MODULE-2

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.

vtucode

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.

YuCode

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)
```

```
power = 1 - type_2_probability      # 0.936
```

This is a more powerful test, since it no longer rejects H_0 when X is below 469 (which is very unlikely to happen if H_1 is true) and instead rejects H_0 when X is between 526 and 531 (which is somewhat likely to happen if H_1 is true). === p-values

An alternative way of thinking about the preceding test involves *p-values*. Instead of choosing bounds based on some probability cutoff, we compute the probability — assuming H_0 is true — that we would see a value at least as extreme as the one we actually observed.

For our two-sided test of whether the coin is fair, we compute:

```
def two_sided_p_value(x, mu=0, sigma=1):
    if x >= mu:
        # if x is greater than the mean, the tail is what's greater than x
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # if x is less than the mean, the tail is what's less than x
        return 2 * normal_probability_below(x, mu, sigma)
```

If we were to see 530 heads, we would compute:

```
two_sided_p_value(529.5, mu_0, sigma_0)   # 0.062
```

> **NOTE**
>
> Why did we use 529.5 instead of 530? This is what's called a *continuity correction*. It reflects the fact that `normal_probability_between(529.5, 530.5, mu_0, sigma_0)` is a better estimate of the probability of seeing 530 heads than `normal_probability_between(530, 531, mu_0, sigma_0)` is.
>
> Correspondingly, `normal_probability_above(529.5, mu_0, sigma_0)` is a better estimate of the probability of seeing at least 530 heads. You may have noticed that we also used this in the code that produced Figure 6-4.

One way to convince yourself that this is a sensible estimate is with a simulation:

```
extreme_value_count = 0
for _ in range(100000):
    num_heads = sum(1 if random.random() < 0.5 else 0     # count # of heads
                    for _ in range(1000))                 # in 1000 flips
    if num_heads >= 530 or num_heads <= 470:              # and count how often
        extreme_value_count += 1                          # the # is 'extreme'

print extreme_value_count / 100000   # 0.062
```

Since the *p*-value is greater than our 5% significance, we don't reject the null. If we instead saw 532 heads, the *p*-value would be:

```
two_sided_p_value(531.5, mu_0, sigma_0)   # 0.0463
```

which is smaller than the 5% significance, which means we would reject the null. It's the exact same test as before. It's just a different way of approaching the statistics.

Similarly, we would have:

```
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

For our one-sided test, if we saw 525 heads we would compute:

```
upper_p_value(524.5, mu_0, sigma_0) # 0.061
```

which means we wouldn't reject the null. If we saw 527 heads, the computation would be:

```
upper_p_value(526.5, mu_0, sigma_0) # 0.047
```

and we would reject the null.

> **WARNING**
>
> Make sure your data is roughly normally distributed before using `normal_probability_above` to compute p-values. The annals of bad data science are filled with examples of people opining that the chance of some observed event occurring at random is one in a million, when what they really mean is "the chance, assuming the data is distributed normally," which is pretty meaningless if the data isn't.
>
> There are various statistical tests for normality, but even plotting the data is a good start.

Vtucode

Confidence Intervals

We've been testing hypotheses about the value of the heads probability *p*, which is a *parameter* of the unknown "heads" distribution. When this is the case, a third approach is to construct a *confidence interval* around the observed value of the parameter.

For example, we can estimate the probability of the unfair coin by looking at the average value of the Bernoulli variables corresponding to each flip — 1 if heads, 0 if tails. If we observe 525 heads out of 1,000 flips, then we estimate *p* equals 0.525.

How *confident* can we be about this estimate? Well, if we knew the exact value of *p*, the central limit theorem (recall "The Central Limit Theorem") tells us that the average of those Bernoulli variables should be approximately normal, with mean *p* and standard deviation:

```
math.sqrt(p * (1 - p) / 1000)
```

Here we don't know *p*, so instead we use our estimate:

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)   # 0.0158
```

This is not entirely justified, but people seem to do it anyway. Using the normal approximation, we conclude that we are "95% confident" that the following interval contains the true parameter *p*:

```
normal_two_sided_bounds(0.95, mu, sigma)        # [0.4940, 0.5560]
```

> **NOTE**
>
> This is a statement about the *interval*, not about *p*. You should understand it as the assertion that if you were to repeat the experiment many times, 95% of the time the "true" parameter (which is the same every time) would lie within the observed confidence interval (which might be different every time).

In particular, we do not conclude that the coin is unfair, since 0.5 falls within our confidence interval.

If instead we'd seen 540 heads, then we'd have:

```
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
normal_two_sided_bounds(0.95, mu, sigma) # [0.5091, 0.5709]
```

Here, "fair coin" doesn't lie in the confidence interval. (The "fair coin" hypothesis doesn't pass a test that you'd expect it to pass 95% of the time if it were true.)

P-hacking

A procedure that erroneously rejects the null hypothesis only 5% of the time will — by definition — 5% of the time erroneously reject the null hypothesis:

```
def run_experiment():
    """flip a fair coin 1000 times, True = heads, False = tails"""
    return [random.random() < 0.5 for _ in range(1000)]

def reject_fairness(experiment):
    """using the 5% significance levels"""
    num_heads = len([flip for flip in experiment if flip])
    return num_heads < 469 or num_heads > 531

random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
                      for experiment in experiments
                      if reject_fairness(experiment)])

print num_rejections   # 46
```

What this means is that if you're setting out to find "significant" results, you usually can. Test enough hypotheses against your data set, and one of them will almost certainly appear significant. Remove the right outliers, and you can probably get your *p* value below 0.05. (We did something vaguely similar in "Correlation"; did you notice?)

This is sometimes called P-hacking and is in some ways a consequence of the "inference from *p*-values framework." A good article criticizing this approach is "The Earth Is Round."

If you want to do good *science*, you should determine your hypotheses before looking at the data, you should clean your data without the hypotheses in mind, and you should keep in mind that *p*-values are not substitutes for common sense. (An alternative approach is "Bayesian Inference".)

Example: Running an A/B Test

One of your primary responsibilities at DataSciencester is experience optimization, which is a euphemism for trying to get people to click on advertisements. One of your advertisers has developed a new energy drink targeted at data scientists, and the VP of Advertisements wants your help choosing between advertisement A ("tastes great!") and advertisement B ("less bias!").

Being a *scientist*, you decide to run an *experiment* by randomly showing site visitors one of the two advertisements and tracking how many people click on each one.

If 990 out of 1,000 A-viewers click their ad while only 10 out of 1,000 B-viewers click their ad, you can be pretty confident that A is the better ad. But what if the differences are not so stark? Here's where you'd use statistical inference.

Let's say that N_A people see ad A, and that n_A of them click it. We can think of each ad view as a Bernoulli trial where p_A is the probability that someone clicks ad A. Then (if N_A is large, which it is here) we know that n_A / N_A is approximately a normal random variable with mean p_A and standard deviation $\sigma_A = \sqrt{p_A(1-p_A)/N_A}$.

Similarly, n_B / N_B is approximately a normal random variable with mean p_B and standard deviation $\sigma_B = \sqrt{p_B(1-p_B)/N_B}$:

```
def estimated_parameters(N, n):
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

If we assume those two normals are independent (which seems reasonable, since the individual Bernoulli trials ought to be), then their difference should also be normal with mean $p_B - p_A$ and standard deviation $\sqrt{\sigma_A^2 + \sigma_B^2}$.

> **NOTE**
>
> This is sort of cheating. The math only works out exactly like this if you *know* the standard deviations. Here we're estimating them from the data, which means that we really should be using a *t*-distribution. But for large enough data sets, it's close enough that it doesn't make much of a difference.

This means we can test the *null hypothesis* that p_A and p_B are the same (that is, that $p_A - p_B$ is zero), using the statistic:

```
def a_b_test_statistic(N_A, n_A, N_B, n_B):
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

which should approximately be a standard normal.

For example, if "tastes great" gets 200 clicks out of 1,000 views and "less bias" gets 180 clicks out of 1,000 views, the statistic equals:

```
z = a_b_test_statistic(1000, 200, 1000, 180)    # -1.14
```

The probability of seeing such a large difference if the means were actually equal would be:

```
two_sided_p_value(z)                            # 0.254
```

which is large enough that you can't conclude there's much of a difference. On the other hand, if "less bias" only got 150 clicks, we'd have:

```
z = a_b_test_statistic(1000, 200, 1000, 150)    # -2.94
two_sided_p_value(z)                            # 0.003
```

which means there's only a 0.003 probability you'd see such a large difference if the ads were equally effective.

YuCode

Bayesian Inference

The procedures we've looked at have involved making probability statements about our *tests*: "there's only a 3% chance you'd observe such an extreme statistic if our null hypothesis were true."

An alternative approach to inference involves treating the unknown parameters themselves as random variables. The analyst (that's you) starts with a *prior distribution* for the parameters and then uses the observed data and Bayes's Theorem to get an updated *posterior distribution* for the parameters. Rather than making probability judgments about the tests, you make probability judgments about the parameters themselves.

For example, when the unknown parameter is a probability (as in our coin-flipping example), we often use a prior from the *Beta distribution*, which puts all its probability between 0 and 1:

```
def B(alpha, beta):
    """a normalizing constant so that the total probability is 1"""
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)

def beta_pdf(x, alpha, beta):
    if x < 0 or x > 1:          # no weight outside of [0, 1]
        return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```

Generally speaking, this distribution centers its weight at:

```
alpha / (alpha + beta)
```

and the larger `alpha` and `beta` are, the "tighter" the distribution is.

For example, if `alpha` and `beta` are both 1, it's just the uniform distribution (centered at 0.5, very dispersed). If `alpha` is much larger than `beta`, most of the weight is near 1. And if `alpha` is much smaller than `beta`, most of the weight is near zero. Figure 7-1 shows several different Beta distributions.

So let's say we assume a prior distribution on *p*. Maybe we don't want to take a stand on whether the coin is fair, and we choose `alpha` and `beta` to both equal 1. Or maybe we have a strong belief that it lands heads 55% of the time, and we choose `alpha` equals 55, `beta` equals 45.

Then we flip our coin a bunch of times and see *h* heads and *t* tails. Bayes's Theorem (and some mathematics that's too tedious for us to go through here) tells us that the posterior distribution for *p* is again a Beta distribution but with parameters `alpha + h` and `beta + t`.

NOTE

It is no coincidence that the posterior distribution was again a Beta distribution. The number of heads is given by a Binomial distribution, and the Beta is the *conjugate prior* to the Binomial distribution. This means that whenever you update a Beta prior using observations from the corresponding binomial, you will get back a Beta posterior.

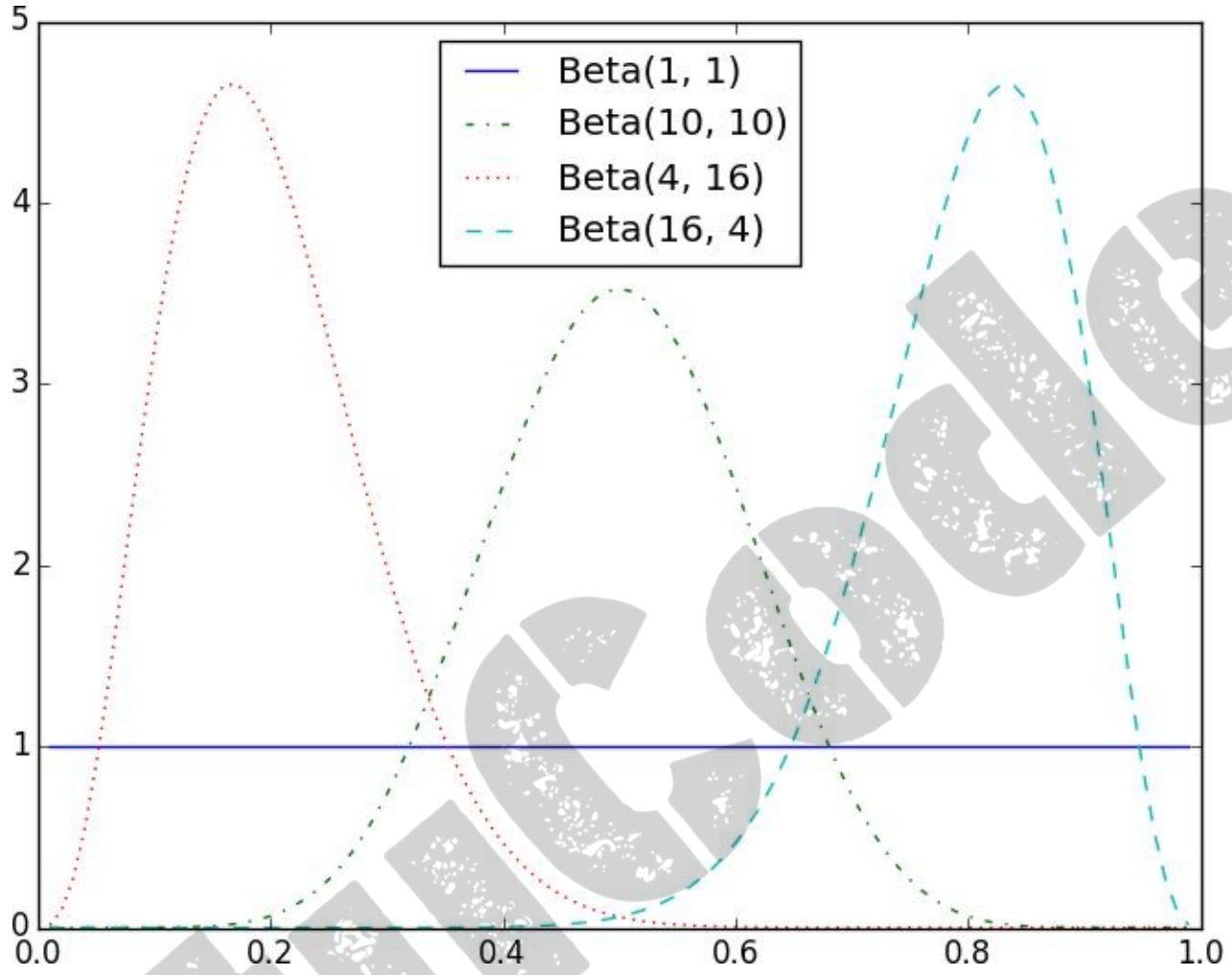

Figure 7-1. Example Beta distributions

Let's say you flip the coin 10 times and see only 3 heads.

If you started with the uniform prior (in some sense refusing to take a stand about the coin's fairness), your posterior distribution would be a Beta(4, 8), centered around 0.33. Since you considered all probabilities equally likely, your best guess is something pretty close to the observed probability.

If you started with a Beta(20, 20) (expressing the belief that the coin was roughly fair), your posterior distribution would be a Beta(23, 27), centered around 0.46, indicating a revised belief that maybe the coin is slightly biased toward tails.

And if you started with a Beta(30, 10) (expressing a belief that the coin was biased to flip 75% heads), your posterior distribution would be a Beta(33, 17), centered around 0.66. In that case you'd still believe in a heads bias, but less strongly than you did initially. These three different posteriors are plotted in Figure 7-2.

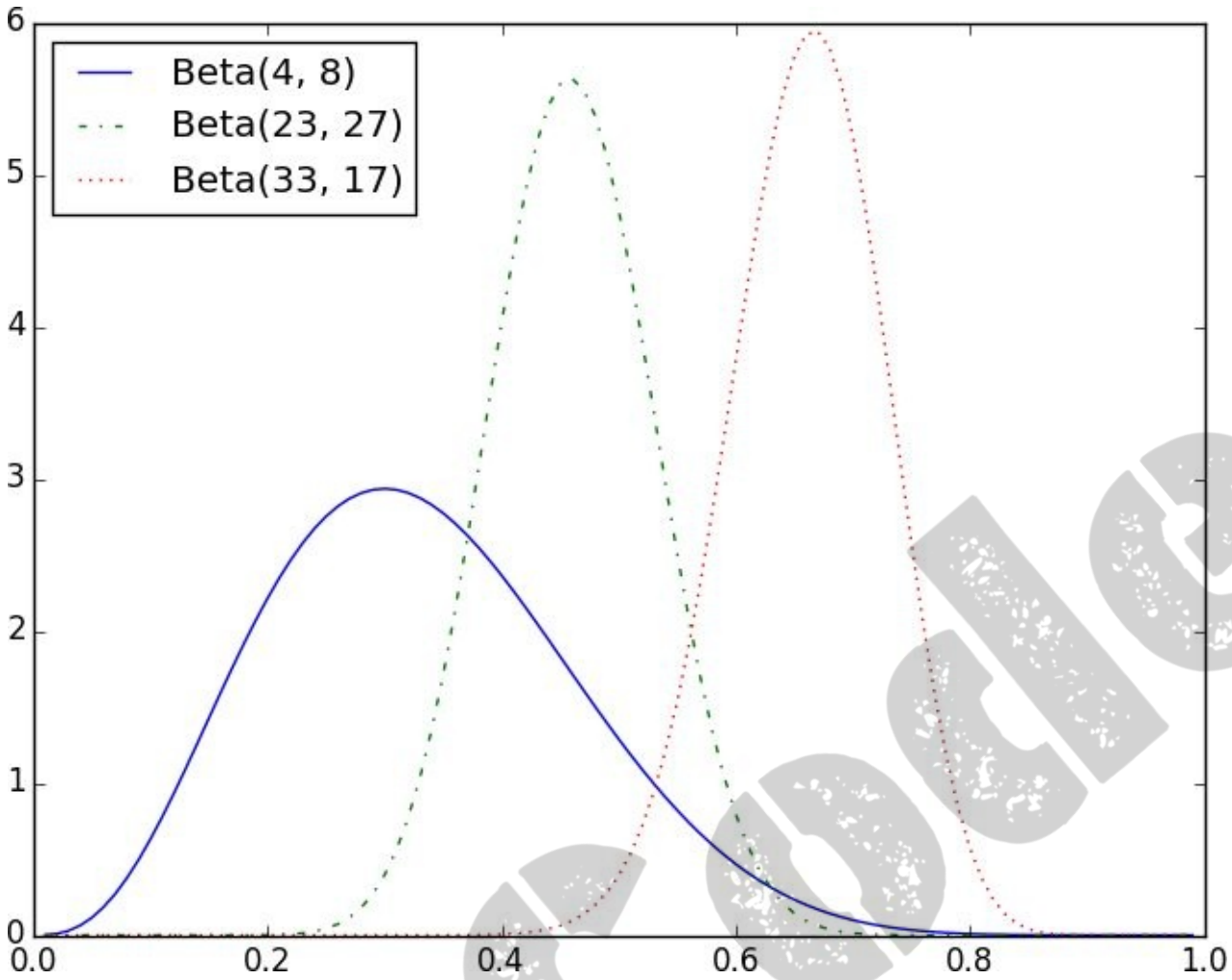

Figure 7-2. Posteriors arising from different priors

If you flipped the coin more and more times, the prior would matter less and less until eventually you'd have (nearly) the same posterior distribution no matter which prior you started with.

For example, no matter how biased you initially thought the coin was, it would be hard to maintain that belief after seeing 1,000 heads out of 2,000 flips (unless you are a lunatic who picks something like a Beta(1000000,1) prior).

What's interesting is that this allows us to make probability statements about hypotheses: "Based on the prior and the observed data, there is only a 5% likelihood the coin's heads probability is between 49% and 51%." This is philosophically very different from a statement like "if the coin were fair we would expect to observe data so extreme only 5% of the time."

Using Bayesian inference to test hypotheses is considered somewhat controversial — in part because its mathematics can get somewhat complicated, and in part because of the subjective nature of choosing a prior. We won't use it any further in this book, but it's good to know about.

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.

VuCode

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.

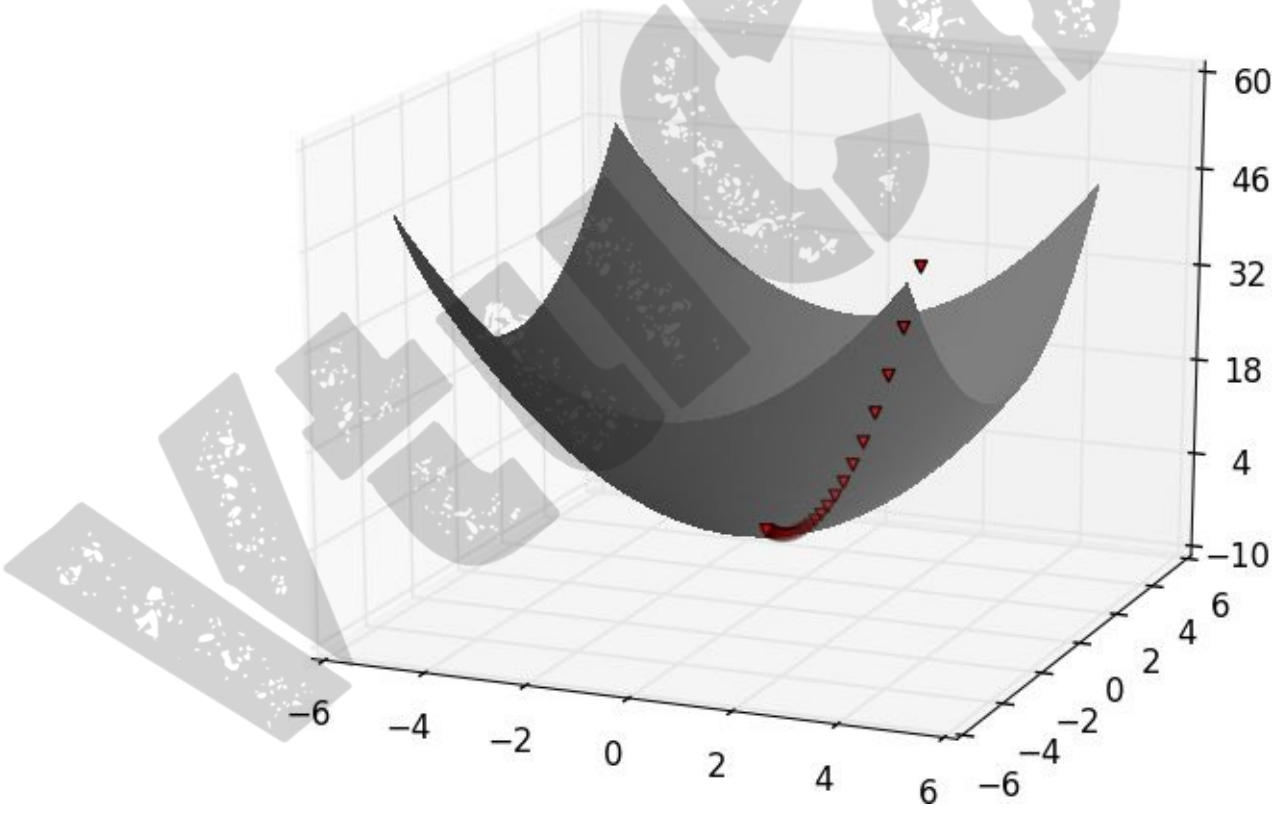

Figure 8-1. Finding a minimum using gradient descent

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.)

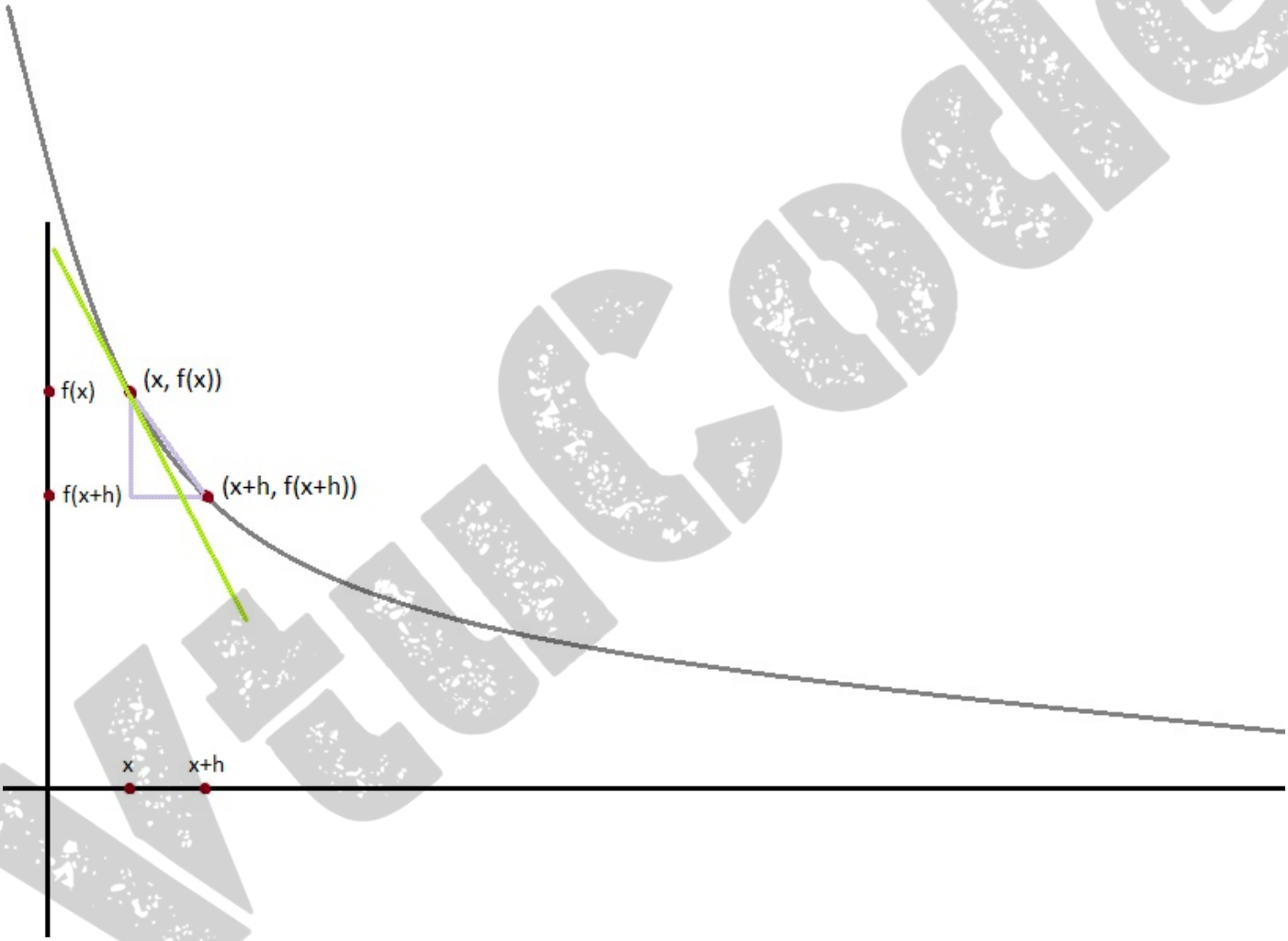

Figure 8-2. Approximating a derivative with a difference quotient

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()
```

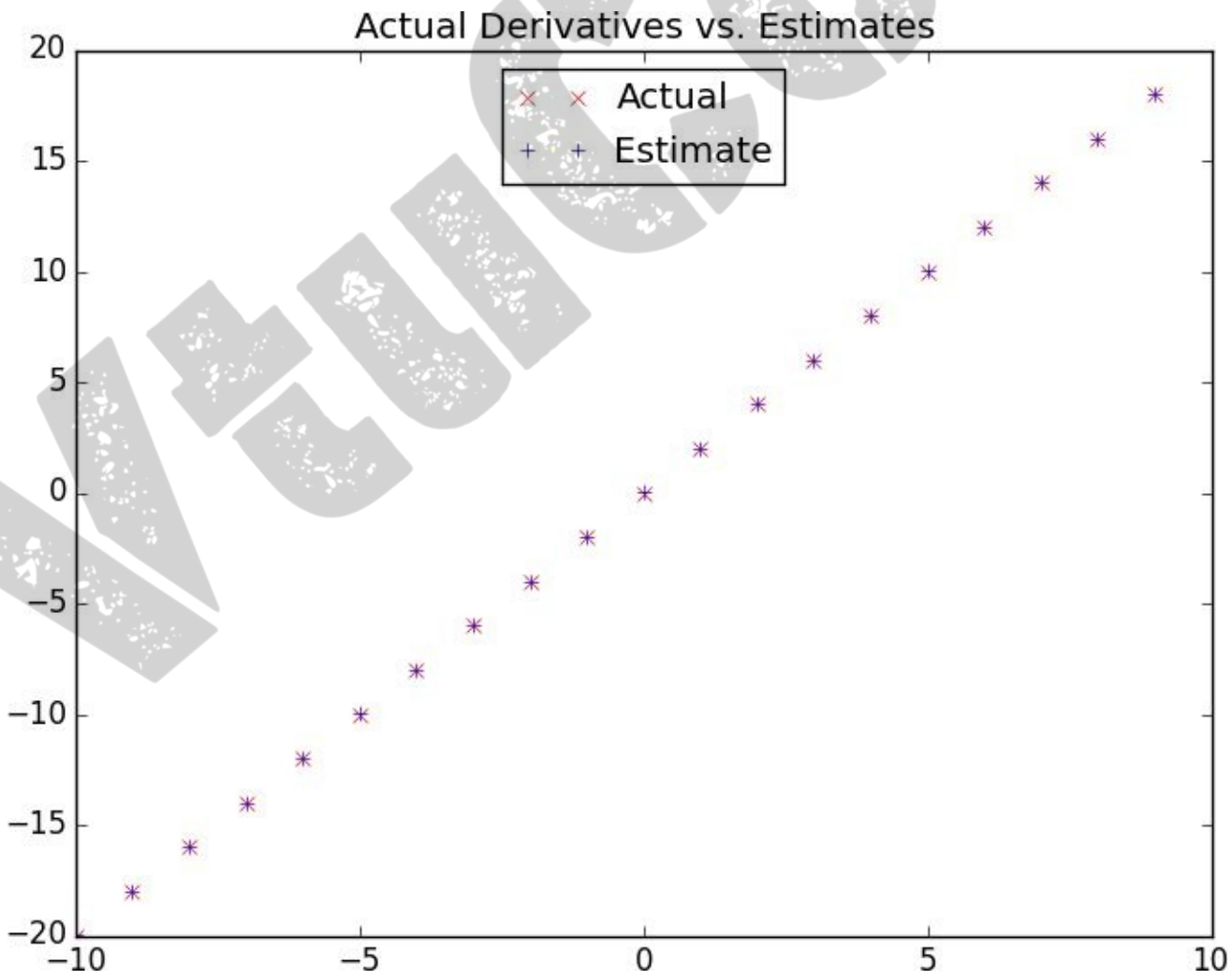

Figure 8-3. Goodness of difference quotient approximation

When `f` is a function of many variables, it has multiple *partial derivatives*, each indicating how `f` changes when we make small changes in just one of the input variables.

We calculate its *i*th partial derivative by treating it as a function of just its *i*th variable, holding the other variables fixed:

```
def partial_difference_quotient(f, v, i, h):
    """compute the ith partial difference quotient of f at v"""
    w = [v_j + (h if j == i else 0)    # add h to just the ith element of v
         for j, v_j in enumerate(v)]

    return (f(w) - f(v)) / h
```

after which we can estimate the gradient the same way:

```
def estimate_gradient(f, v, h=0.00001):
    return [partial_difference_quotient(f, v, i, h)
            for i, _ in enumerate(v)]
```

> **NOTE**
>
> A major drawback to this "estimate using difference quotients" approach is that it's computationally expensive. If `v` has length *n*, `estimate_gradient` has to evaluate `f` on *2n* different inputs. If you're repeatedly estimating gradients, you're doing a whole lot of extra work.

YuCode

Using the Gradient

It's easy to see that the `sum_of_squares` function is smallest when its input `v` is a vector of zeroes. But imagine we didn't know that. Let's use gradients to find the minimum among all three-dimensional vectors. We'll just pick a random starting point and then take tiny steps in the opposite direction of the gradient until we reach a point where the gradient is very small:

```
def step(v, direction, step_size):
    """move step_size in the direction from v"""
    return [v_i + step_size * direction_i
            for v_i, direction_i in zip(v, direction)]

def sum_of_squares_gradient(v):
    return [2 * v_i for v_i in v]

# pick a random starting point
v = [random.randint(-10,10) for i in range(3)]

tolerance = 0.0000001

while True:
    gradient = sum_of_squares_gradient(v)   # compute the gradient at v
    next_v = step(v, gradient, -0.01)       # take a negative gradient step
    if distance(next_v, v) < tolerance:     # stop if we're converging
        break
    v = next_v                              # continue if we're not
```

If you run this, you'll find that it always ends up with a `v` that's very close to `[0,0,0]`. The smaller you make the `tolerance`, the closer it will get.

Choosing the Right Step Size

Although the rationale for moving against the gradient is clear, how far to move is not. Indeed, choosing the right step size is more of an art than a science. Popular options include:

- Using a fixed step size
- Gradually shrinking the step size over time
- At each step, choosing the step size that minimizes the value of the objective function

The last sounds optimal but is, in practice, a costly computation. We can approximate it by trying a variety of step sizes and choosing the one that results in the smallest value of the objective function:

```
step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
```

It is possible that certain step sizes will result in invalid inputs for our function. So we'll need to create a "safe apply" function that returns infinity (which should never be the minimum of anything) for invalid inputs:

```
def safe(f):
    """return a new function that's the same as f,
    except that it outputs infinity whenever f produces an error"""
    def safe_f(*args, **kwargs):
        try:
            return f(*args, **kwargs)
        except:
            return float('inf')       # this means "infinity" in Python
    return safe_f
```

Putting It All Together

In the general case, we have some `target_fn` that we want to minimize, and we also have its `gradient_fn`. For example, the `target_fn` could represent the errors in a model as a function of its parameters, and we might want to find the parameters that make the errors as small as possible.

Furthermore, let's say we have (somehow) chosen a starting value for the parameters `theta_0`. Then we can implement gradient descent as:

```
def minimize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    """use gradient descent to find theta that minimizes target function"""

    step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]

    theta = theta_0                           # set theta to initial value
    target_fn = safe(target_fn)               # safe version of target_fn
    value = target_fn(theta)                  # value we're minimizing

    while True:
        gradient = gradient_fn(theta)
        next_thetas = [step(theta, gradient, -step_size)
                       for step_size in step_sizes]

        # choose the one that minimizes the error function
        next_theta = min(next_thetas, key=target_fn)
        next_value = target_fn(next_theta)

        # stop if we're "converging"
        if abs(value - next_value) < tolerance:
            return theta
        else:
            theta, value = next_theta, next_value
```

We called it `minimize_batch` because, for each gradient step, it looks at the entire data set (because `target_fn` returns the error on the whole data set). In the next section, we'll see an alternative approach that only looks at one data point at a time.

Sometimes we'll instead want to *maximize* a function, which we can do by minimizing its negative (which has a corresponding negative gradient):

```
def negate(f):
    """return a function that for any input x returns -f(x)"""
    return lambda *args, **kwargs: -f(*args, **kwargs)

def negate_all(f):
    """the same when f returns a list of numbers"""
    return lambda *args, **kwargs: [-y for y in f(*args, **kwargs)]

def maximize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    return minimize_batch(negate(target_fn),
                          negate_all(gradient_fn),
                          theta_0,
                          tolerance)
```

Stochastic Gradient Descent

As we mentioned before, often we'll be using gradient descent to choose the parameters of a model in a way that minimizes some notion of error. Using the previous batch approach, each gradient step requires us to make a prediction and compute the gradient for the whole data set, which makes each step take a long time.

Now, usually these error functions are *additive*, which means that the predictive error on the whole data set is simply the sum of the predictive errors for each data point.

When this is the case, we can instead apply a technique called *stochastic gradient descent*, which computes the gradient (and takes a step) for only one point at a time. It cycles over our data repeatedly until it reaches a stopping point.

During each cycle, we'll want to iterate through our data in a random order:

```
def in_random_order(data):
    """generator that returns the elements of data in random order"""
    indexes = [i for i, _ in enumerate(data)]  # create a list of indexes
    random.shuffle(indexes)                    # shuffle them
    for i in indexes:                          # return the data in that order
        yield data[i]
```

And we'll want to take a gradient step for each data point. This approach leaves the possibility that we might circle around near a minimum forever, so whenever we stop getting improvements we'll decrease the step size and eventually quit:

```
def minimize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):

    data = zip(x, y)
    theta = theta_0                            # initial guess
    alpha = alpha_0                            # initial step size
    min_theta, min_value = None, float("inf")   # the minimum so far
    iterations_with_no_improvement = 0

    # if we ever go 100 iterations with no improvement, stop
    while iterations_with_no_improvement < 100:
        value = sum( target_fn(x_i, y_i, theta) for x_i, y_i in data )

        if value < min_value:
            # if we've found a new minimum, remember it
            # and go back to the original step size
            min_theta, min_value = theta, value
            iterations_with_no_improvement = 0
            alpha = alpha_0
        else:
            # otherwise we're not improving, so try shrinking the step size
            iterations_with_no_improvement += 1
            alpha *= 0.9

        # and take a gradient step for each of the data points
        for x_i, y_i in in_random_order(data):
            gradient_i = gradient_fn(x_i, y_i, theta)
            theta = vector_subtract(theta, scalar_multiply(alpha, gradient_i))

    return min_theta
```

The stochastic version will typically be a lot faster than the batch version. Of course, we'll want a version that maximizes as well:

```
def maximize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.01):
    return minimize_stochastic(negate(target_fn),
                               negate_all(gradient_fn),
                               x, y, theta_0, alpha_0)
```

VuCode

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.

vtucode

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.

vtucode

The Basics of Text Files

The first step to working with a text file is to obtain a *file object* using `open`:

```
# 'r' means read-only
file_for_reading = open('reading_file.txt', 'r')

# 'w' is write—will destroy the file if it already exists!
file_for_writing = open('writing_file.txt', 'w')

# 'a' is append—for adding to the end of the file
file_for_appending = open('appending_file.txt', 'a')

# don't forget to close your files when you're done
file_for_writing.close()
```

Because it is easy to forget to close your files, you should always use them in a `with` block, at the end of which they will be closed automatically:

```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)

# at this point f has already been closed, so don't try to use it
process(data)
```

If you need to read a whole text file, you can just iterate over the lines of the file using `for`:

```
starts_with_hash = 0

with open('input.txt','r') as f:
    for line in file:                # look at each line in the file
        if re.match("^#",line):      # use a regex to see if it starts with '#'
            starts_with_hash += 1    # if it does, add 1 to the count
```

Every line you get this way ends in a newline character, so you'll often want to `strip()` it before doing anything with it.

For example, imagine you have a file full of email addresses, one per line, and that you need to generate a histogram of the domains. The rules for correctly extracting domains are somewhat subtle (e.g., the Public Suffix List), but a good first approximation is to just take the parts of the email addresses that come after the `@`. (Which gives the wrong answer for email addresses like `joel@mail.datasciencester.com`.)

```
def get_domain(email_address):
    """split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

Delimited Files

The hypothetical email addresses file we just processed had one address per line. More frequently you'll work with files with lots of data on each line. These files are very often either *comma-separated* or *tab-separated*. Each line has several fields, with a comma (or a tab) indicating where one field ends and the next field starts.

This starts to get complicated when you have fields with commas and tabs and newlines in them (which you inevitably do). For this reason, it's pretty much always a mistake to try to parse them yourself. Instead, you should use Python's `csv` module (or the `pandas` library). For technical reasons that you should feel free to blame on Microsoft, you should always work with `csv` files in *binary* mode by including a *b* after the *r* or *w* (see Stack Overflow).

If your file has no headers (which means you probably want each row as a `list`, and which places the burden on you to know what's in each column), you can use `csv.reader` to iterate over the rows, each of which will be an appropriately split list.

For example, if we had a tab-delimited file of stock prices:

```
6/20/2014	AAPL	90.91
6/20/2014	MSFT	41.68
6/20/2014	FB	64.5
6/19/2014	AAPL	91.86
6/19/2014	MSFT	41.51
6/19/2014	FB	64.34
```

we could process them with:

```
import csv

with open('tab_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

If your file has headers:

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

you can either skip the header row (with an initial call to `reader.next()`) or get each row as a `dict` (with the headers as keys) by using `csv.DictReader`:

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.DictReader(f, delimiter=':')
    for row in reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(row["closing_price"])
        process(date, symbol, closing_price)
```

Even if your file doesn't have headers you can still use `DictReader` by passing it the keys as a `fieldnames` parameter.

You can similarly write out delimited data using `csv.writer`:

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

`csv.writer` will do the right thing if your fields themselves have commas in them. Your own hand-rolled writer probably won't. For example, if you attempt:

```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
           ["test3", "failure, kind of", "Wednesday"],
           ["test4", "failure, utter", "Thursday"]]

# don't do this!
with open('bad_csv.txt', 'wb') as f:
    for row in results:
        f.write(",".join(map(str, row))) # might have too many commas in it!
        f.write("\n")                    # row might have newlines as well!
```

You will end up with a `csv` file that looks like:

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

and that no one will ever be able to make sense of.

Scraping the Web

Another way to get data is by scraping it from web pages. Fetching web pages, it turns out, is pretty easy; getting meaningful structured information out of them less so.

vtucode

HTML and the Parsing Thereof

Pages on the Web are written in HTML, in which text is (ideally) marked up into elements and their attributes:

```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

In a perfect world, where all web pages are marked up semantically for our benefit, we would be able to extract data using rules like "find the `<p>` element whose `id` is `subject` and return the text it contains." In the actual world, HTML is not generally well-formed, let alone annotated. This means we'll need help making sense of it.

To get data out of HTML, we will use the BeautifulSoup library, which builds a tree out of the various elements on a web page and provides a simple interface for accessing them. As I write this, the latest version is Beautiful Soup 4.3.2 (`pip install beautifulsoup4`), which is what we'll be using. We'll also be using the requests library (`pip install requests`), which is a much nicer way of making HTTP requests than anything that's built into Python.

Python's built-in HTML parser is not that lenient, which means that it doesn't always cope well with HTML that's not perfectly formed. For that reason, we'll use a different parser, which we need to install:

```
pip install html5lib
```

To use Beautiful Soup, we'll need to pass some HTML into the `BeautifulSoup()` function. In our examples, this will be the result of a call to `requests.get`:

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

after which we can get pretty far using a few simple methods.

We'll typically work with `Tag` objects, which correspond to the tags representing the structure of an HTML page.

For example, to find the first `<p>` tag (and its contents) you can use:

```
first_paragraph = soup.find('p')        # or just soup.p
```

You can get the text contents of a `Tag` using its `text` property:

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

And you can extract a tag's attributes by treating it like a `dict`:

```
first_paragraph_id = soup.p['id']       # raises KeyError if no 'id'
first_paragraph_id2 = soup.p.get('id')  # returns None if no 'id'
```

You can get multiple tags at once:

```
all_paragraphs = soup.find_all('p')  # or just soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

Frequently you'll want to find tags with a specific `class`:

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

And you can combine these to implement more elaborate logic. For example, if you want to find every `<span>` element that is contained inside a `<div>` element, you could do this:

```
# warning, will return the same span multiple times
# if it sits inside multiple divs
# be more clever if that's the case
spans_inside_divs = [span
                     for div in soup('div')     # for each <div> on the page
                     for span in div('span')]   # find each <span> inside it
```

Just this handful of features will allow us to do quite a lot. If you end up needing to do more-complicated things (or if you're just curious), check the documentation.

Of course, whatever data is important won't typically be labeled as `class="important"`. You'll need to carefully inspect the source HTML, reason through your selection logic, and worry about edge cases to make sure your data is correct. Let's look at an example.

Example: O'Reilly Books About Data

A potential investor in DataSciencester thinks data is just a fad. To prove him wrong, you decide to examine how many data books O'Reilly has published over time. After digging through its website, you find that it has many pages of data books (and videos), reachable through 30-items-at-a-time directory pages with URLs like:

```
http://shop.oreilly.com/category/browse-subjects/data.do?
sortby=publicationDate&page=1
```

Unless you want to be a jerk (and unless you want your scraper to get banned), whenever you want to scrape data from a website you should first check to see if it has some sort of access policy. Looking at:

```
http://oreilly.com/terms/
```

there seems to be nothing prohibiting this project. In order to be good citizens, we should also check for a *robots.txt* file that tells webcrawlers how to behave. The important lines in `http://shop.oreilly.com/robots.txt` are:

```
Crawl-delay: 30
Request-rate: 1/30
```

The first tells us that we should wait 30 seconds between requests, the second that we should request only one page every 30 seconds. So basically they're two different ways of saying the same thing. (There are other lines that indicate directories not to scrape, but they don't include our URL, so we're OK there.)

> **NOTE**
>
> There's always the possibility that O'Reilly will at some point revamp its website and break all the logic in this section. I will do what I can to prevent that, of course, but I don't have a ton of influence over there. Although, if every one of you were to convince everyone you know to buy a copy of this book…

To figure out how to extract the data, let's download one of those pages and feed it to Beautiful Soup:

```
# you don't have to split the url like this unless it needs to fit in a book
url = "http://shop.oreilly.com/category/browse-subjects/" + \
      "data.do?sortby=publicationDate&page=1"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
```

If you view the source of the page (in your browser, right-click and select "View source" or "View page source" or whatever option looks the most like that), you'll see that each book (or video) seems to be uniquely contained in a `<td>` table cell element whose `class` is `thumbtext`. Here is (an abridged version of) the relevant HTML for one book:

```
<td class="thumbtext">
  <div class="thumbcontainer">
    <div class="thumbdiv">
      <a href="/product/9781118903407.do">
```

```
        <img src="..."/>
      </a>
    </div>
  </div>
  <div class="widthchange">
    <div class="thumbheader">
      <a href="/product/9781118903407.do">Getting a Big Data Job For Dummies</a>
    </div>
    <div class="AuthorName">By Jason Williamson</div>
    <span class="directorydate">        December 2014    </span>
    <div style="clear:both;">
      <div id="146350">
        <span class="pricelabel">
                          Ebook:

                          <span class="price"> $29.99</span>
        </span>
      </div>
    </div>
  </div>
</td>
```

A good first step is to find all of the `td thumbtext` tag elements:

```
tds = soup('td', 'thumbtext')
print len(tds)
# 30
```

Next we'd like to filter out the videos. (The would-be investor is only impressed by books.) If we inspect the HTML further, we see that each `td` contains one or more `span` elements whose `class` is `pricelabel`, and whose text looks like `Ebook:` or `Video:` or `Print:`. It appears that the videos contain only one `pricelabel`, whose `text` starts with `Video` (after removing leading spaces). This means we can test for videos with:

```
def is_video(td):
    """it's a video if it has exactly one pricelabel, and if
    the stripped text inside that pricelabel starts with 'Video'"""
    pricelabels = td('span', 'pricelabel')
    return (len(pricelabels) == 1 and
            pricelabels[0].text.strip().startswith("Video"))

print len([td for td in tds if not is_video(td)])
# 21 for me, might be different for you
```

Now we're ready to start pulling data out of the `td` elements. It looks like the book title is the text inside the `<a>` tag inside the `<div class="thumbheader">`:

```
title = td.find("div", "thumbheader").a.text
```

The author(s) are in the text of the `AuthorName` `<div>`. They are prefaced by a `By` (which we want to get rid of) and separated by commas (which we want to split out, after which we'll need to get rid of spaces):

```
author_name = td.find('div', 'AuthorName').text
authors = [x.strip() for x in re.sub("^By ", "", author_name).split(",")]
```

The ISBN seems to be contained in the link that's in the `thumbheader` `<div>`:

```
    isbn_link = td.find("div", "thumbheader").a.get("href")

    # re.match captures the part of the regex in parentheses
    isbn = re.match("/product/(.*)\.do", isbn_link).group(1)
```

And the date is just the contents of the `<span class="directorydate">`:

```
    date = td.find("span", "directorydate").text.strip()
```

Let's put this all together into a function:

```
def book_info(td):
    """given a BeautifulSoup <td> Tag representing a book,
    extract the book's details and return a dict"""

    title = td.find("div", "thumbheader").a.text
    by_author = td.find('div', 'AuthorName').text
    authors = [x.strip() for x in re.sub("^By ", "", by_author).split(",")]
    isbn_link = td.find("div", "thumbheader").a.get("href")
    isbn = re.match("/product/(.*)\.do", isbn_link).groups()[0]
    date = td.find("span", "directorydate").text.strip()

    return {
        "title" : title,
        "authors" : authors,
        "isbn" : isbn,
        "date" : date
    }
```

And now we're ready to scrape:

```
from bs4 import BeautifulSoup
import requests
from time import sleep
base_url = "http://shop.oreilly.com/category/browse-subjects/" + \
           "data.do?sortby=publicationDate&page="

books = []

NUM_PAGES = 31     # at the time of writing, probably more by now

for page_num in range(1, NUM_PAGES + 1):
    print "souping page", page_num, ",", len(books), " found so far"
    url = base_url + str(page_num)
    soup = BeautifulSoup(requests.get(url).text, 'html5lib')

    for td in soup('td', 'thumbtext'):
        if not is_video(td):
            books.append(book_info(td))

    # now be a good citizen and respect the robots.txt!
    sleep(30)
```

> **NOTE**
>
> Extracting data from HTML like this is more data art than data science. There are countless other find-the-books and find-the-title logics that would have worked just as well.

Now that we've collected the data, we can plot the number of books published each year (Figure 9-1):

```
def get_year(book):
    """book["date"] looks like 'November 2014' so we need to
    split on the space and then take the second piece"""
```

```
    return int(book["date"].split()[1])

# 2014 is the last complete year of data (when I ran this)
year_counts = Counter(get_year(book) for book in books
                      if get_year(book) <= 2014)

import matplotlib.pyplot as plt
years = sorted(year_counts)
book_counts = [year_counts[year] for year in years]
plt.plot(years, book_counts)
plt.ylabel("# of data books")
plt.title("Data is Big!")
plt.show()
```

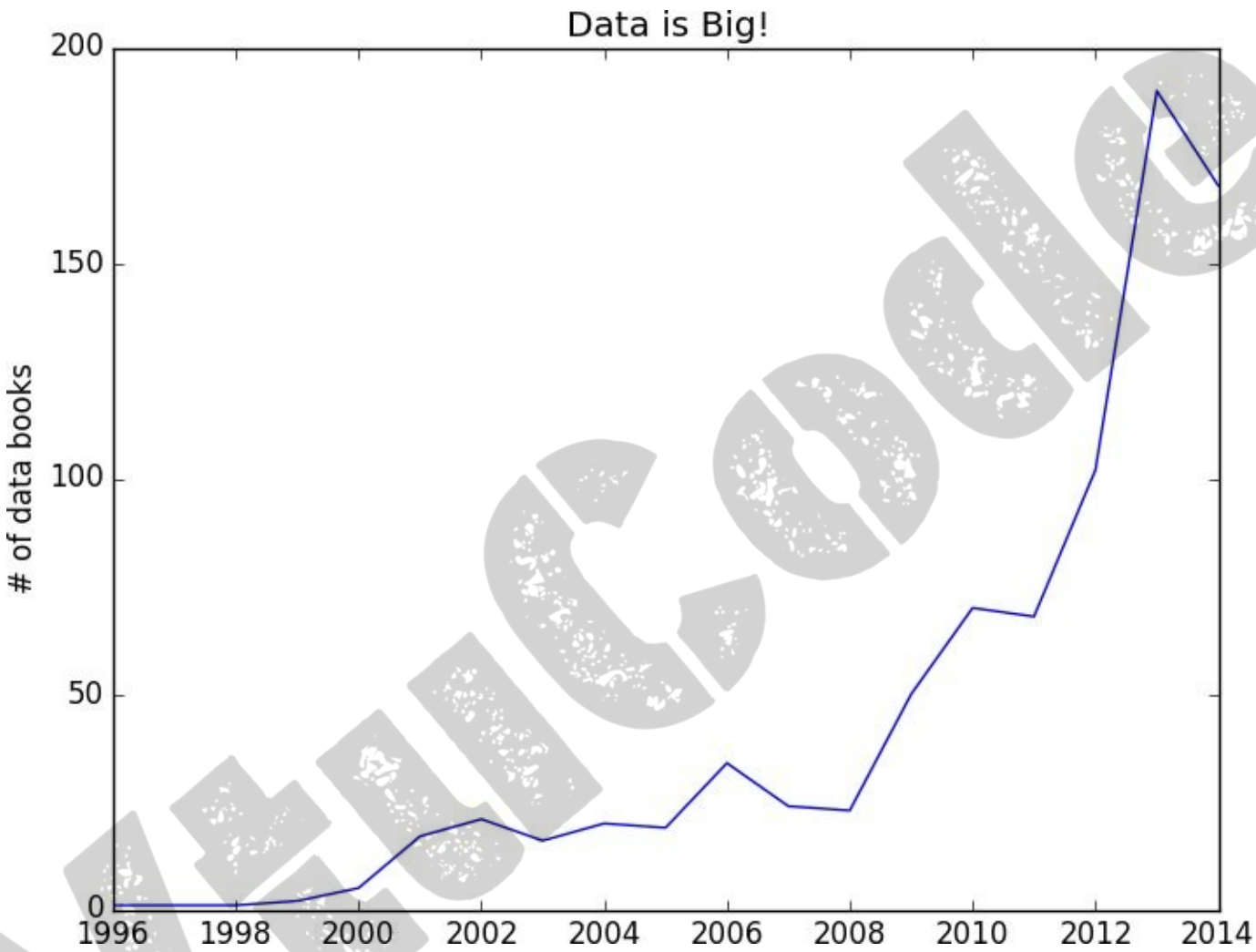

Figure 9-1. Number of data books per year

Unfortunately, the would-be investor looks at the graph and decides that 2013 was “peak data.”

Using APIs

Many websites and web services provide application programming interfaces (APIs), which allow you to explicitly request data in a structured format. This saves you the trouble of having to scrape them!

vtucode

JSON (and XML)

Because HTTP is a protocol for transferring *text*, the data you request through a web API needs to be *serialized* into a string format. Often this serialization uses JavaScript Object Notation (JSON). JavaScript objects look quite similar to Python `dicts`, which makes their string representations easy to interpret:

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2014,
  "topics" : [ "data", "science", "data science"] }
```

We can parse JSON using Python's `json` module. In particular, we will use its `loads` function, which deserializes a string representing a JSON object into a Python object:

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2014,
                  "topics" : [ "data", "science", "data science"] }"""

# parse the JSON to create a Python dict
deserialized = json.loads(serialized)
if "data science" in deserialized["topics"]:
    print deserialized
```

Sometimes an API provider hates you and only provides responses in XML:

```
<Book>
  <Title>Data Science Book</Title>
  <Author>Joel Grus</Author>
  <PublicationYear>2014</PublicationYear>
  <Topics>
    <Topic>data</Topic>
    <Topic>science</Topic>
    <Topic>data science</Topic>
  </Topics>
</Book>
```

You can use `BeautifulSoup` to get data from XML similarly to how we used it to get data from HTML; check its documentation for details.

Using an Unauthenticated API

Most APIs these days require you to first authenticate yourself in order to use them. While we don't begrudge them this policy, it creates a lot of extra boilerplate that muddies up our exposition. Accordingly, we'll first take a look at GitHub's API, with which you can do some simple things unauthenticated:

```
import requests, json
endpoint = "https://api.github.com/users/joelgrus/repos"

repos = json.loads(requests.get(endpoint).text)
```

At this point `repos` is a `list` of Python `dicts`, each representing a public repository in my GitHub account. (Feel free to substitute your username and get your GitHub repository data instead. You do have a GitHub account, right?)

We can use this to figure out which months and days of the week I'm most likely to create a repository. The only issue is that the dates in the response are (Unicode) strings:

```
u'created_at': u'2013-07-05T02:02:28Z'
```

Python doesn't come with a great date parser, so we'll need to install one:

```
pip install python-dateutil
```

from which you'll probably only ever need the `dateutil.parser.parse` function:

```
from dateutil.parser import parse

dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

Similarly, you can get the languages of my last five repositories:

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["created_at"],
                             reverse=True)[:5]

last_5_languages = [repo["language"]
                    for repo in last_5_repositories]
```

Typically we won't be working with APIs at this low "make the requests and parse the responses ourselves" level. One of the benefits of using Python is that someone has already built a library for pretty much any API you're interested in accessing. When they're done well, these libraries can save you a lot of the trouble of figuring out the hairier details of API access. (When they're not done well, or when it turns out they're based on defunct versions of the corresponding APIs, they can cause you enormous headaches.)

Nonetheless, you'll occasionally have to roll your own API-access library (or, more likely,

debug why someone else’s isn’t working), so it’s good to know some of the details.

VtuCode

Finding APIs

If you need data from a specific site, look for a developers or API section of the site for details, and try searching the Web for “python __ api” to find a library. There is a Rotten Tomatoes API for Python. There are multiple Python wrappers for the Klout API, for the Yelp API, for the IMDB API, and so on.

If you’re looking for lists of APIs that have Python wrappers, two directories are at Python API and Python for Beginners.

If you want a directory of web APIs more broadly (without Python wrappers necessarily), a good resource is Programmable Web, which has a huge directory of categorized APIs.

And if after all that you can’t find what you need, there’s always scraping, the last refuge of the data scientist.

vtucode

Example: Using the Twitter APIs

Twitter is a fantastic source of data to work with. You can use it to get real-time news. You can use it to measure reactions to current events. You can use it to find links related to specific topics. You can use it for pretty much anything you can imagine, just as long as you can get access to its data. And you can get access to its data through its API.

To interact with the Twitter APIs we'll be using the Twython library (`pip install twython`). There are quite a few Python Twitter libraries out there, but this is the one that I've had the most success working with. You are encouraged to explore the others as well!

YuCode

Getting Credentials

In order to use Twitter's APIs, you need to get some credentials (for which you need a Twitter account, which you should have anyway so that you can be part of the lively and friendly Twitter `#datascience` community). Like all instructions that relate to websites that I don't control, these may go obsolete at some point but will hopefully work for a while. (Although they have already changed at least once while I was writing this book, so good luck!)

1. Go to *https://apps.twitter.com/*.
2. If you are not signed in, click Sign in and enter your Twitter username and password.
3. Click Create New App.
4. Give it a name (such as "Data Science") and a description, and put any URL as the website (it doesn't matter which one).
5. Agree to the Terms of Service and click Create.
6. Take note of the consumer key and consumer secret.
7. Click "Create my access token."
8. Take note of the access token and access token secret (you may have to refresh the page).

The consumer key and consumer secret tell Twitter what application is accessing its APIs, while the access token and access token secret tell Twitter *who* is accessing its APIs. If you have ever used your Twitter account to log in to some other site, the "click to authorize" page was generating an access token for that site to use to convince Twitter that it was you (or, at least, acting on your behalf). As we don't need this "let anyone log in" functionality, we can get by with the statically generated access token and access token secret.

Caution

The consumer key/secret and access token key/secret should be treated like *passwords*. You shouldn't share them, you shouldn't publish them in your book, and you shouldn't check them into your public GitHub repository. One simple solution is to store them in a *credentials.json* file that doesn't get checked in, and to have your code use `json.loads` to retrieve them.

Using Twython

First we'll look at the Search API, which requires only the consumer key and secret, not the access token or secret:

```
from twython import Twython

twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)

# search for tweets containing the phrase "data science"
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"].encode('utf-8')
    text = status["text"].encode('utf-8')
    print user, ":", text
    print
```

> **NOTE**
>
> The `.encode("utf-8")` is necessary to deal with the fact that tweets often contain Unicode characters that `print` can't deal with. (If you leave it out, you will very likely get a `UnicodeEncodeError`.)
>
> It is almost certain that at some point in your data science career you will run into some serious Unicode problems, at which point you will need to refer to the Python documentation or else grudgingly start using Python 3, which plays much more nicely with Unicode text.

If you run this, you should get some tweets back like:

```
haithemnyc: Data scientists with the technical savvy &amp; analytical chops to
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment for
@rdpeng in @coursera data science specialization.  So easy and Awesome.
```

This isn't that interesting, largely because the Twitter Search API just shows you whatever handful of recent results it feels like. When you're doing data science, more often you want a lot of tweets. This is where the Streaming API is useful. It allows you to connect to (a sample of) the great Twitter firehose. To use it, you'll need to authenticate using your access tokens.

In order to access the Streaming API with Twython, we need to define a class that inherits from `TwythonStreamer` and that overrides its `on_success` method (and possibly its `on_error` method):

```
from twython import TwythonStreamer

# appending data to a global variable is pretty poor form
# but it makes the example much simpler
tweets = []

class MyStreamer(TwythonStreamer):
    """our own subclass of TwythonStreamer that specifies
    how to interact with the stream"""

    def on_success(self, data):
        """what do we do when twitter sends us data?
        here data will be a Python dict representing a tweet"""

        # only want to collect English-language tweets
        if data['lang'] == 'en':
            tweets.append(data)
            print "received tweet #", len(tweets)

        # stop when we've collected enough
        if len(tweets) >= 1000:
            self.disconnect()

    def on_error(self, status_code, data):
```

```
        print status_code, data
        self.disconnect()
```

MyStreamer will connect to the Twitter stream and wait for Twitter to feed it data. Each time it receives some data (here, a Tweet represented as a Python object) it passes it to the `on_success` method, which appends it to our `tweets` list if its language is English, and then disconnects the streamer after it's collected 1,000 tweets.

All that's left is to initialize it and start it running:

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

# starts consuming public statuses that contain the keyword 'data'
stream.statuses.filter(track='data')

# if instead we wanted to start consuming a sample of *all* public statuses
# stream.statuses.sample()
```

This will run until it collects 1,000 tweets (or until it encounters an error) and stop, at which point you can start analyzing those tweets. For instance, you could find the most common hashtags with:

```
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet["entities"]["hashtags"])

print top_hashtags.most_common(5)
```

Each tweet contains a lot of data. You can either poke around yourself or dig through the Twitter API documentation.

> **NOTE**
>
> In a non-toy project you probably wouldn't want to rely on an in-memory `list` for storing the tweets. Instead you'd want to save them to a file or a database, so that you'd have them permanently.

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.

vtucode

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.

vtucode

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.

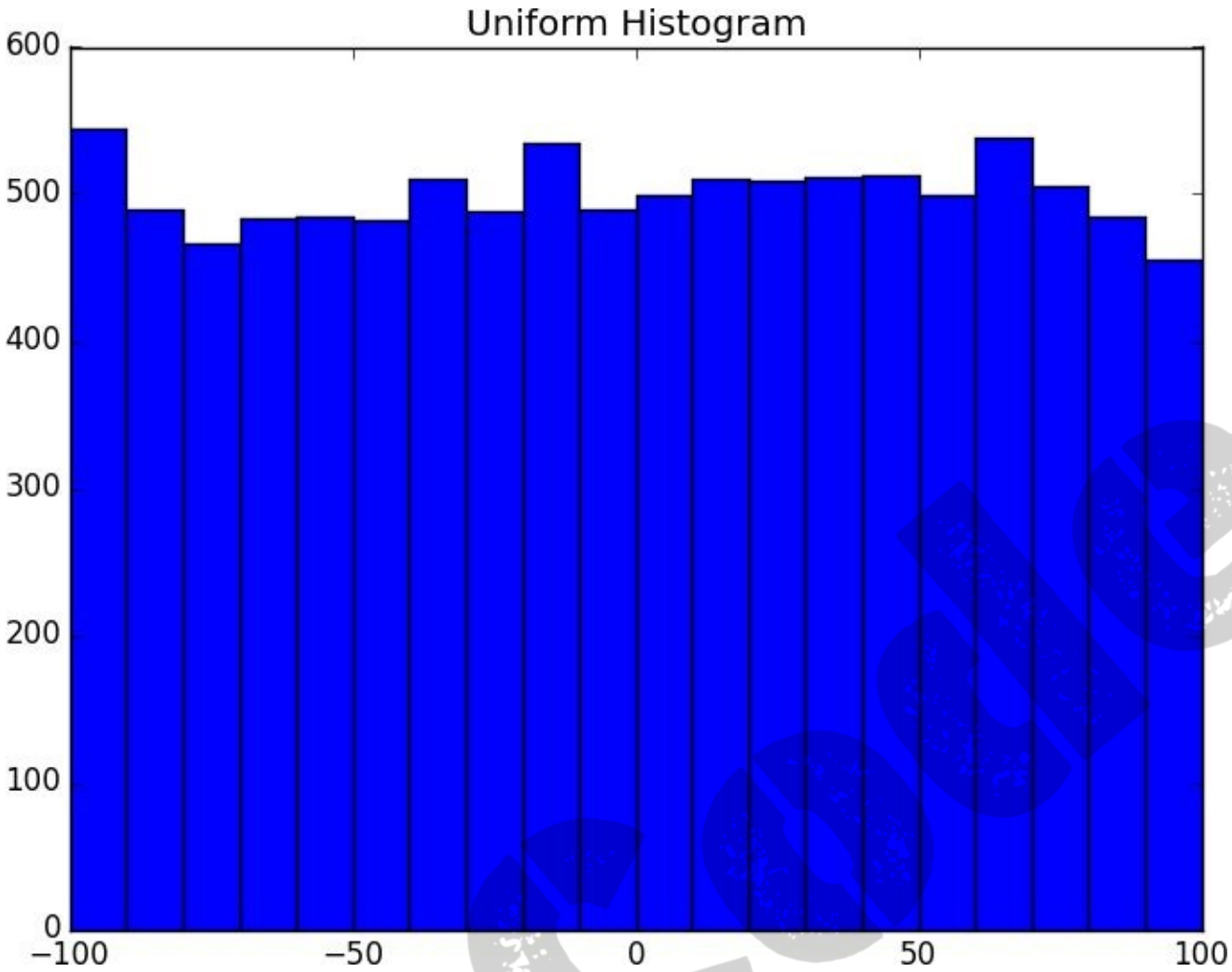

Figure 10-1. Histogram of `uniform`

Two Dimensions

Now imagine you have a data set with two dimensions. Maybe in addition to daily minutes you have years of data science experience. Of course you'd want to understand each dimension individually. But you probably also want to scatter the data.

For example, consider another fake data set:

```
def random_normal():
    """returns a random draw from a standard normal distribution"""
    return inverse_normal_cdf(random.random())

xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

If you were to run `plot_histogram` on `ys1` and `ys2` you'd get very similar looking plots (indeed, both are normally distributed with the same mean and standard deviation).

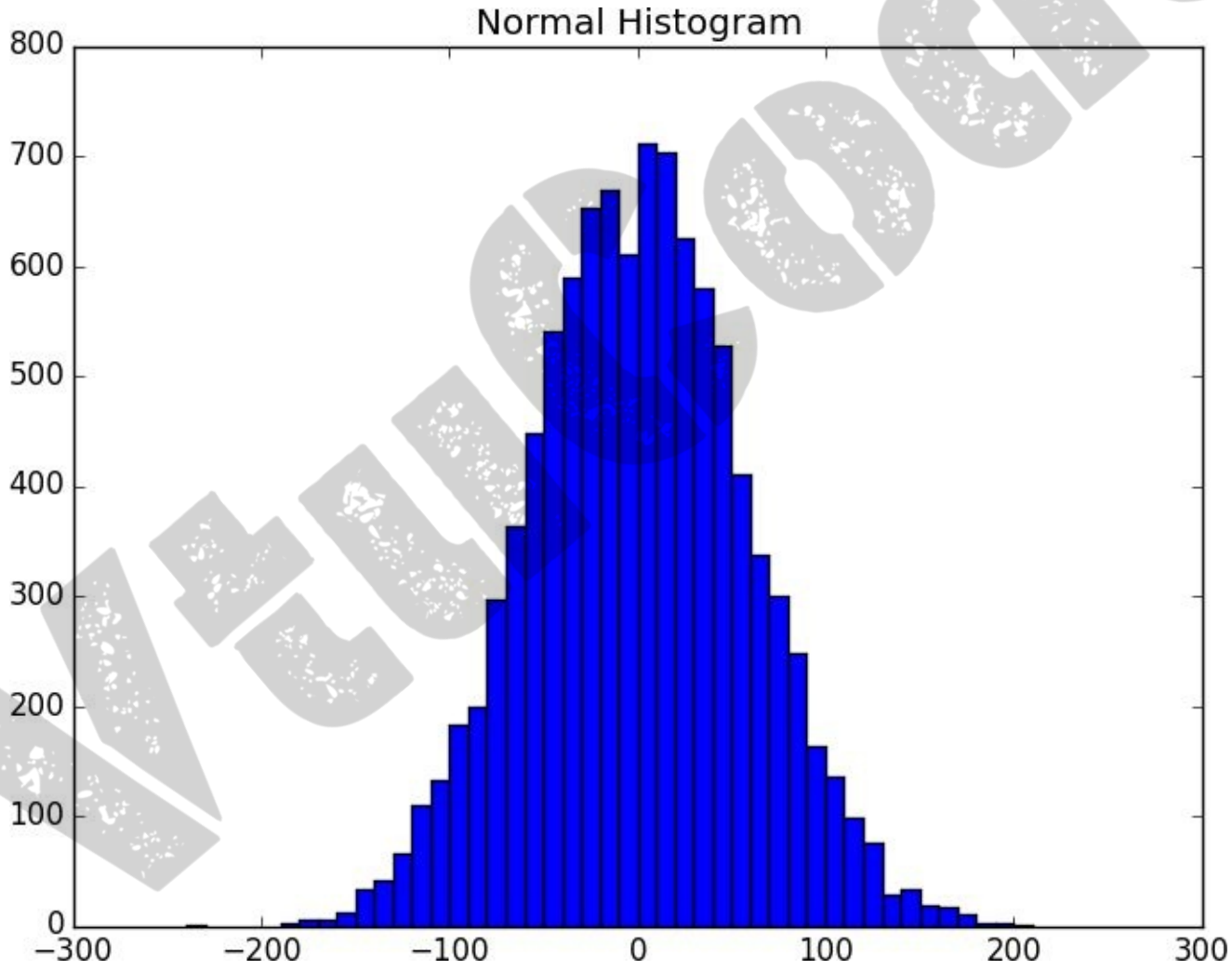

Figure 10-2. Histogram of `normal`

But each has a very different joint distribution with `xs`, as shown in Figure 10-3:

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray',  label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
```

```
plt.title("Very Different Joint Distributions")
plt.show()
```

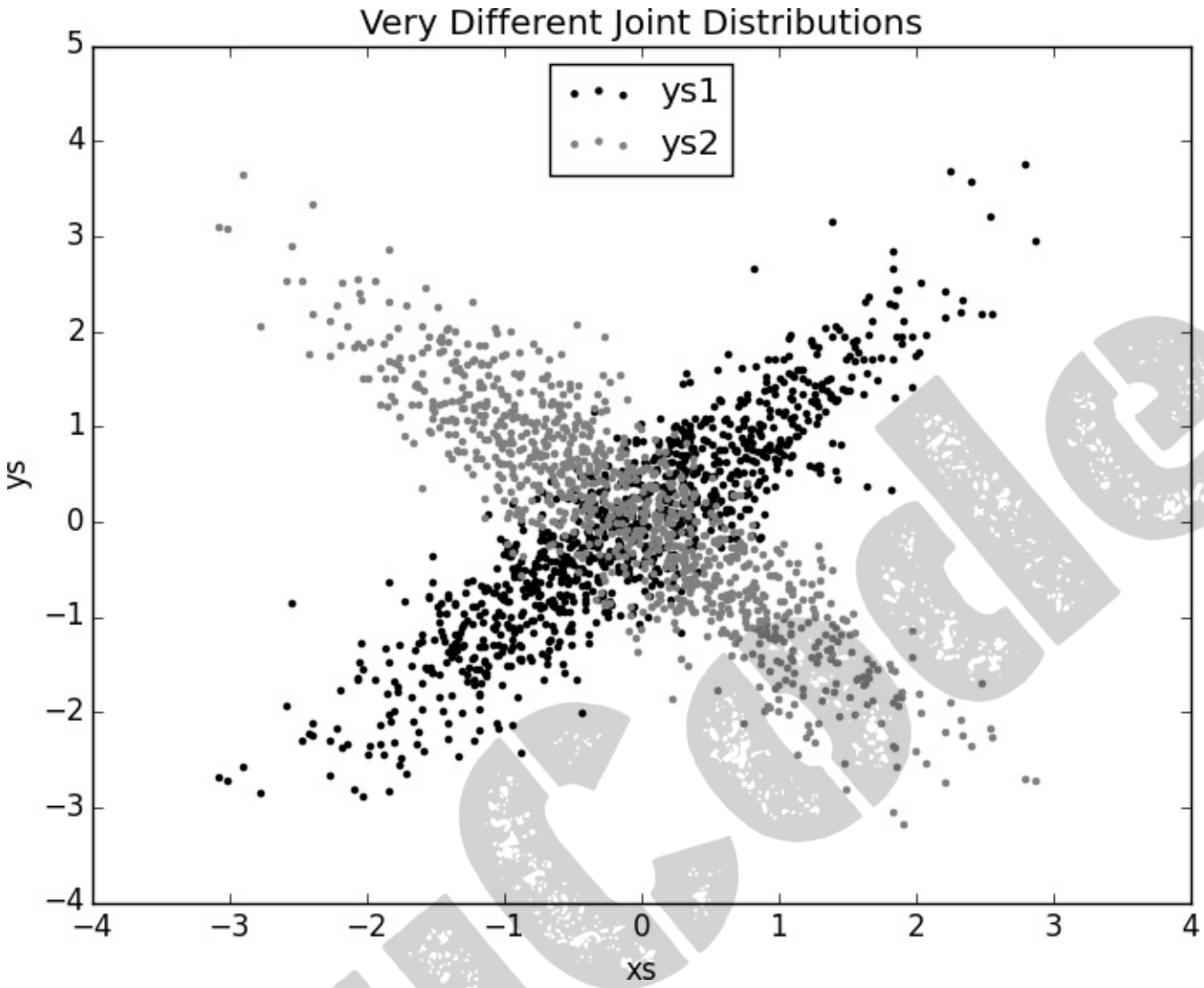

Figure 10-3. Scattering two different ys

This difference would also be apparent if you looked at the correlations:

```
print correlation(xs, ys1)      #  0.9
print correlation(xs, ys2)      # -0.9
```

Many Dimensions

With many dimensions, you'd like to know how all the dimensions relate to one another. A simple approach is to look at the *correlation matrix*, in which the entry in row *i* and column *j* is the correlation between the *i*th dimension and the *j*th dimension of the data:

```
def correlation_matrix(data):
    """returns the num_columns x num_columns matrix whose (i, j)th entry
    is the correlation between columns i and j of data"""

    _, num_columns = shape(data)

    def matrix_entry(i, j):
        return correlation(get_column(data, i), get_column(data, j))

    return make_matrix(num_columns, num_columns, matrix_entry)
```

A more visual approach (if you don't have too many dimensions) is to make a *scatterplot matrix* (Figure 10-4) showing all the pairwise scatterplots. To do that we'll use `plt.subplots()`, which allows us to create subplots of our chart. We give it the number of rows and the number of columns, and it returns a `figure` object (which we won't use) and a two-dimensional array of `axes` objects (each of which we'll plot to):

```
import matplotlib.pyplot as plt

_, num_columns = shape(data)
fig, ax = plt.subplots(num_columns, num_columns)

for i in range(num_columns):
    for j in range(num_columns):

        # scatter column_j on the x-axis vs column_i on the y-axis
        if i != j: ax[i][j].scatter(get_column(data, j), get_column(data, i))

        # unless i == j, in which case show the series name
        else: ax[i][j].annotate("series " + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")

        # then hide axis labels except left and bottom charts
        if i < num_columns - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

# fix the bottom right and top left axis labels, which are wrong because
# their charts only have text in them
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())

plt.show()
```

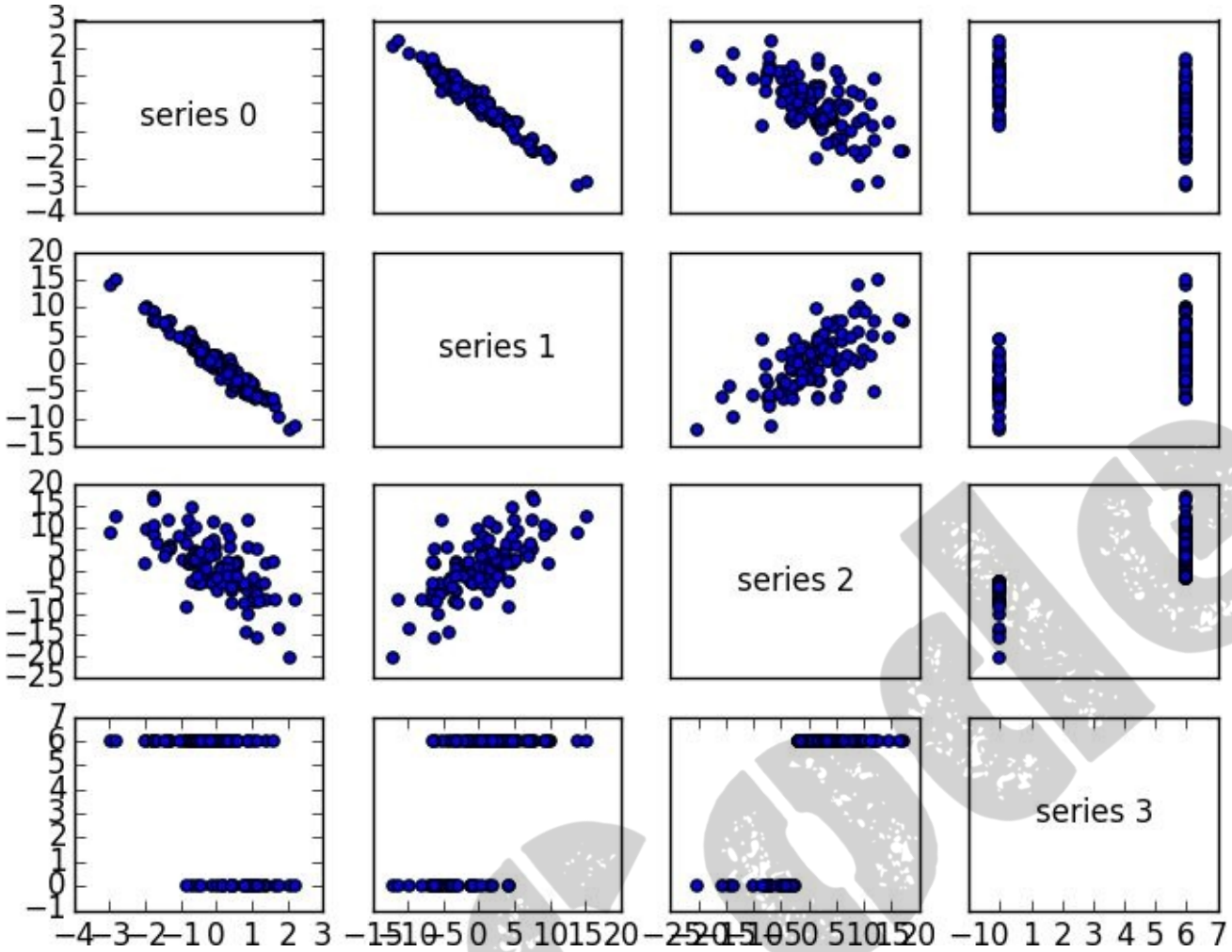

Figure 10-4. Scatterplot matrix

Looking at the scatterplots, you can see that series 1 is very negatively correlated with series 0, series 2 is positively correlated with series 1, and series 3 only takes on the values 0 and 6, with 0 corresponding to small values of series 2 and 6 corresponding to large values.

This is a quick way to get a rough sense of which of your variables are correlated (unless you spend hours tweaking `matplotlib` to display things exactly the way you want them to, in which case it's not a quick way).

Cleaning and Munging

Real-world data is *dirty*. Often you'll have to do some work on it before you can use it. We've seen examples of this in Chapter 9. We have to convert strings to `floats` or `ints` before we can use them. Previously, we did that right before using the data:

```
        closing_price = float(row[2])
```

But it's probably less error-prone to do the parsing on the way in, which we can do by creating a function that wraps `csv.reader`. We'll give it a list of parsers, each specifying how to parse one of the columns. And we'll use `None` to represent "don't do anything to this column":

```
def parse_row(input_row, parsers):
    """given a list of parsers (some of which may be None)
    apply the appropriate one to each element of the input_row"""

    return [parser(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]

def parse_rows_with(reader, parsers):
    """wrap a reader to apply the parsers to each of its rows"""
    for row in reader:
        yield parse_row(row, parsers)
```

What if there's bad data? A "float" value that doesn't actually represent a number? We'd usually rather get a `None` than crash our program. We can do this with a helper function:

```
def try_or_none(f):
    """wraps f to return None if f raises an exception
    assumes f takes only one input"""
    def f_or_none(x):
        try: return f(x)
        except: return None
    return f_or_none
```

after which we can rewrite `parse_row` to use it:

```
def parse_row(input_row, parsers):
    return [try_or_none(parser)(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]
```

For example, if we have comma-delimited stock prices with bad data:

```
6/20/2014,AAPL,90.91
6/20/2014,MSFT,41.68
6/20/3014,FB,64.5
6/19/2014,AAPL,91.86
6/19/2014,MSFT,n/a
6/19/2014,FB,64.34
```

we can now read and parse in a single step:

```
import dateutil.parser
data = []
```

```
with open("comma_delimited_stock_prices.csv", "rb") as f:
    reader = csv.reader(f)
    for line in parse_rows_with(reader, [dateutil.parser.parse, None, float]):
        data.append(line)
```

after which we just need to check for `None` rows:

```
for row in data:
    if any(x is None for x in row):
        print row
```

and decide what we want to do about them. (Generally speaking, the three options are to get rid of them, to go back to the source and try to fix the bad/missing data, or to do nothing and cross our fingers.)

We could create similar helpers for `csv.DictReader`. In that case, you'd probably want to supply a `dict` of parsers by field name. For example:

```
def try_parse_field(field_name, value, parser_dict):
    """try to parse value using the appropriate function from parser_dict"""
    parser = parser_dict.get(field_name)   # None if no such entry
    if parser is not None:
        return try_or_none(parser)(value)
    else:
        return value

def parse_dict(input_dict, parser_dict):
    return { field_name : try_parse_field(field_name, value, parser_dict)
             for field_name, value in input_dict.iteritems() }
```

A good next step is to check for outliers, using techniques from "Exploring Your Data" or by ad hoc investigating. For example, did you notice that one of the dates in the stocks file had the year 3014? That won't (necessarily) give you an error, but it's quite plainly wrong, and you'll get screwy results if you don't catch it. Real-world data sets have missing decimal points, extra zeroes, typographical errors, and countless other problems that it's your job to catch. (Maybe it's not officially your job, but who else is going to do it?)

Manipulating Data

One of the most important skills of a data scientist is *manipulating data*. It's more of a general approach than a specific technique, so we'll just work through a handful of examples to give you the flavor of it.

Imagine we're working with `dicts` of stock prices that look like:

```
data = [
    {'closing_price': 102.06,
     'date': datetime.datetime(2014, 8, 29, 0, 0),
     'symbol': 'AAPL'},
    # ...
]
```

Conceptually we'll think of them as rows (as in a spreadsheet).

Let's start asking questions about this data. Along the way we'll try to notice patterns in what we're doing and abstract out some tools to make the manipulation easier.

For instance, suppose we want to know the highest-ever closing price for AAPL. Let's break this down into concrete steps:

1. Restrict ourselves to AAPL rows.
2. Grab the `closing_price` from each row.
3. Take the `max` of those prices.

We can do all three at once using a list comprehension:

```
max_aapl_price = max(row["closing_price"]
                     for row in data
                     if row["symbol"] == "AAPL")
```

More generally, we might want to know the highest-ever closing price for each stock in our data set. One way to do this is:

1. Group together all the rows with the same `symbol`.
2. Within each group, do the same as before:

```
# group rows by symbol
by_symbol = defaultdict(list)
for row in data:
    by_symbol[row["symbol"]].append(row)

# use a dict comprehension to find the max for each symbol
max_price_by_symbol = { symbol : max(row["closing_price"]
                                     for row in grouped_rows)
                        for symbol, grouped_rows in by_symbol.iteritems() }
```

There are some patterns here already. In both examples, we needed to pull the `closing_price` value out of every `dict`. So let's create a function to pick a field out of a `dict`, and another function to pluck the same field out of a collection of `dicts`:

```
def picker(field_name):
    """returns a function that picks a field out of a dict"""
    return lambda row: row[field_name]

def pluck(field_name, rows):
    """turn a list of dicts into the list of field_name values"""
    return map(picker(field_name), rows)
```

We can also create a function to group rows by the result of a `grouper` function and to optionally apply some sort of `value_transform` to each group:

```
def group_by(grouper, rows, value_transform=None):
    # key is output of grouper, value is list of rows
    grouped = defaultdict(list)
    for row in rows:
        grouped[grouper(row)].append(row)

    if value_transform is None:
        return grouped
    else:
        return { key : value_transform(rows)
                 for key, rows in grouped.iteritems() }
```

This allows us to rewrite our previous examples quite simply. For example:

```
max_price_by_symbol = group_by(picker("symbol"),
                               data,
                               lambda rows: max(pluck("closing_price", rows)))
```

We can now start to ask more complicated things, like what are the largest and smallest one-day percent changes in our data set. The percent change is `price_today / price_yesterday - 1`, which means we need some way of associating today's price and yesterday's price. One approach is to group the prices by symbol, then, within each group:

1. Order the prices by date.
2. Use `zip` to get pairs (previous, current).
3. Turn the pairs into new "percent change" rows.

We'll start by writing a function to do all the within-each-group work:

```
def percent_price_change(yesterday, today):
    return today["closing_price"] / yesterday["closing_price"] - 1

def day_over_day_changes(grouped_rows):
    # sort the rows by date
    ordered = sorted(grouped_rows, key=picker("date"))

    # zip with an offset to get pairs of consecutive days
    return [{ "symbol" : today["symbol"],
              "date" : today["date"],
              "change" : percent_price_change(yesterday, today) }
            for yesterday, today in zip(ordered, ordered[1:])]
```

Then we can just use this as the `value_transform` in a `group_by`:

```
# key is symbol, value is list of "change" dicts
changes_by_symbol = group_by(picker("symbol"), data, day_over_day_changes)

# collect all "change" dicts into one big list
all_changes = [change
               for changes in changes_by_symbol.values()
               for change in changes]
```

At which point it's easy to find the largest and smallest:

```
max(all_changes, key=picker("change"))
# {'change': 0.3283582089552237,
#  'date': datetime.datetime(1997, 8, 6, 0, 0),
#  'symbol': 'AAPL'}
# see, e.g. http://news.cnet.com/2100-1001-202143.html

min(all_changes, key=picker("change"))
# {'change': -0.5193370165745856,
#  'date': datetime.datetime(2000, 9, 29, 0, 0),
#  'symbol': 'AAPL'}
# see, e.g. http://money.cnn.com/2000/09/29/markets/techwrap/
```

We can now use this new `all_changes` data set to find which month is the best to invest in tech stocks. First we group the changes by month; then we compute the overall change within each group.

Once again, we write an appropriate `value_transform` and then use `group_by`:

```
# to combine percent changes, we add 1 to each, multiply them, and subtract 1
# for instance, if we combine +10% and -20%, the overall change is
#    (1 + 10%) * (1 - 20%) - 1 = 1.1 * .8 - 1 = -12%
def combine_pct_changes(pct_change1, pct_change2):
    return (1 + pct_change1) * (1 + pct_change2) - 1

def overall_change(changes):
    return reduce(combine_pct_changes, pluck("change", changes))

overall_change_by_month = group_by(lambda row: row['date'].month,
                                   all_changes,
                                   overall_change)
```

We'll be doing these sorts of manipulations throughout the book, usually without calling too much explicit attention to them.

Rescaling

Many techniques are sensitive to the *scale* of your data. For example, imagine that you have a data set consisting of the heights and weights of hundreds of data scientists, and that you are trying to identify *clusters* of body sizes.

Intuitively, we'd like clusters to represent points near each other, which means that we need some notion of distance between points. We already have a Euclidean `distance` function, so a natural approach might be to treat (height, weight) pairs as points in two-dimensional space. Consider the people listed in Table 10-1.

Table 10-1. Heights and Weights

Person	Height (inches)	Height (centimeters)	Weight
A	63 inches	160 cm	150 pounds
B	67 inches	170.2 cm	160 pounds
C	70 inches	177.8 cm	171 pounds

If we measure height in inches, then B's nearest neighbor is A:

```
a_to_b = distance([63, 150], [67, 160])        # 10.77
a_to_c = distance([63, 150], [70, 171])        # 22.14
b_to_c = distance([67, 160], [70, 171])        # 11.40
```

However, if we measure height in centimeters, then B's nearest neighbor is instead C:

```
a_to_b = distance([160, 150], [170.2, 160])    # 14.28
a_to_c = distance([160, 150], [177.8, 171])    # 27.53
b_to_c = distance([170.2, 160], [177.8, 171])  # 13.37
```

Obviously it's problematic if changing units can change results like this. For this reason, when dimensions aren't comparable with one another, we will sometimes *rescale* our data so that each dimension has mean 0 and standard deviation 1. This effectively gets rid of the units, converting each dimension to "standard deviations from the mean."

To start with, we'll need to compute the `mean` and the `standard_deviation` for each column:

```
def scale(data_matrix):
    """returns the means and standard deviations of each column"""
    num_rows, num_cols = shape(data_matrix)
    means = [mean(get_column(data_matrix,j))
             for j in range(num_cols)]
    stdevs = [standard_deviation(get_column(data_matrix,j))
              for j in range(num_cols)]
    return means, stdevs
```

And then use them to create a new data matrix:

```
def rescale(data_matrix):
    """rescales the input data so that each column
    has mean 0 and standard deviation 1
    leaves alone columns with no deviation"""
    means, stdevs = scale(data_matrix)

    def rescaled(i, j):
        if stdevs[j] > 0:
            return (data_matrix[i][j] - means[j]) / stdevs[j]
        else:
            return data_matrix[i][j]

    num_rows, num_cols = shape(data_matrix)
    return make_matrix(num_rows, num_cols, rescaled)
```

As always, you need to use your judgment. If you were to take a huge data set of heights and weights and filter it down to only the people with heights between 69.5 inches and 70.5 inches, it's quite likely (depending on the question you're trying to answer) that the variation remaining is simply *noise*, and you might not want to put its standard deviation on equal footing with other dimensions' deviations.

YuCode

Dimensionality Reduction

Sometimes the “actual” (or useful) dimensions of the data might not correspond to the dimensions we have. For example, consider the data set pictured in Figure 10-5.

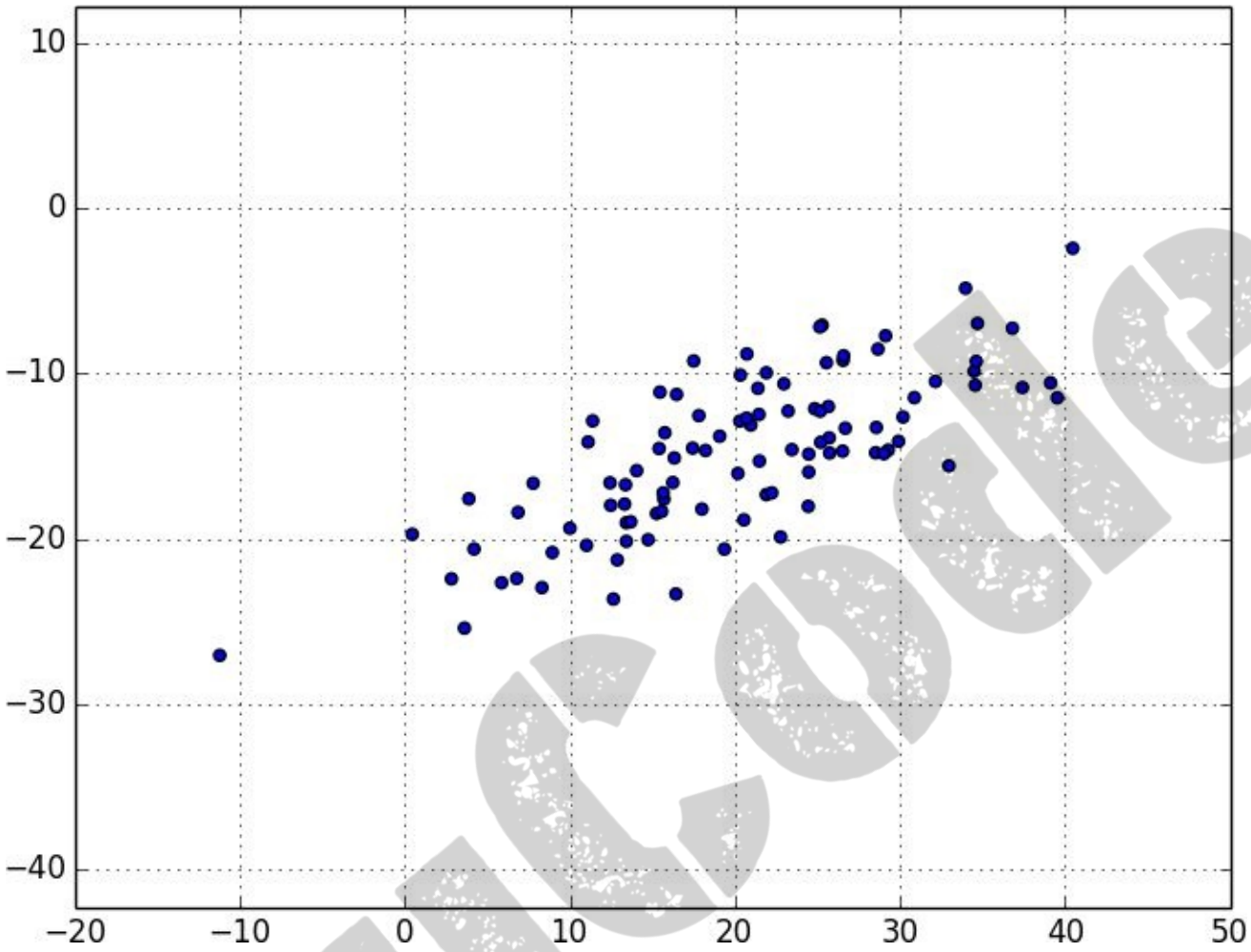

Figure 10-5. Data with the “wrong” axes

Most of the variation in the data seems to be along a single dimension that doesn’t correspond to either the x-axis or the y-axis.

When this is the case, we can use a technique called *principal component analysis* to extract one or more dimensions that capture as much of the variation in the data as possible.

> **NOTE**
>
> In practice, you wouldn’t use this technique on such a low-dimensional data set. Dimensionality reduction is mostly useful when your data set has a large number of dimensions and you want to find a small subset that captures most of the variation. Unfortunately, that case is difficult to illustrate in a two-dimensional book format.

As a first step, we’ll need to translate the data so that each dimension has mean zero:

```
def de_mean_matrix(A):
    """returns the result of subtracting from every value in A the mean
    value of its column. the resulting matrix has mean 0 in every column"""
    nr, nc = shape(A)
    column_means, _ = scale(A)
    return make_matrix(nr, nc, lambda i, j: A[i][j] - column_means[j])
```

(If we don't do this, our techniques are likely to identify the mean itself rather than the variation in the data.)

Figure 10-6 shows the example data after de-meaning.

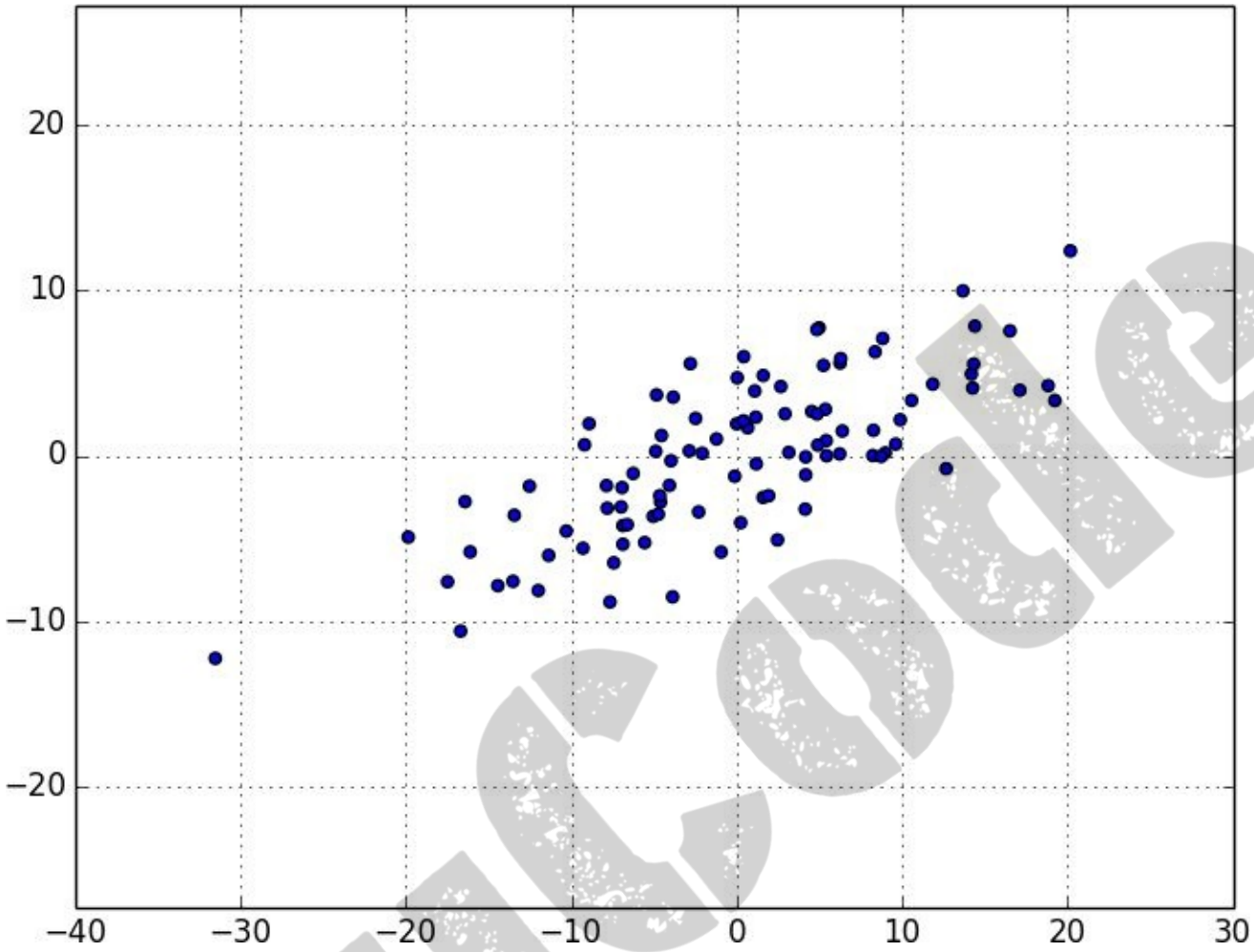

Figure 10-6. Data after de-meaning

Now, given a de-meaned matrix *X*, we can ask which is the direction that captures the greatest variance in the data?

Specifically, given a direction *d* (a vector of magnitude 1), each row *x* in the matrix extends `dot(x, d)` in the *d* direction. And every nonzero vector *w* determines a direction if we rescale it to have magnitude 1:

```
def direction(w):
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

Therefore, given a nonzero vector *w*, we can compute the variance of our data set in the direction determined by *w*:

```
def directional_variance_i(x_i, w):
    """the variance of the row x_i in the direction determined by w"""
    return dot(x_i, direction(w)) ** 2

def directional_variance(X, w):
    """the variance of the data in the direction determined w"""
    return sum(directional_variance_i(x_i, w)
               for x_i in X)
```

We'd like to find the direction that maximizes this variance. We can do this using gradient descent, as soon as we have the gradient function:

```
def directional_variance_gradient_i(x_i, w):
    """the contribution of row x_i to the gradient of
    the direction-w variance"""
    projection_length = dot(x_i, direction(w))
    return [2 * projection_length * x_ij for x_ij in x_i]

def directional_variance_gradient(X, w):
    return vector_sum(directional_variance_gradient_i(x_i,w)
                      for x_i in X)
```

The first principal component is just the direction that maximizes the `directional_variance` function:

```
def first_principal_component(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_batch(
        partial(directional_variance, X),           # is now a function of w
        partial(directional_variance_gradient, X),  # is now a function of w
        guess)
    return direction(unscaled_maximizer)
```

Or, if you'd rather use stochastic gradient descent:

```
# here there is no "y", so we just pass in a vector of Nones
# and functions that ignore that input
def first_principal_component_sgd(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_stochastic(
        lambda x, _, w: directional_variance_i(x, w),
        lambda x, _, w: directional_variance_gradient_i(x, w),
        X,
        [None for _ in X],   # the fake "y"
        guess)
    return direction(unscaled_maximizer)
```

On the de-meaned data set, this returns the direction `[0.924, 0.383]`, which does appear to capture the primary axis along which our data varies (Figure 10-7).

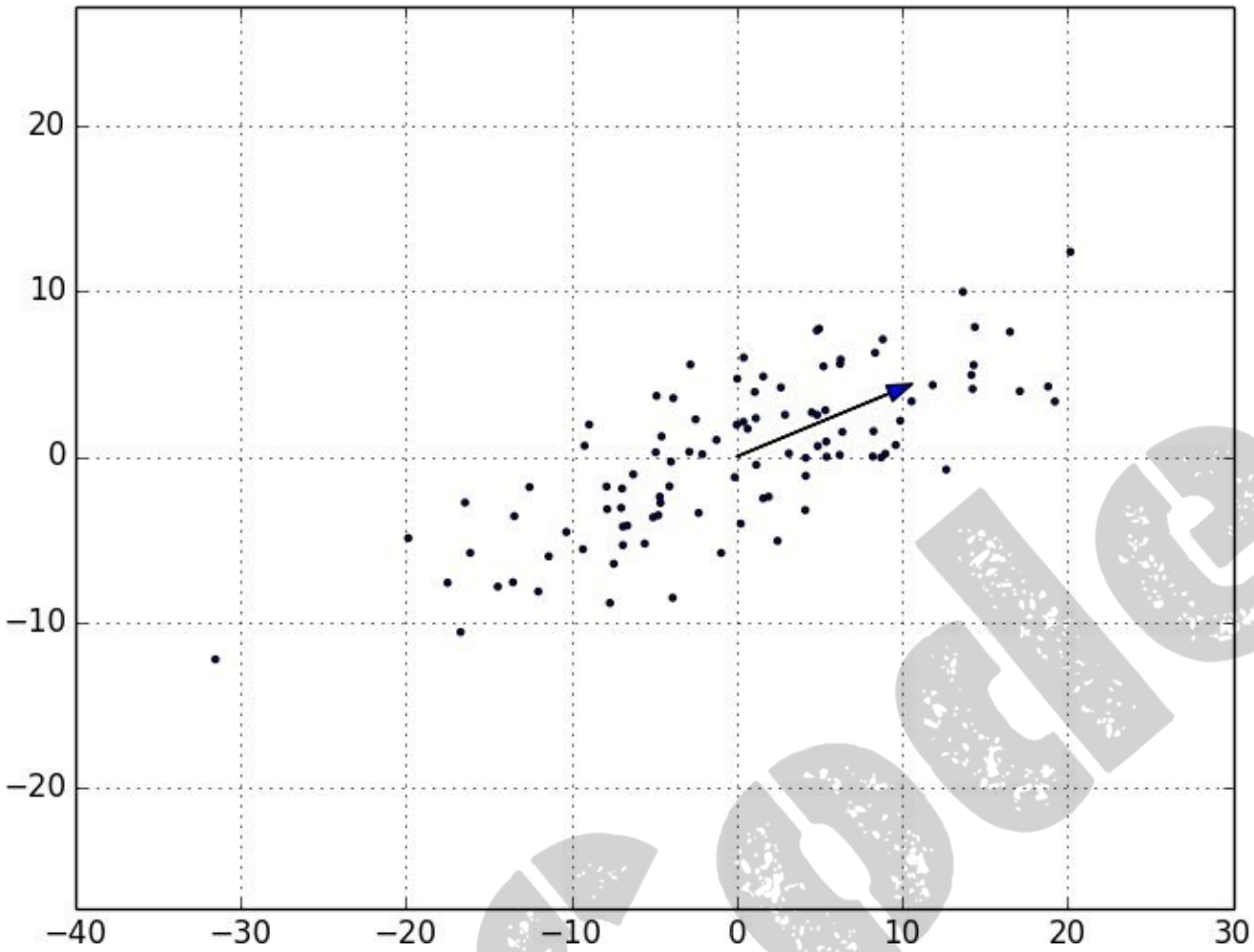

Figure 10-7. First principal component

Once we've found the direction that's the first principal component, we can project our data onto it to find the values of that component:

```
def project(v, w):
    """return the projection of v onto the direction w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

If we want to find further components, we first remove the projections from the data:

```
def remove_projection_from_vector(v, w):
    """projects v onto w and subtracts the result from v"""
    return vector_subtract(v, project(v, w))

def remove_projection(X, w):
    """for each row of X
    projects the row onto w, and subtracts the result from the row"""
    return [remove_projection_from_vector(x_i, w) for x_i in X]
```

Because this example data set is only two-dimensional, after we remove the first component, what's left will be effectively one-dimensional (Figure 10-8).

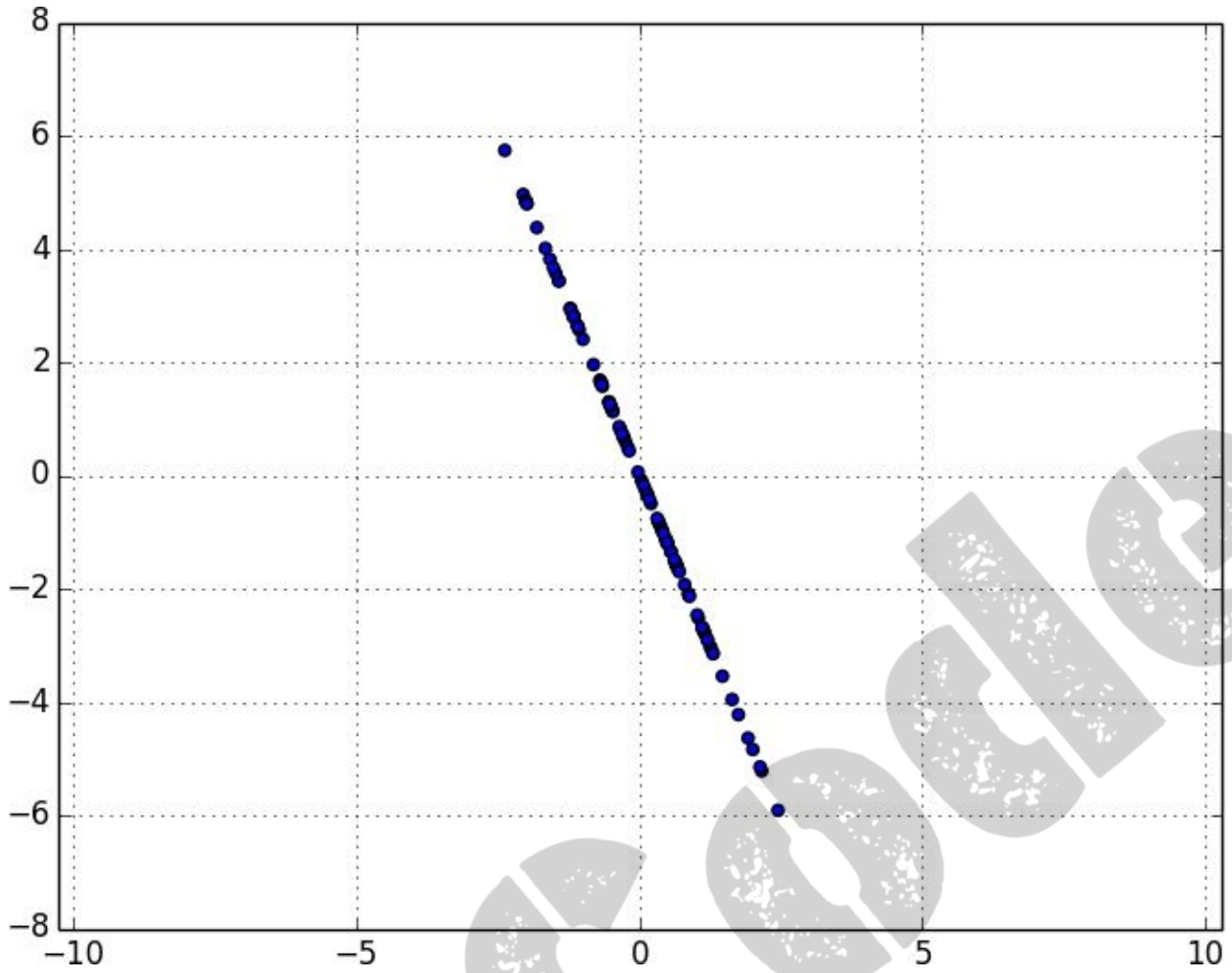

Figure 10-8. Data after removing first principal component

At that point, we can find the next principal component by repeating the process on the result of `remove_projection` (Figure 10-9).

On a higher-dimensional data set, we can iteratively find as many components as we want:

```
def principal_component_analysis(X, num_components):
    components = []
    for _ in range(num_components):
        component = first_principal_component(X)
        components.append(component)
        X = remove_projection(X, component)

    return components
```

We can then *transform* our data into the lower-dimensional space spanned by the components:

```
def transform_vector(v, components):
    return [dot(v, w) for w in components]

def transform(X, components):
    return [transform_vector(x_i, components) for x_i in X]
```

This technique is valuable for a couple of reasons. First, it can help us clean our data by eliminating noise dimensions and consolidating dimensions that are highly correlated.

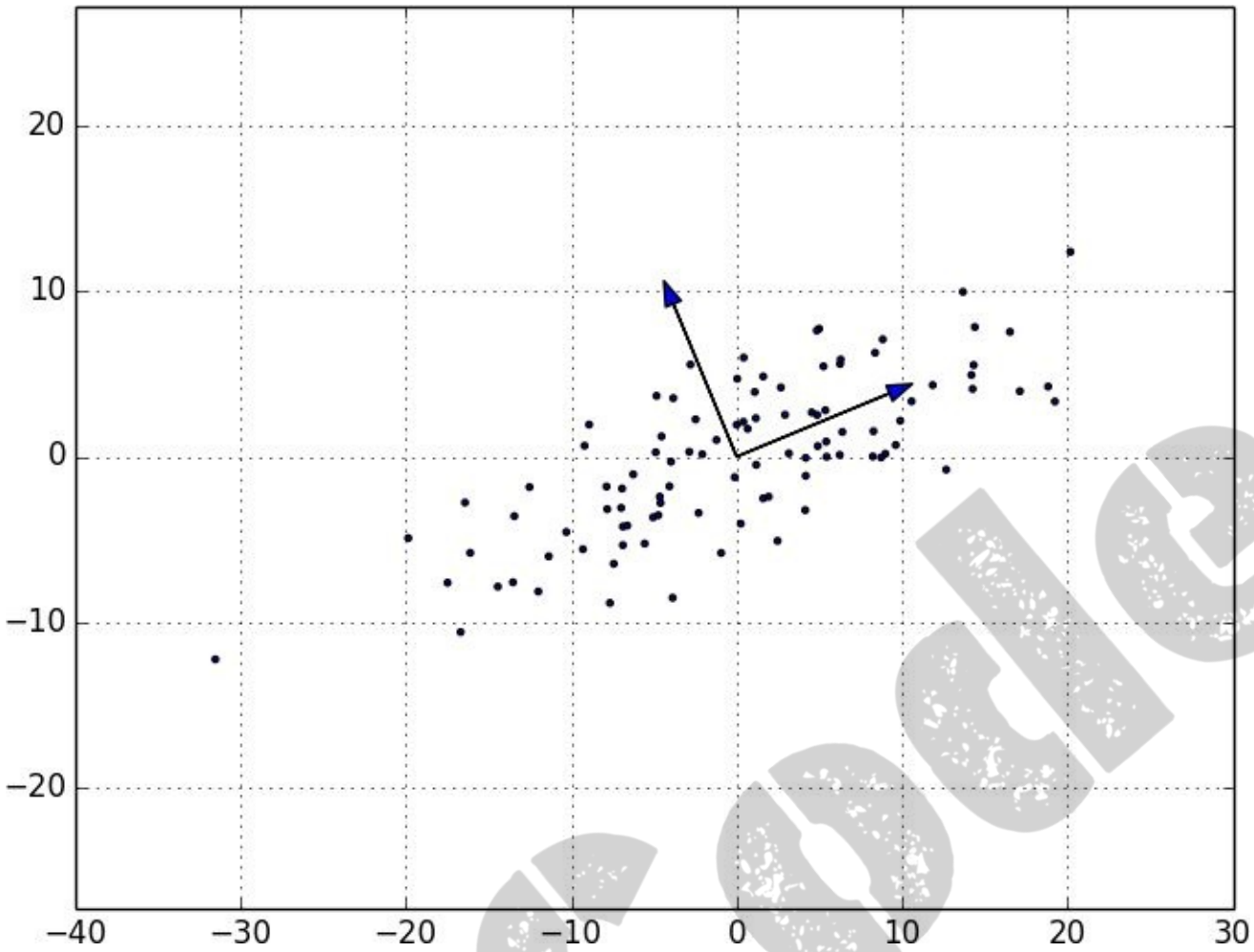

Figure 10-9. First two principal components

Second, after extracting a low-dimensional representation of our data, we can use a variety of techniques that don't work as well on high-dimensional data. We'll see examples of such techniques throughout the book.

At the same time, while it can help you build better models, it can also make those models harder to interpret. It's easy to understand conclusions like "every extra year of experience adds an average of $10k in salary." It's much harder to make sense of "every increase of 0.1 in the third principal component adds an average of $10k in salary."

VeriCode