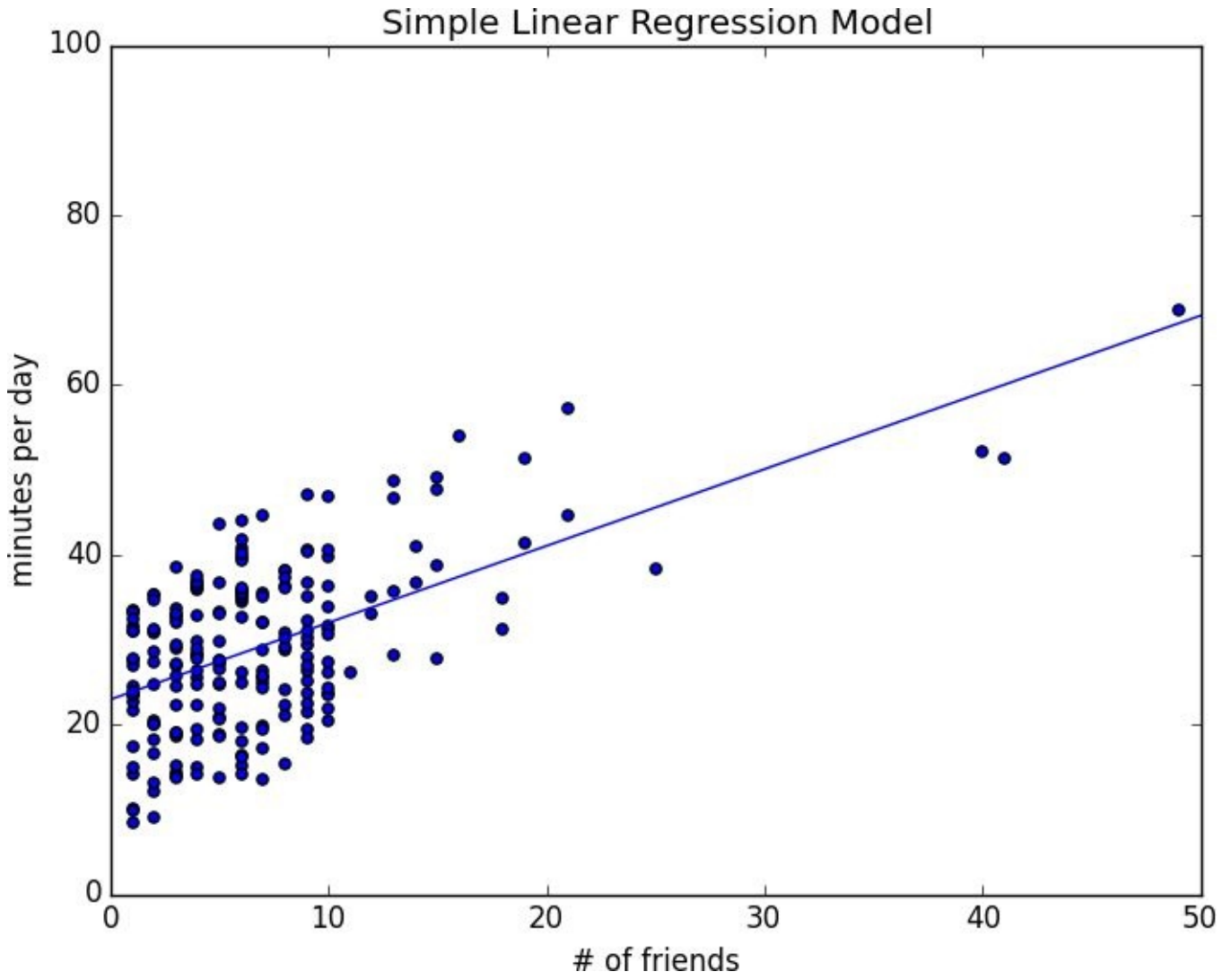

Figure 14-1. Our simple linear model

Of course, we need a better way to figure out how well we've fit the data than staring at the graph. A common measure is the *coefficient of determination* (or *R-squared*), which measures the fraction of the total variation in the dependent variable that is captured by the model:

```
def total_sum_of_squares(y):
    """the total squared variation of y_i's from their mean"""
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha, beta, x, y):
    """the fraction of variation in y captured by the model, which equals
    1 - the fraction of variation in y not captured by the model"""

    return 1.0 - (sum_of_squared_errors(alpha, beta, x, y) /
                  total_sum_of_squares(y))

r_squared(alpha, beta, num_friends_good, daily_minutes_good)      # 0.329
```

Now, we chose the `alpha` and `beta` that minimized the sum of the squared prediction errors. One linear model we could have chosen is "always predict `mean(y)`" (corresponding to `alpha = mean(y)` and `beta = 0`), whose sum of squared errors exactly equals its total sum of squares. This means an R-squared of zero, which indicates a model that (obviously, in this case) performs no better than just predicting the mean.

Chapter 15. Multiple Regression

> I don't look at a problem and put variables in there that don't affect it.
>
> Bill Parcells

Although the VP is pretty impressed with your predictive model, she thinks you can do better. To that end, you've collected additional data: for each of your users, you know how many hours he works each day, and whether he has a PhD. You'd like to use this additional data to improve your model.

Accordingly, you hypothesize a linear model with more independent variables:

$$\text{minutes} = \alpha + \beta_1 \,\text{friends} + \beta_2 \,\text{work hours} + \beta_3 \,\text{phd} + \varepsilon$$

Obviously, whether a user has a PhD is not a number, but — as we mentioned in Chapter 11 — we can introduce a *dummy variable* that equals 1 for users with PhDs and 0 for users without, after which it's just as numeric as the other variables.

The Model

Recall that in Chapter 14 we fit a model of the form:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

Now imagine that each input x_i is not a single number but rather a vector of k numbers $x_{i1}, \ldots, x_{ik}$. The multiple regression model assumes that:

$$y_i = \alpha + \beta_1 x_{i1} + \ldots + \beta_k x_{ik} + \varepsilon_i$$

In multiple regression the vector of parameters is usually called β. We'll want this to include the constant term as well, which we can achieve by adding a column of ones to our data:

```
beta = [alpha, beta_1, ..., beta_k]
```

and:

```
x_i = [1, x_i1, ..., x_ik]
```

Then our model is just:

```
def predict(x_i, beta):
    """assumes that the first element of each x_i is 1"""
    return dot(x_i, beta)
```

In this particular case, our independent variable `x` will be a list of vectors, each of which looks like this:

```
[1,    # constant term
 49,   # number of friends
 4,    # work hours per day
 0]    # doesn't have PhD
```

Further Assumptions of the Least Squares Model

There are a couple of further assumptions that are required for this model (and our solution) to make sense.

The first is that the columns of *x* are *linearly independent* — that there's no way to write any one as a weighted sum of some of the others. If this assumption fails, it's impossible to estimate `beta`. To see this in an extreme case, imagine we had an extra field `num_acquaintances` in our data that for every user was exactly equal to `num_friends`.

Then, starting with any `beta`, if we add *any* amount to the `num_friends` coefficient and subtract that same amount from the `num_acquaintances` coefficient, the model's predictions will remain unchanged. Which means that there's no way to find *the* coefficient for `num_friends`. (Usually violations of this assumption won't be so obvious.)

The second important assumption is that the columns of *x* are all uncorrelated with the errors ε. If this fails to be the case, our estimates of `beta` will be systematically wrong.

For instance, in Chapter 14, we built a model that predicted that each additional friend was associated with an extra 0.90 daily minutes on the site.

Imagine that it's also the case that:

- People who work more hours spend less time on the site.
- People with more friends tend to work more hours.

That is, imagine that the "actual" model is:

$$\text{minutes} = \alpha + \beta_1 \text{ friends} + \beta_2 \text{ work hours} + \varepsilon$$

and that work hours and friends are positively correlated. In that case, when we minimize the errors of the single variable model:

$$\text{minutes} = \alpha + \beta_1 \text{ friends} + \varepsilon$$

we will underestimate β_1.

Think about what would happen if we made predictions using the single variable model with the "actual" value of β_1. (That is, the value that arises from minimizing the errors of what we called the "actual" model.) The predictions would tend to be too small for users who work many hours and too large for users who work few hours, because $\beta_2 > 0$ and we "forgot" to include it. Because work hours is positively correlated with number of friends, this means the predictions tend to be too small for users with many friends and too large for users with few friends.

The result of this is that we can reduce the errors (in the single-variable model) by

decreasing our estimate of β_1, which means that the error-minimizing β_1 is smaller than the “actual” value. That is, in this case the single-variable least squares solution is biased to underestimate β_1. And, in general, whenever the independent variables are correlated with the errors like this, our least squares solution will give us a biased estimate of β.

Chapter 16. Logistic Regression

> A lot of people say there's a fine line between genius and insanity. I don't think there's a fine line, I actually think there's a yawning gulf.
>
> Bill Bailey

In Chapter 1, we briefly looked at the problem of trying to predict which DataSciencester users paid for premium accounts. Here we'll revisit that problem.

The Problem

We have an anonymized data set of about 200 users, containing each user's salary, her years of experience as a data scientist, and whether she paid for a premium account (Figure 16-1). As is usual with categorical variables, we represent the dependent variable as either 0 (no premium account) or 1 (premium account).

As usual, our data is in a matrix where each row is a list `[experience, salary, paid_account]`. Let's turn it into the format we need:

```
x = [[1] + row[:2] for row in data]  # each element is [1, experience, salary]
y = [row[2] for row in data]         # each element is paid_account
```

An obvious first attempt is to use linear regression and find the best model:

$$\text{paid account} = \beta_0 + \beta_1 \text{experience} + \beta_2 \text{salary} + \varepsilon$$

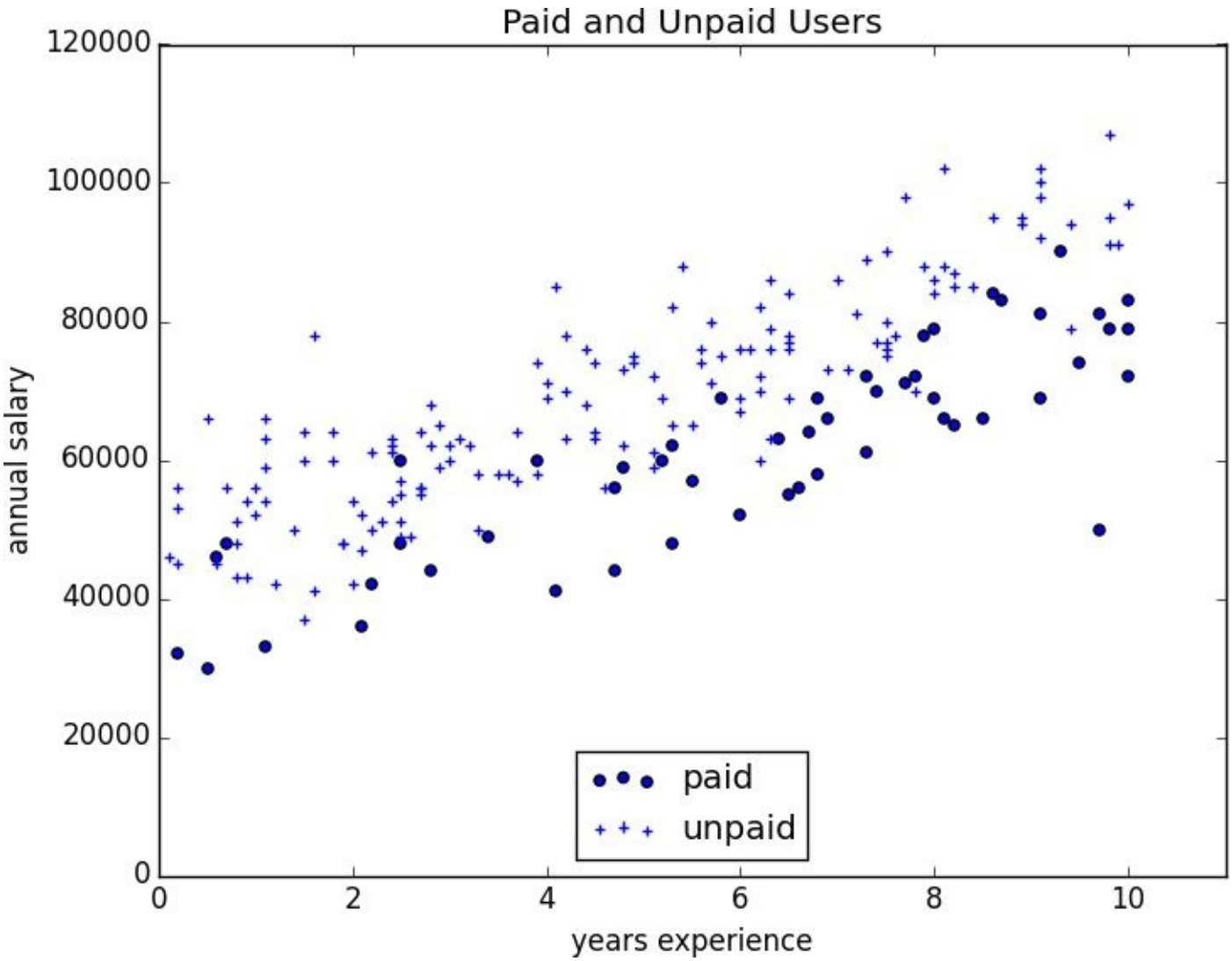

Figure 16-1. Paid and unpaid users

And certainly, there's nothing preventing us from modeling the problem this way. The results are shown in Figure 16-2:

```
rescaled_x = rescale(x)
```

```
beta = estimate_beta(rescaled_x, y)  # [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_x]

plt.scatter(predictions, y)
plt.xlabel("predicted")
plt.ylabel("actual")
plt.show()
```

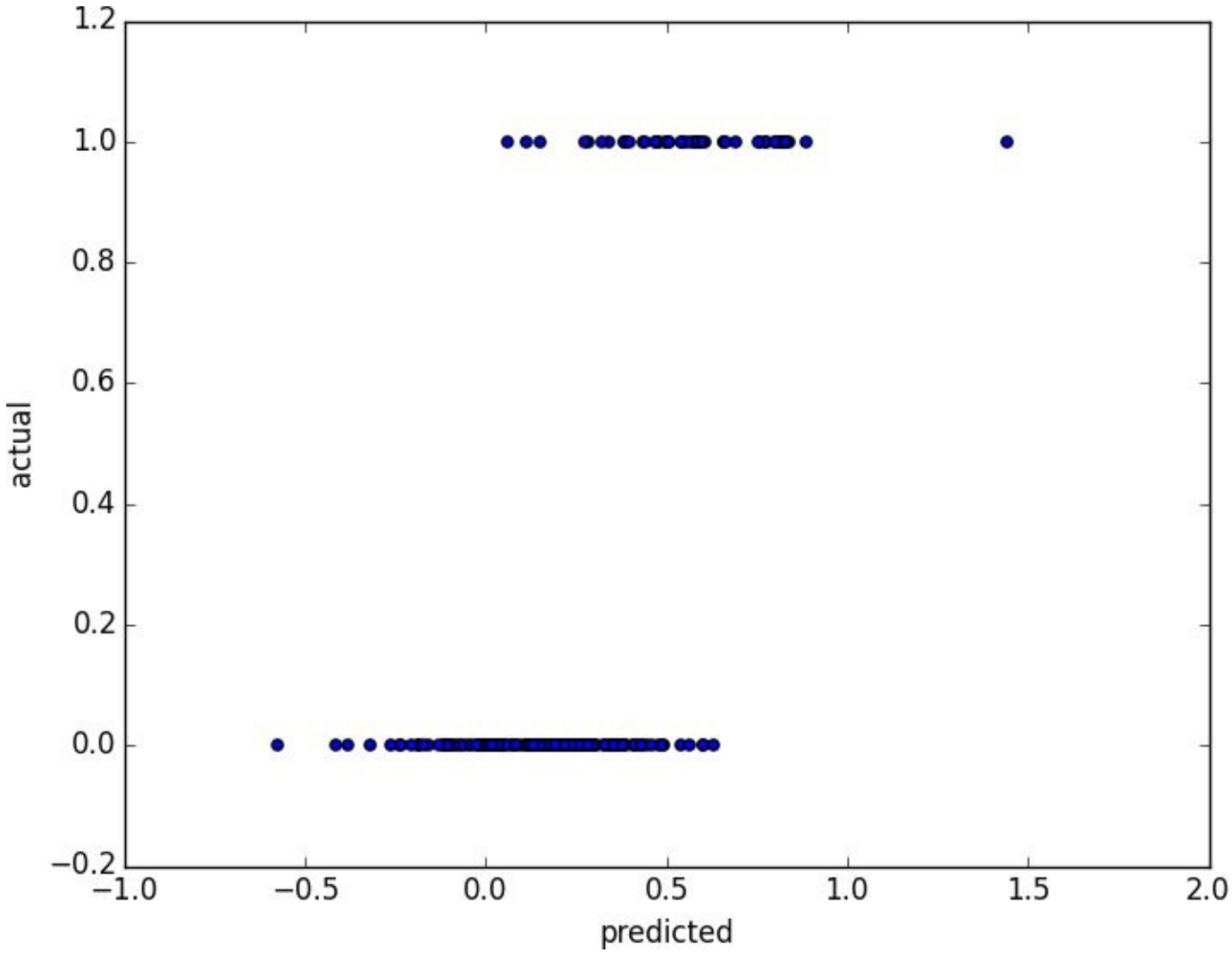

Figure 16-2. Using linear regression to predict premium accounts

But this approach leads to a couple of immediate problems:

- We'd like for our predicted outputs to be 0 or 1, to indicate class membership. It's fine if they're between 0 and 1, since we can interpret these as probabilities — an output of 0.25 could mean 25% chance of being a paid member. But the outputs of the linear model can be huge positive numbers or even negative numbers, which it's not clear how to interpret. Indeed, here a lot of our predictions were negative.

- The linear regression model assumed that the errors were uncorrelated with the columns of x. But here, the regression coefficent for `experience` is 0.43, indicating that more experience leads to a greater likelihood of a premium account. This means that our model outputs very large values for people with lots of experience. But we know that the actual values must be at most 1, which means that necessarily very large outputs (and therefore very large values of `experience`) correspond to very large negative values of the error term. Because this is the case, our estimate of `beta` is biased.

What we’d like instead is for large positive values of `dot(x_i, beta)` to correspond to probabilities close to 1, and for large negative values to correspond to probabilities close to 0. We can accomplish this by applying another function to the result.

Chapter 17. Decision Trees

> A tree is an incomprehensible mystery.
>
> Jim Woodring

DataSciencester's VP of Talent has interviewed a number of job candidates from the site, with varying degrees of success. He's collected a data set consisting of several (qualitative) attributes of each candidate, as well as whether that candidate interviewed well or poorly. Could you, he asks, use this data to build a model identifying which candidates will interview well, so that he doesn't have to waste time conducting interviews?

This seems like a good fit for a *decision tree*, another predictive modeling tool in the data scientist's kit.

What Is a Decision Tree?

A decision tree uses a tree structure to represent a number of possible *decision paths* and an outcome for each path.

If you have ever played the game Twenty Questions, then it turns out you are familiar with decision trees. For example:

- “I am thinking of an animal.”
- “Does it have more than five legs?”
- “No.”
- “Is it delicious?”
- “No.”
- “Does it appear on the back of the Australian five-cent coin?”
- “Yes.”
- “Is it an echidna?”
- “Yes, it is!”

This corresponds to the path:

“Not more than 5 legs” → “Not delicious” → “On the 5-cent coin” → “Echidna!”

in an idiosyncratic (and not very comprehensive) “guess the animal” decision tree (Figure 17-1).

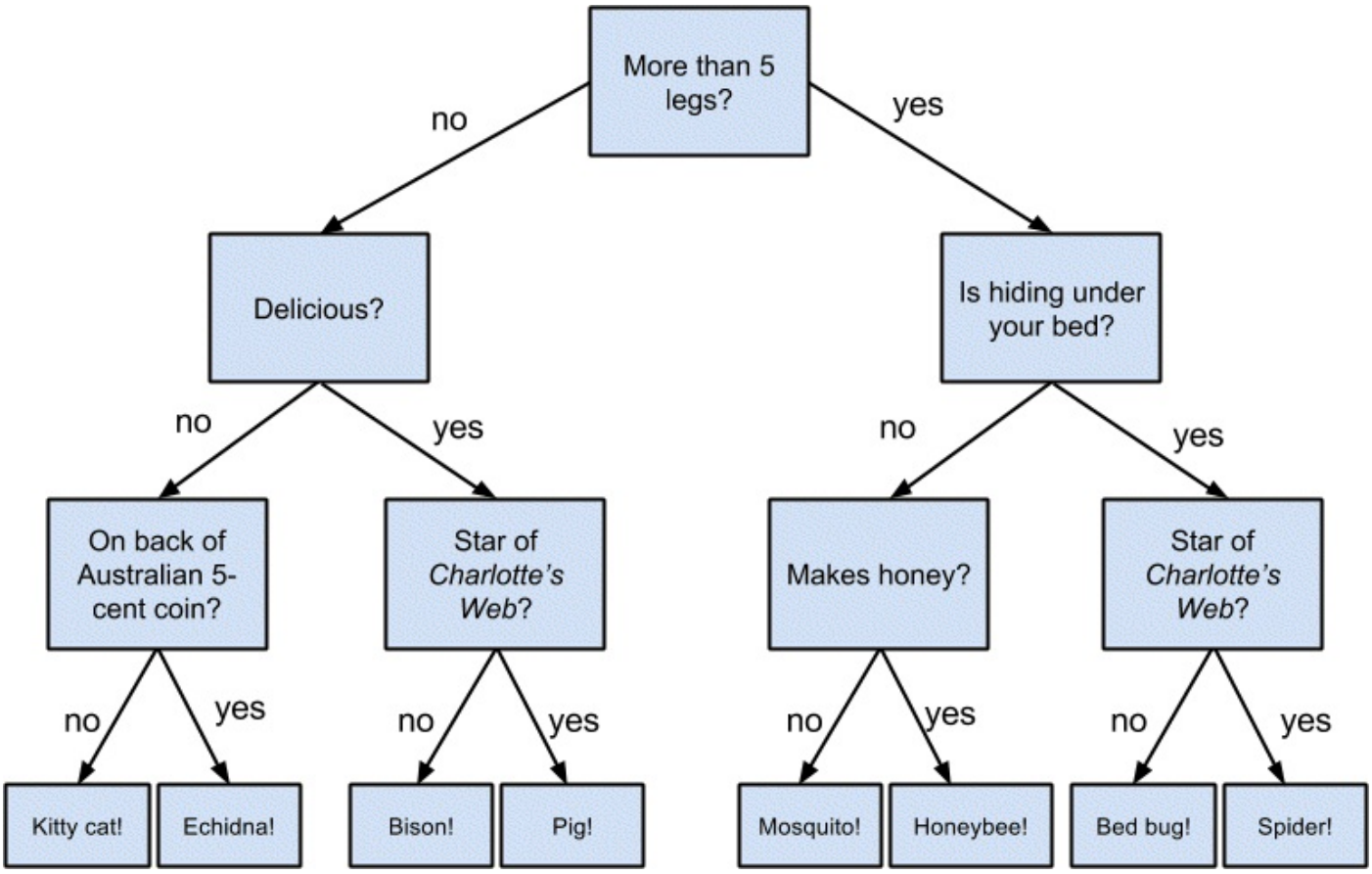

Figure 17-1. A "guess the animal" decision tree

Decision trees have a lot to recommend them. They're very easy to understand and interpret, and the process by which they reach a prediction is completely transparent. Unlike the other models we've looked at so far, decision trees can easily handle a mix of numeric (e.g., number of legs) and categorical (e.g., delicious/not delicious) attributes and can even classify data for which attributes are missing.

At the same time, finding an "optimal" decision tree for a set of training data is computationally a very hard problem. (We will get around this by trying to build a good-enough tree rather than an optimal one, although for large data sets this can still be a lot of work.) More important, it is very easy (and very bad) to build decision trees that are *overfitted* to the training data, and that don't generalize well to unseen data. We'll look at ways to address this.

Most people divide decision trees into *classification trees* (which produce categorical outputs) and *regression trees* (which produce numeric outputs). In this chapter, we'll focus on classification trees, and we'll work through the ID3 algorithm for learning a decision tree from a set of labeled data, which should help us understand how decision trees actually work. To make things simple, we'll restrict ourselves to problems with binary outputs like "should I hire this candidate?" or "should I show this website visitor advertisement A or advertisement B?" or "will eating this food I found in the office fridge make me sick?"

Entropy

In order to build a decision tree, we will need to decide what questions to ask and in what order. At each stage of the tree there are some possibilities we've eliminated and some that we haven't. After learning that an animal doesn't have more than five legs, we've eliminated the possibility that it's a grasshopper. We haven't eliminated the possibility that it's a duck. Every possible question partitions the remaining possibilities according to their answers.

Ideally, we'd like to choose questions whose answers give a lot of information about what our tree should predict. If there's a single yes/no question for which "yes" answers always correspond to `True` outputs and "no" answers to `False` outputs (or vice versa), this would be an awesome question to pick. Conversely, a yes/no question for which neither answer gives you much new information about what the prediction should be is probably not a good choice.

We capture this notion of "how much information" with *entropy*. You have probably heard this used to mean disorder. We use it to represent the uncertainty associated with data.

Imagine that we have a set S of data, each member of which is labeled as belonging to one of a finite number of classes $C_1, \ldots, C_n$. If all the data points belong to a single class, then there is no real uncertainty, which means we'd like there to be low entropy. If the data points are evenly spread across the classes, there is a lot of uncertainty and we'd like there to be high entropy.

In math terms, if p_i is the proportion of data labeled as class c_i, we define the entropy as:

$$H(S) = -p_1 \log_2 p_1 - \ldots - p_n \log_2 p_n$$

with the (standard) convention that $0 \log 0 = 0$.

Without worrying too much about the grisly details, each term $-p_i \log_2 p_i$ is non-negative and is close to zero precisely when p_i is either close to zero or close to one (Figure 17-2).

Chapter 18. Neural Networks

> I like nonsense; it wakes up the brain cells.
>
> Dr. Seuss

An *artificial neural network* (or neural network for short) is a predictive model motivated by the way the brain operates. Think of the brain as a collection of neurons wired together. Each neuron looks at the outputs of the other neurons that feed into it, does a calculation, and then either fires (if the calculation exceeds some threshhold) or doesn't (if it doesn't).

Accordingly, artificial neural networks consist of artificial neurons, which perform similar calculations over their inputs. Neural networks can solve a wide variety of problems like handwriting recognition and face detection, and they are used heavily in deep learning, one of the trendiest subfields of data science. However, most neural networks are "black boxes" — inspecting their details doesn't give you much understanding of *how* they're solving a problem. And large neural networks can be difficult to train. For most problems you'll encounter as a budding data scientist, they're probably not the right choice. Someday, when you're trying to build an artificial intelligence to bring about the Singularity, they very well might be.

Perceptrons

Pretty much the simplest neural network is the *perceptron*, which approximates a single neuron with *n* binary inputs. It computes a weighted sum of its inputs and "fires" if that weighted sum is zero or greater:

```
def step_function(x):
    return 1 if x >= 0 else 0

def perceptron_output(weights, bias, x):
    """returns 1 if the perceptron 'fires', 0 if not"""
    calculation = dot(weights, x) + bias
    return step_function(calculation)
```

The perceptron is simply distinguishing between the half spaces separated by the hyperplane of points `x` for which:

```
dot(weights,x) + bias == 0
```

With properly chosen weights, perceptrons can solve a number of simple problems (Figure 18-1). For example, we can create an *AND gate* (which returns 1 if both its inputs are 1 but returns 0 if one of its inputs is 0) with:

```
weights = [2, 2]
bias = -3
```

If both inputs are 1, the `calculation` equals 2 + 2 - 3 = 1, and the output is 1. If only one of the inputs is 1, the `calculation` equals 2 + 0 - 3 = -1, and the output is 0. And if both of the inputs are 0, the `calculation` equals -3, and the output is 0.

Similarly, we could build an *OR gate* with:

```
weights = [2, 2]
bias = -1
```

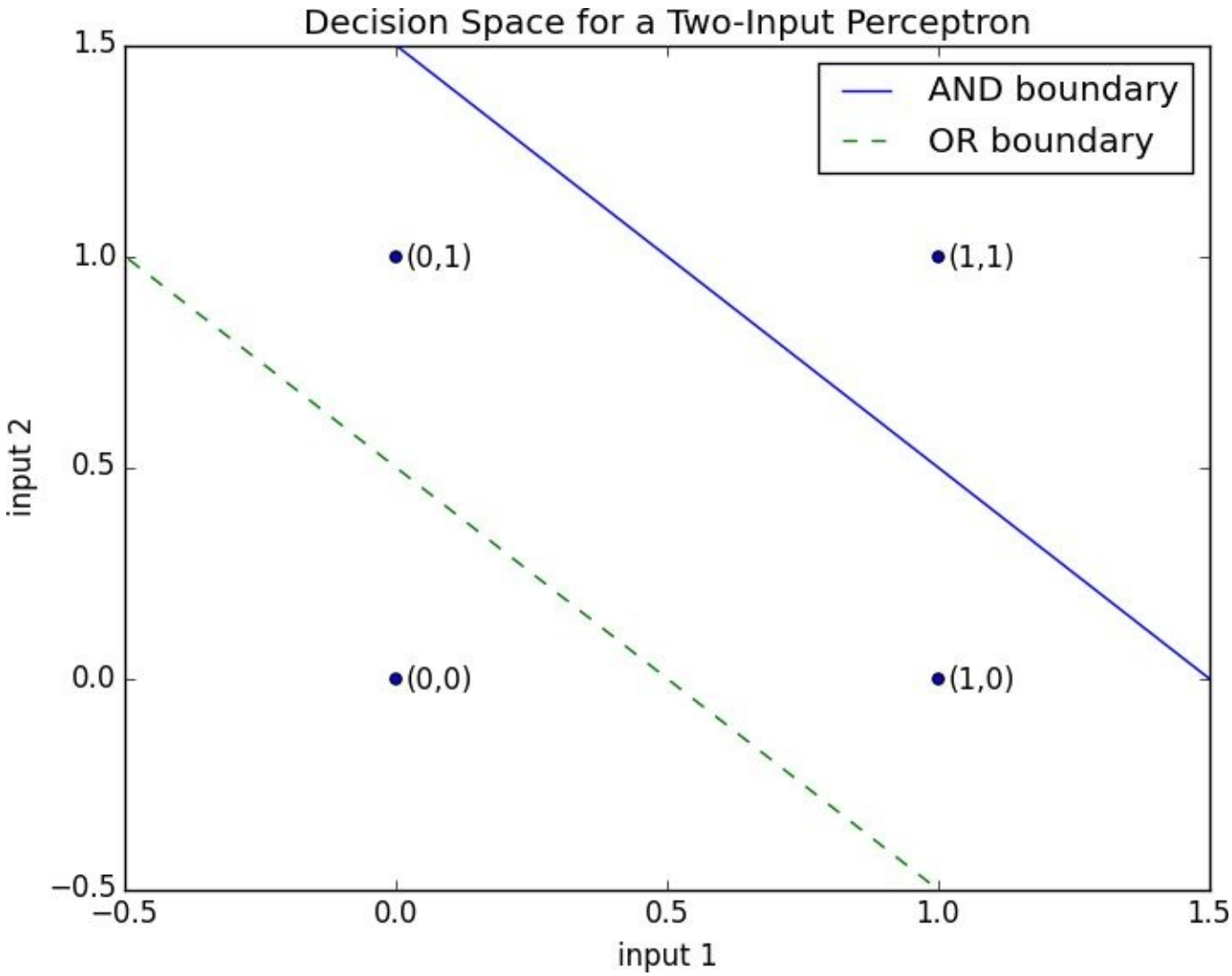

Figure 18-1. Decision space for a two-input perceptron

And we could build a *NOT gate* (which has one input and converts 1 to 0 and 0 to 1) with:

```
weights = [-2]
bias = 1
```

However, there are some problems that simply can't be solved by a single perceptron. For example, no matter how hard you try, you cannot use a perceptron to build an *XOR gate* that outputs 1 if exactly one of its inputs is 1 and 0 otherwise. This is where we start needing more-complicated neural networks.

Of course, you don't need to approximate a neuron in order to build a logic gate:

```
and_gate = min
or_gate = max
xor_gate = lambda x, y: 0 if x == y else 1
```

Like real neurons, artificial neurons start getting more interesting when you start connecting them together.

Feed-Forward Neural Networks

The topology of the brain is enormously complicated, so it's common to approximate it with an idealized *feed-forward* neural network that consists of discrete *layers* of neurons, each connected to the next. This typically entails an input layer (which receives inputs and feeds them forward unchanged), one or more "hidden layers" (each of which consists of neurons that take the outputs of the previous layer, performs some calculation, and passes the result to the next layer), and an output layer (which produces the final outputs).

Just like the perceptron, each (noninput) neuron has a weight corresponding to each of its inputs and a bias. To make our representation simpler, we'll add the bias to the end of our weights vector and give each neuron a *bias input* that always equals 1.

As with the perceptron, for each neuron we'll sum up the products of its inputs and its weights. But here, rather than outputting the `step_function` applied to that product, we'll output a smooth approximation of the step function. In particular, we'll use the `sigmoid` function (Figure 18-2):

```
def sigmoid(t):
    return 1 / (1 + math.exp(-t))
```

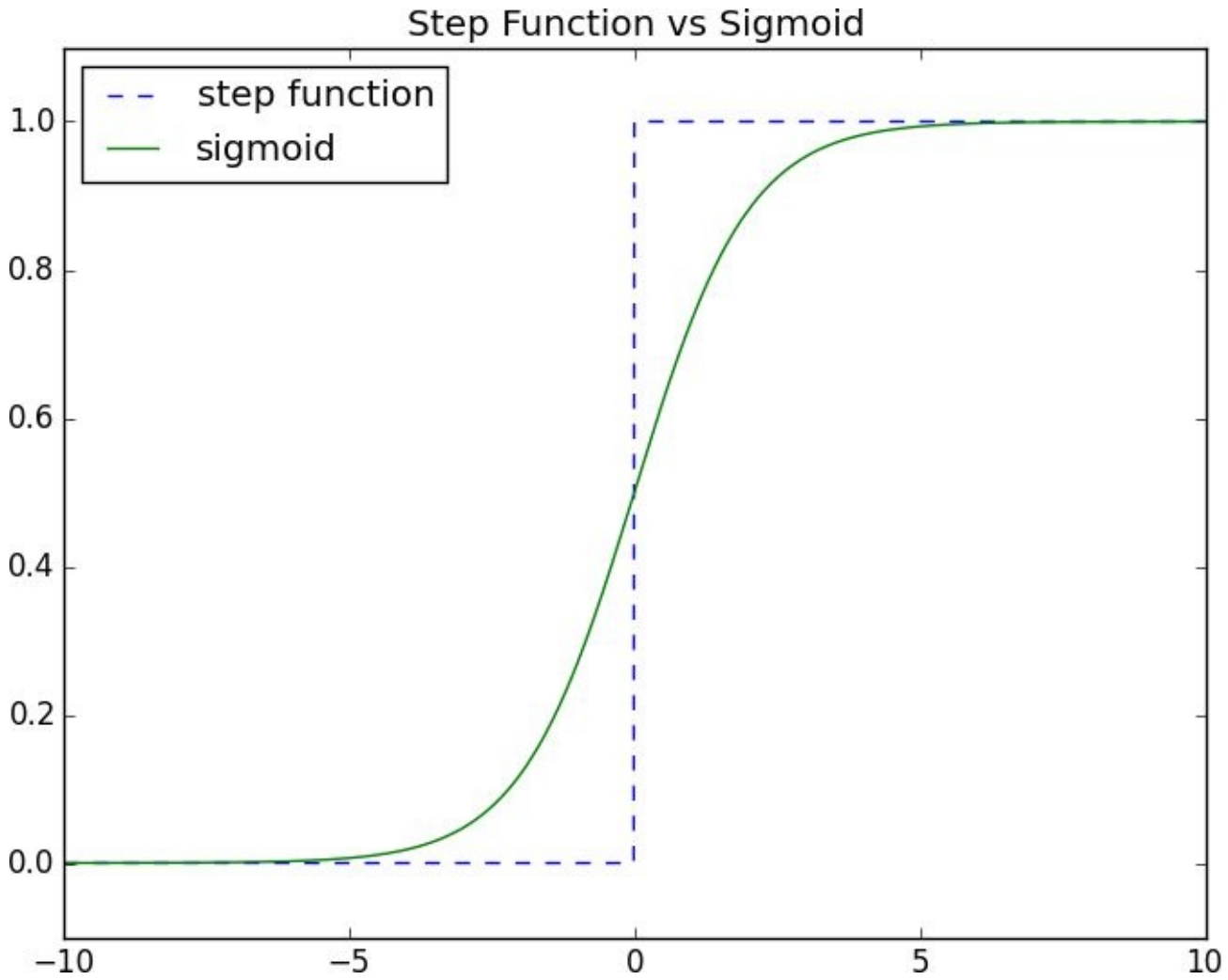

Figure 18-2. The sigmoid function

Chapter 19. Clustering

> Where we such clusters had
>
> As made us nobly wild, not mad
>
> Robert Herrick

Most of the algorithms in this book are what's known as supervised learning, in that they start with a set of *labeled* data and use that as the basis for making predictions about new, unlabeled data. Clustering, however, is an example of unsupervised learning, in which we work with completely unlabeled data (or in which our data has labels but we ignore them).

The Idea

Whenever you look at some source of data, it's likely that the data will somehow form *clusters*. A data set showing where millionaires live probably has clusters in places like Beverly Hills and Manhattan. A data set showing how many hours people work each week probably has a cluster around 40 (and if it's taken from a state with laws mandating special benefits for people who work at least 20 hours a week, it probably has another cluster right around 19). A data set of demographics of registered voters likely forms a variety of clusters (e.g., "soccer moms," "bored retirees," "unemployed millennials") that pollsters and political consultants likely consider relevant.

Unlike some of the problems we've looked at, there is generally no "correct" clustering. An alternative clustering scheme might group some of the "unemployed millenials" with "grad students," others with "parents' basement dwellers." Neither scheme is necessarily more correct — instead, each is likely more optimal with respect to its own "how good are the clusters?" metric.

Furthermore, the clusters won't label themselves. You'll have to do that by looking at the data underlying each one.

The Model

For us, each `input` will be a vector in *d*-dimensional space (which, as usual, we will represent as a list of numbers). Our goal will be to identify clusters of similar inputs and (sometimes) to find a representative value for each cluster.

For example, each input could be (a numeric vector that somehow represents) the title of a blog post, in which case the goal might be to find clusters of similar posts, perhaps in order to understand what our users are blogging about. Or imagine that we have a picture containing thousands of `(red, green, blue)` colors and that we need to screen-print a 10-color version of it. Clustering can help us choose 10 colors that will minimize the total "color error."

One of the simplest clustering methods is *k-means*, in which the number of clusters *k* is chosen in advance, after which the goal is to partition the inputs into sets $S_1, \ldots, S_k$ in a way that minimizes the total sum of squared distances from each point to the mean of its assigned cluster.

There are a lot of ways to assign *n* points to *k* clusters, which means that finding an optimal clustering is a very hard problem. We'll settle for an iterative algorithm that usually finds a good clustering:

1. Start with a set of *k-means*, which are points in *d*-dimensional space.
2. Assign each point to the mean to which it is closest.
3. If no point's assignment has changed, stop and keep the clusters.
4. If some point's assignment has changed, recompute the means and return to step 2.

Using the `vector_mean` function from Chapter 4, it's pretty simple to create a class that does this:

```
class KMeans:
    """performs k-means clustering"""

    def __init__(self, k):
        self.k = k          # number of clusters
        self.means = None   # means of clusters

    def classify(self, input):
        """return the index of the cluster closest to the input"""
        return min(range(self.k),
                   key=lambda i: squared_distance(input, self.means[i]))

    def train(self, inputs):
        # choose k random points as the initial means
        self.means = random.sample(inputs, self.k)
        assignments = None

        while True:
            # Find new assignments
            new_assignments = map(self.classify, inputs)

            # If no assignments have changed, we're done.
            if assignments == new_assignments:
```

```
            return

        # Otherwise keep the new assignments,
        assignments = new_assignments

        # And compute new means based on the new assignments
        for i in range(self.k):
            # find all the points assigned to cluster i
            i_points = [p for p, a in zip(inputs, assignments) if a == i]

            # make sure i_points is not empty so don't divide by 0
            if i_points:
                self.means[i] = vector_mean(i_points)
```

Let’s take a look at how this works.

Chapter 20. Natural Language Processing

> They have been at a great feast of languages, and stolen the scraps.
>
> William Shakespeare

Natural language processing (NLP) refers to computational techniques involving language. It’s a broad field, but we’ll look at a few techniques both simple and not simple.

Word Clouds

In Chapter 1, we computed word counts of users' interests. One approach to visualizing words and counts is word clouds, which artistically lay out the words with sizes proportional to their counts.

Generally, though, data scientists don't think much of word clouds, in large part because the placement of the words doesn't mean anything other than "here's some space where I was able to fit a word."

If you ever are forced to create a word cloud, think about whether you can make the axes convey something. For example, imagine that, for each of some collection of data science–related buzzwords, you have two numbers between 0 and 100 — the first representing how frequently it appears in job postings, the second how frequently it appears on resumes:

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
         ("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 60),
         ("data science", 60, 70), ("analytics", 90, 3),
         ("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0),
         ("actionable insights", 40, 30), ("think out of the box", 45, 10),
         ("self-starter", 30, 50), ("customer focus", 65, 15),
         ("thought leadership", 35, 35)]
```

The word cloud approach is just to arrange the words on a page in a cool-looking font (Figure 20-1).

Figure 20-1. Buzzword cloud

This looks neat but doesn't really tell us anything. A more interesting approach might be to scatter them so that horizontal position indicates posting popularity and vertical position indicates resume popularity, which produces a visualization that conveys a few insights (Figure 20-2):

```
def text_size(total):
    """equals 8 if total is 0, 28 if total is 200"""
    return 8 + total / 200 * 20

for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
plt.xlabel("Popularity on Job Postings")
plt.ylabel("Popularity on Resumes")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

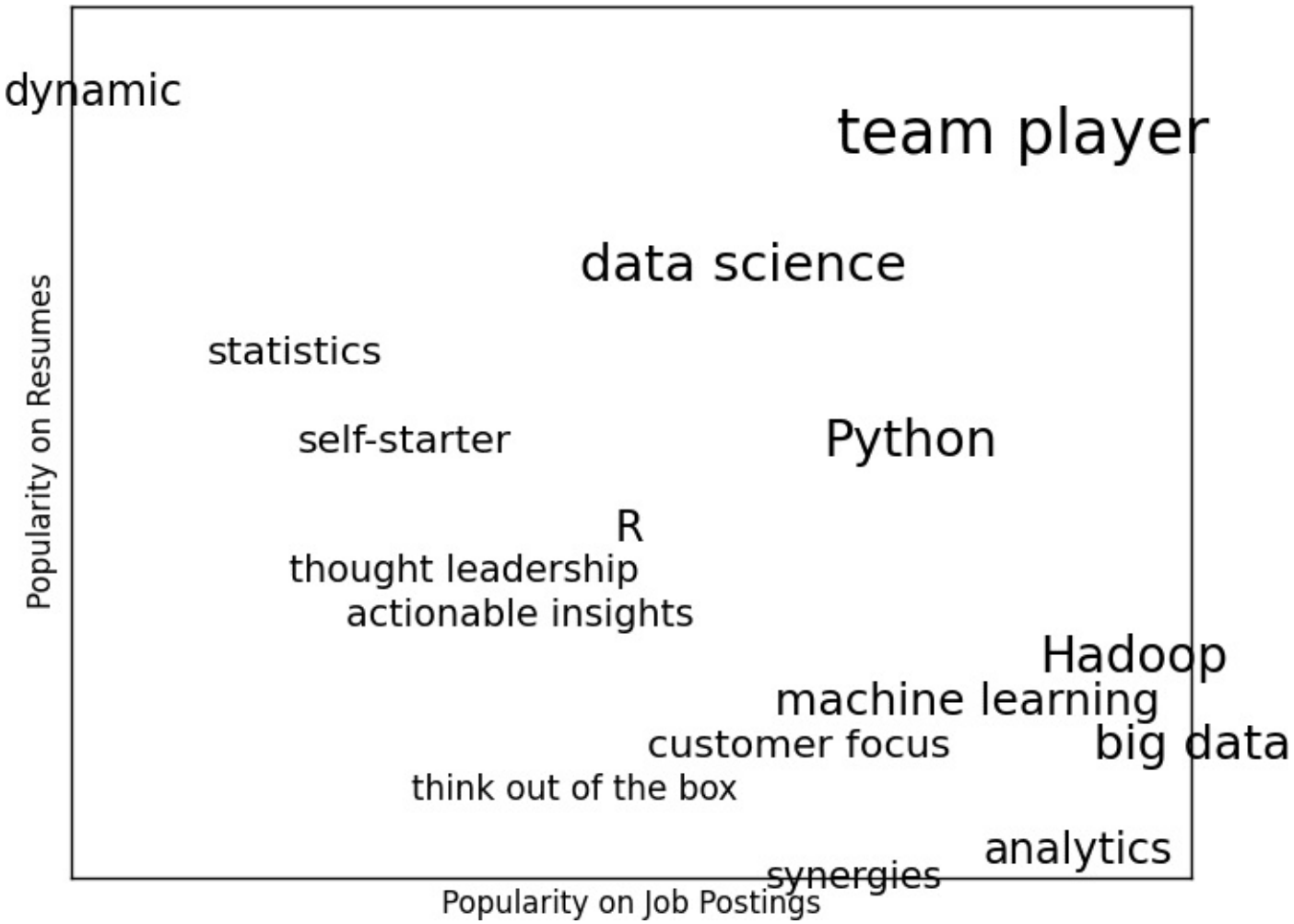

Figure 20-2. A more meaningful (if less attractive) word cloud

Chapter 21. Network Analysis

> Your connections to all the things around you literally define who you are.
>
> Aaron O’Connell

Many interesting data problems can be fruitfully thought of in terms of *networks*, consisting of *nodes* of some type and the *edges* that join them.

For instance, your Facebook friends form the nodes of a network whose edges are friendship relations. A less obvious example is the World Wide Web itself, with each web page a node, and each hyperlink from one page to another an edge.

Facebook friendship is mutual — if I am Facebook friends with you than necessarily you are friends with me. In this case, we say that the edges are *undirected*. Hyperlinks are not — my website links to whitehouse.gov, but (for reasons inexplicable to me) whitehouse.gov refuses to link to my website. We call these types of edges *directed*. We’ll look at both kinds of networks.

Betweenness Centrality

In Chapter 1, we computed the key connectors in the DataSciencester network by counting the number of friends each user had. Now we have enough machinery to look at other approaches. Recall that the network (Figure 21-1) comprised users:

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

and friendships:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

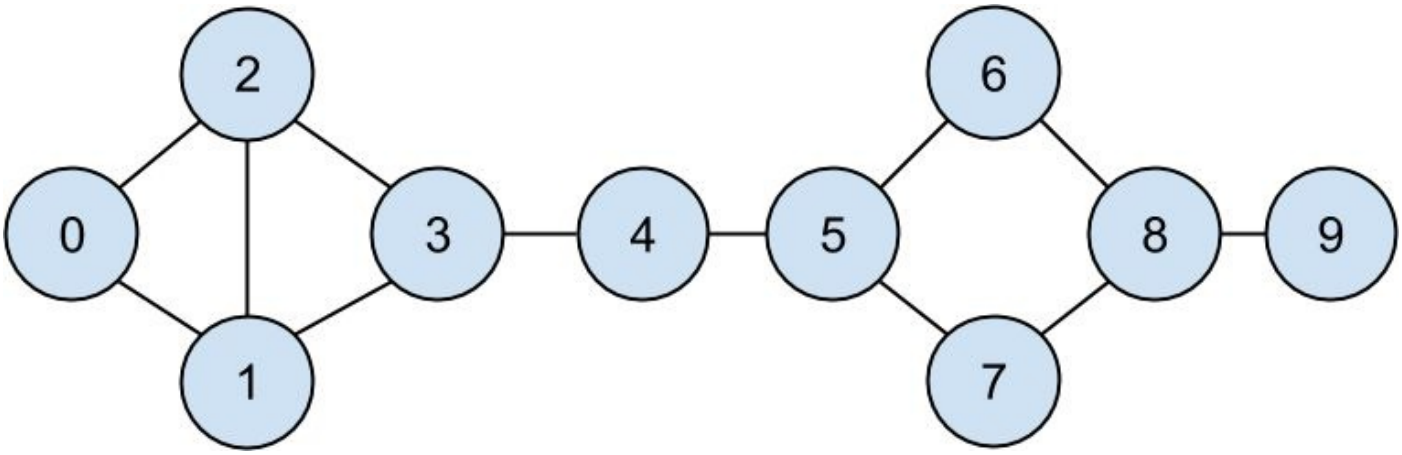

Figure 21-1. The DataSciencester network

We also added friend lists to each user `dict`:

```
for user in users:
    user["friends"] = []

for i, j in friendships:
    # this works because users[i] is the user whose id is i
    users[i]["friends"].append(users[j]) # add i as a friend of j
    users[j]["friends"].append(users[i]) # add j as a friend of i
```

When we left off we were dissatisfied with our notion of *degree centrality*, which didn't really agree with our intuition about who were the key connectors of the network.

An alternative metric is *betweenness centrality*, which identifies people who frequently are on the shortest paths between pairs of other people. In particular, the betweenness centrality of node *i* is computed by adding up, for every other pair of nodes *j* and *k*, the proportion of shortest paths between node *j* and node *k* that pass through *i*.

That is, to figure out Thor's betweenness centrality, we'll need to compute all the shortest paths between all pairs of people who aren't Thor. And then we'll need to count how many of those shortest paths pass through Thor. For instance, the only shortest path between Chi (`id` 3) and Clive (`id` 5) passes through Thor, while neither of the two shortest paths between Hero (`id` 0) and Chi (`id` 3) does.

So, as a first step, we'll need to figure out the shortest paths between all pairs of people. There are some pretty sophisticated algorithms for doing so efficiently, but (as is almost always the case) we will use a less efficient, easier-to-understand algorithm.

This algorithm (an implementation of breadth-first search) is one of the more complicated ones in the book, so let's talk through it carefully:

1. Our goal is a function that takes a `from_user` and finds *all* shortest paths to every other user.

2. We'll represent a path as `list` of user IDs. Since every path starts at `from_user`, we won't include her ID in the list. This means that the length of the list representing the path will be the length of the path itself.

3. We'll maintain a dictionary `shortest_paths_to` where the keys are user IDs and the values are lists of paths that end at the user with the specified ID. If there is a unique shortest path, the list will just contain that one path. If there are multiple shortest paths, the list will contain all of them.

4. We'll also maintain a queue `frontier` that contains the users we want to explore in the order we want to explore them. We'll store them as pairs `(prev_user, user)` so that we know how we got to each one. We initialize the queue with all the neighbors of `from_user`. (We haven't ever talked about queues, which are data structures optimized for "add to the end" and "remove from the front" operations. In Python, they are implemented as `collections.deque` which is actually a double-ended queue.)

5. As we explore the graph, whenever we find new neighbors that we don't already know shortest paths to, we add them to the end of the queue to explore later, with the current user as `prev_user`.

6. When we take a user off the queue, and we've never encountered that user before, we've definitely found one or more shortest paths to him — each of the shortest paths to `prev_user` with one extra step added.

7. When we take a user off the queue and we *have* encountered that user before, then either we've found another shortest path (in which case we should add it) or we've found a longer path (in which case we shouldn't).

8. When no more users are left on the queue, we've explored the whole graph (or, at least, the parts of it that are reachable from the starting user) and we're done.

We can put this all together into a (large) function:

```
from collections import deque

def shortest_paths_from(from_user):

    # a dictionary from "user_id" to *all* shortest paths to that user
    shortest_paths_to = { from_user["id"] : [[]] }

    # a queue of (previous user, next user) that we need to check.
    # starts out with all pairs (from_user, friend_of_from_user)
    frontier = deque((from_user, friend)
                     for friend in from_user["friends"])

    # keep going until we empty the queue
    while frontier:

        prev_user, user = frontier.popleft()   # remove the user who's
        user_id = user["id"]                   # first in the queue

        # because of the way we're adding to the queue,
        # necessarily we already know some shortest paths to prev_user
        paths_to_prev_user = shortest_paths_to[prev_user["id"]]
        new_paths_to_user = [path + [user_id] for path in paths_to_prev_user]

        # it's possible we already know a shortest path
        old_paths_to_user = shortest_paths_to.get(user_id, [])

        # what's the shortest path to here that we've seen so far?
        if old_paths_to_user:
            min_path_length = len(old_paths_to_user[0])
        else:
            min_path_length = float('inf')

        # only keep paths that aren't too long and are actually new
        new_paths_to_user = [path
                             for path in new_paths_to_user
                             if len(path) <= min_path_length
                             and path not in old_paths_to_user]

        shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user

        # add never-seen neighbors to the frontier
        frontier.extend((user, friend)
                        for friend in user["friends"]
                        if friend["id"] not in shortest_paths_to)

    return shortest_paths_to
```

Now we can store these `dicts` with each node:

```
for user in users:
    user["shortest_paths"] = shortest_paths_from(user)
```

And we're finally ready to compute betweenness centrality. For every pair of nodes *i* and *j*, we know the *n* shortest paths from *i* to *j*. Then, for each of those paths, we just add *1/n* to the centrality of each node on that path:

```
for user in users:
    user["betweenness_centrality"] = 0.0

for source in users:
    source_id = source["id"]
    for target_id, paths in source["shortest_paths"].iteritems():
        if source_id < target_id:      # don't double count
            num_paths = len(paths)     # how many shortest paths?
            contrib = 1 / num_paths    # contribution to centrality
            for path in paths:
```

Chapter 22. Recommender Systems

> O nature, nature, why art thou so dishonest, as ever to send men with these false recommendations into the world!
>
> Henry Fielding

Another common data problem is producing *recommendations* of some sort. Netflix recommends movies you might want to watch. Amazon recommends products you might want to buy. Twitter recommends users you might want to follow. In this chapter, we'll look at several ways to use data to make recommendations.

In particular, we'll look at the data set of `users_interests` that we've used before:

```
users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

And we'll think about the problem of recommending new interests to a user based on her currently specified interests.

Manual Curation

Before the Internet, when you needed book recommendations you would go to the library, where a librarian was available to suggest books that were relevant to your interests or similar to books you liked.

Given DataSciencester's limited number of users and interests, it would be easy for you to spend an afternoon manually recommending interests for each user. But this method doesn't scale particularly well, and it's limited by your personal knowledge and imagination. (Not that I'm suggesting that your personal knowledge and imagination are limited.) So let's think about what we can do with *data*.

Recommending What's Popular

One easy approach is to simply recommend what's popular:

```
popular_interests = Counter(interest
                            for user_interests in users_interests
                            for interest in user_interests).most_common()
```

which looks like:

```
[('Python', 4),
 ('R', 4),
 ('Java', 3),
 ('regression', 3),
 ('statistics', 3),
 ('probability', 3),
 # ...
]
```

Having computed this, we can just suggest to a user the most popular interests that he's not already interested in:

```
def most_popular_new_interests(user_interests, max_results=5):
    suggestions = [(interest, frequency)
                   for interest, frequency in popular_interests
                   if interest not in user_interests]
    return suggestions[:max_results]
```

So, if you are user 1, with interests:

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

then we'd recommend you:

```
most_popular_new_interests(users_interests[1], 5)

# [('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics', 3)]
```

If you are user 3, who's already interested in many of those things, you'd instead get:

```
[('Java', 3),
 ('HBase', 3),
 ('Big Data', 3),
 ('neural networks', 2),
 ('Hadoop', 2)]
```

Of course, "lots of people are interested in Python so maybe you should be too" is not the most compelling sales pitch. If someone is brand new to our site and we don't know anything about them, that's possibly the best we can do. Let's see how we can do better by basing each user's recommendations on her interests.

User-Based Collaborative Filtering

One way of taking a user's interests into account is to look for users who are somehow *similar* to him, and then suggest the things that those users are interested in.

In order to do that, we'll need a way to measure how similar two users are. Here we'll use a metric called *cosine similarity*. Given two vectors, `v` and `w`, it's defined as:

```
def cosine_similarity(v, w):
    return dot(v, w) / math.sqrt(dot(v, v) * dot(w, w))
```

It measures the "angle" between `v` and `w`. If `v` and `w` point in the same direction, then the numerator and denominator are equal, and their cosine similarity equals 1. If `v` and `w` point in opposite directions, then their cosine similarity equals -1. And if `v` is 0 whenever `w` is not (and vice versa) then `dot(v, w)` is 0 and so the cosine similarity will be 0.

We'll apply this to vectors of 0s and 1s, each vector `v` representing one user's interests. `v[i]` will be 1 if the user is specified the *i*th interest, 0 otherwise. Accordingly, "similar users" will mean "users whose interest vectors most nearly point in the same direction." Users with identical interests will have similarity 1. Users with no identical interests will have similarity 0. Otherwise the similarity will fall in between, with numbers closer to 1 indicating "very similar" and numbers closer to 0 indicating "not very similar."

A good place to start is collecting the known interests and (implicitly) assigning indices to them. We can do this by using a set comprehension to find the unique interests, putting them in a list, and then sorting them. The first interest in the resulting list will be interest 0, and so on:

```
unique_interests = sorted(list({ interest
                                 for user_interests in users_interests
                                 for interest in user_interests }))
```

This gives us a list that starts:

```
['Big Data',
 'C++',
 'Cassandra',
 'HBase',
 'Hadoop',
 'Haskell',
 # ...
]
```

Next we want to produce an "interest" vector of 0s and 1s for each user. We just need to iterate over the `unique_interests` list, substituting a 1 if the user has each interest, a 0 if not:

```
def make_user_interest_vector(user_interests):
    """given a list of interests, produce a vector whose ith element is 1
    if unique_interests[i] is in the list, 0 otherwise"""
    return [1 if interest in user_interests else 0
            for interest in unique_interests]
```

Chapter 23. Databases and SQL

> Memory is man’s greatest friend and worst enemy.
>
> Gilbert Parker

The data you need will often live in *databases*, systems designed for efficiently storing and querying data. The bulk of these are *relational* databases, such as Oracle, MySQL, and SQL Server, which store data in *tables* and are typically queried using Structured Query Language (SQL), a declarative language for manipulating data.

SQL is a pretty essential part of the data scientist’s toolkit. In this chapter, we’ll create NotQuiteABase, a Python implementation of something that’s not quite a database. We’ll also cover the basics of SQL while showing how they work in our not-quite database, which is the most “from scratch” way I could think of to help you understand what they’re doing. My hope is that solving problems in NotQuiteABase will give you a good sense of how you might solve the same problems using SQL.

CREATE TABLE and INSERT

A relational database is a collection of tables (and of relationships among them). A table is simply a collection of rows, not unlike the matrices we've been working with. However, a table also has associated with it a fixed *schema* consisting of column names and column types.

For example, imagine a `users` data set containing for each user her `user_id`, `name`, and `num_friends`:

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
         [2, "Sue", 3],
         [3, "Chi", 3]]
```

In SQL, we might create this table with:

```
CREATE TABLE users (
    user_id INT NOT NULL,
    name VARCHAR(200),
    num_friends INT);
```

Notice that we specified that the `user_id` and `num_friends` must be integers (and that `user_id` isn't allowed to be `NULL`, which indicates a missing value and is sort of like our `None`) and that the name should be a string of length 200 or less. NotQuiteABase won't take types into account, but we'll behave as if it did.

> **NOTE**
>
> SQL is almost completely case and indentation insensitive. The capitalization and indentation style here is my preferred style. If you start learning SQL, you will surely encounter other examples styled differently.

You can insert the rows with `INSERT` statements:

```
INSERT INTO users (user_id, name, num_friends) VALUES (0, 'Hero', 0);
```

Notice also that SQL statements need to end with semicolons, and that SQL requires single quotes for its strings.

In NotQuiteABase, you'll create a `Table` simply by specifying the names of its columns. And to insert a row, you'll use the table's `insert()` method, which takes a `list` of row values that need to be in the same order as the table's column names.

Behind the scenes, we'll store each row as a `dict` from column names to values. A real database would never use such a space-wasting representation, but doing so will make NotQuiteABase much easier to work with:

```
class Table:
    def __init__(self, columns):
        self.columns = columns
        self.rows = []
```

```
    def __repr__(self):
        """pretty representation of the table: columns then rows"""
        return str(self.columns) + "\n" + "\n".join(map(str, self.rows))

    def insert(self, row_values):
        if len(row_values) != len(self.columns):
            raise TypeError("wrong number of elements")
        row_dict = dict(zip(self.columns, row_values))
        self.rows.append(row_dict)
```

For example, we could set up:

```
users = Table(["user_id", "name", "num_friends"])
users.insert([0, "Hero", 0])
users.insert([1, "Dunn", 2])
users.insert([2, "Sue", 3])
users.insert([3, "Chi", 3])
users.insert([4, "Thor", 3])
users.insert([5, "Clive", 2])
users.insert([6, "Hicks", 3])
users.insert([7, "Devin", 2])
users.insert([8, "Kate", 2])
users.insert([9, "Klein", 3])
users.insert([10, "Jen", 1])
```

If you now `print users`, you'll see:

```
['user_id', 'name', 'num_friends']
{'user_id': 0, 'name': 'Hero', 'num_friends': 0}
{'user_id': 1, 'name': 'Dunn', 'num_friends': 2}
{'user_id': 2, 'name': 'Sue', 'num_friends': 3}
...
```

UPDATE

Sometimes you need to update the data that's already in the database. For instance, if Dunn acquires another friend, you might need to do this:

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

The key features are:

- What table to update
- Which rows to update
- Which fields to update
- What their new values should be

We'll add a similar `update` method to NotQuiteABase. Its first argument will be a `dict` whose keys are the columns to update and whose values are the new values for those fields. And its second argument is a `predicate` that returns `True` for rows that should be updated, `False` otherwise:

```
    def update(self, updates, predicate):
        for row in self.rows:
            if predicate(row):
                for column, new_value in updates.iteritems():
                    row[column] = new_value
```

after which we can simply do this:

```
users.update({'num_friends' : 3},               # set num_friends = 3
             lambda row: row['user_id'] == 1)   # in rows where user_id == 1
```

Chapter 24. MapReduce

> The future has already arrived. It's just not evenly distributed yet.
>
> William Gibson

MapReduce is a programming model for performing parallel processing on large data sets. Although it is a powerful technique, its basics are relatively simple.

Imagine we have a collection of items we'd like to process somehow. For instance, the items might be website logs, the texts of various books, image files, or anything else. A basic version of the MapReduce algorithm consists of the following steps:

1. Use a `mapper` function to turn each item into zero or more key-value pairs. (Often this is called the `map` function, but there is already a Python function called `map` and we don't need to confuse the two.)

2. Collect together all the pairs with identical keys.

3. Use a `reducer` function on each collection of grouped values to produce output values for the corresponding key.

This is all sort of abstract, so let's look at a specific example. There are few absolute rules of data science, but one of them is that your first MapReduce example has to involve counting words.

Example: Word Count

DataSciencester has grown to millions of users! This is great for your job security, but it makes routine analyses slightly more difficult.

For example, your VP of Content wants to know what sorts of things people are talking about in their status updates. As a first attempt, you decide to count the words that appear, so that you can prepare a report on the most frequent ones.

When you had a few hundred users this was simple to do:

```
def word_count_old(documents):
    """word count not using MapReduce"""
    return Counter(word
        for document in documents
        for word in tokenize(document))
```

With millions of users the set of `documents` (status updates) is suddenly too big to fit on your computer. If you can just fit this into the MapReduce model, you can use some "big data" infrastructure that your engineers have implemented.

First, we need a function that turns a document into a sequence of key-value pairs. We'll want our output to be grouped by word, which means that the keys should be words. And for each word, we'll just emit the value `1` to indicate that this pair corresponds to one occurrence of the word:

```
def wc_mapper(document):
    """for each word in the document, emit (word,1)"""
    for word in tokenize(document):
        yield (word, 1)
```

Skipping the "plumbing" step 2 for the moment, imagine that for some word we've collected a list of the corresponding counts we emitted. Then to produce the overall count for that word we just need:

```
def wc_reducer(word, counts):
    """sum up the counts for a word"""
    yield (word, sum(counts))
```

Returning to step 2, we now need to collect the results from `wc_mapper` and feed them to `wc_reducer`. Let's think about how we would do this on just one computer:

```
def word_count(documents):
    """count the words in the input documents using MapReduce"""

    # place to store grouped values
    collector = defaultdict(list)

    for document in documents:
        for word, count in wc_mapper(document):
            collector[word].append(count)

    return [output
            for word, counts in collector.iteritems()
            for output in wc_reducer(word, counts)]
```

Imagine that we have three documents `["data science", "big data", "science fiction"]`.

Then `wc_mapper` applied to the first document yields the two pairs `("data", 1)` and `("science", 1)`. After we've gone through all three documents, the `collector` contains

```
{ "data" : [1, 1],
  "science" : [1, 1],
  "big" : [1],
  "fiction" : [1] }
```

Then `wc_reducer` produces the count for each word:

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

Why MapReduce?

As mentioned earlier, the primary benefit of MapReduce is that it allows us to distribute computations by moving the processing to the data. Imagine we want to word-count across billions of documents.

Our original (non-MapReduce) approach requires the machine doing the processing to have access to every document. This means that the documents all need to either live on that machine or else be transferred to it during processing. More important, it means that the machine can only process one document at a time.

> **NOTE**
>
> Possibly it can process up to a few at a time if it has multiple cores and if the code is rewritten to take advantage of them. But even so, all the documents still have to *get to* that machine.

Imagine now that our billions of documents are scattered across 100 machines. With the right infrastructure (and glossing over some of the details), we can do the following:

- Have each machine run the mapper on its documents, producing lots of (key, value) pairs.
- Distribute those (key, value) pairs to a number of "reducing" machines, making sure that the pairs corresponding to any given key all end up on the same machine.
- Have each reducing machine group the pairs by key and then run the reducer on each set of values.
- Return each (key, output) pair.

What is amazing about this is that it scales horizontally. If we double the number of machines, then (ignoring certain fixed-costs of running a MapReduce system) our computation should run approximately twice as fast. Each mapper machine will only need to do half as much work, and (assuming there are enough distinct keys to further distribute the reducer work) the same is true for the reducer machines.

Chapter 25. Go Forth and Do Data Science

> And now, once again, I bid my hideous progeny go forth and prosper.
>
> Mary Shelley

Where do you go from here? Assuming I haven’t scared you off of data science, there are a number of things you should learn next.

IPython

We mentioned IPython earlier in the book. It provides a shell with far more functionality than the standard Python shell, and it adds "magic functions" that allow you to (among other things) easily copy and paste code (which is normally complicated by the combination of blank lines and whitespace formatting) and run scripts from within the shell.

Mastering IPython will make your life far easier. (Even learning just a little bit of IPython will make your life a lot easier.)

Additionally, it allows you to create "notebooks" combining text, live Python code, and visualizations that you can share with other people, or just keep around as a journal of what you did (Figure 25-1).

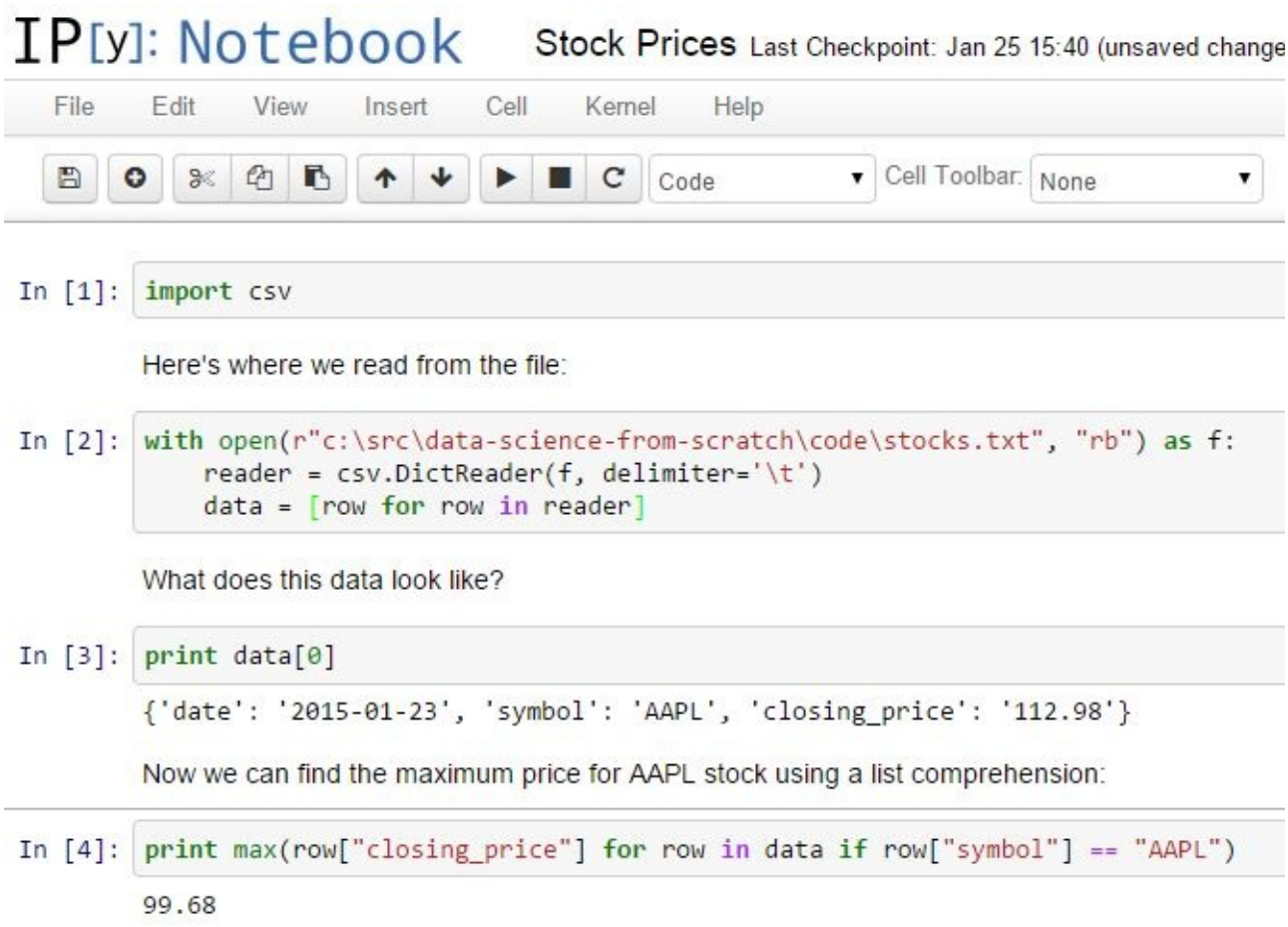

Figure 25-1. An IPython notebook

Mathematics

Throughout this book, we dabbled in linear algebra (Chapter 4), statistics (Chapter 5), probability (Chapter 6), and various aspects of machine learning.

To be a good data scientist, you should know much more about these topics, and I encourage you to give each of them a more in-depth study, using the textbooks recommended at the end of the chapters, your own preferred textbooks, online courses, or even real-life courses.

Not from Scratch

Implementing things “from scratch” is great for understanding how they work. But it’s generally not great for performance (unless you’re implementing them specifically with performance in mind), ease of use, rapid prototyping, or error handling.

In practice, you’ll want to use well-designed libraries that solidly implement the fundamentals. (My original proposal for this book involved a second “now let’s learn the libraries” half that O’Reilly, thankfully, vetoed.)