Think Bayes

Bayesian Statistics Made Simple

Version 0.14.0

Think Bayes

Bayesian Statistics Made Simple

Version 0.14.0

Allen B. Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

Preface

This version of the book is a rough draft. I am making this draft available for comments, but it comes with the warning that it is probably full of errors.

If you find some of those errors, please let me know. But it is probably too early to bother with typos.

My theory, which is mine

The premise of this book is that if you know how to program, you can use that skill to help you learn other topics, including Bayesian statistics.

Most books on Bayesian statistics use mathematical notation and present ideas in terms of mathematical concepts like calculus. This book uses Python code instead of math, and discrete approximations instead of continuous mathematics. As a result, what would be an integral in a math book becomes a simple summation, and most operations on distributions are simple loops.

This presentation is easier to understand, at least for people with programming skills. It is also more general, because when we make modeling decisions, we can choose the most appropriate model without excessive concern about whether the model lends itself to conventional analysis.

And while it's true that the discrete computations I present here yield approximations of the results we would get from continuous functions, it is important to remember that both computations are based on models of a real system, and that any approximation errors (differences between the models) are almost always negligible compared to modeling errors (differences between the models and reality).

If, as is often the case, the discrete approach allows us to make better modeling decisions, then it would be more fair to say that the continuous functions yield an approximation of the more reliable computational results.

However, I have to acknowledge that continuous analysis often yields substantial performance advantages, for example by replacing a linear- or quadratic-time computation with a constant-time solution.

Therefore, I recommend a general process with these steps:

1. While you are exploring a problem, start with simple models and implement them in code that is clear, readable, and demonstrably correct. Focus your attention on good modeling decisions, not optimization.

2. If the performance of your solution is good enough for your application, stop! Or if you insist, review and refine your modeling decisions.

3. If you really feel the need for speed, there are two approaches to consider. You can review your code and look for optimizations; for example, caching previously computed results to avoid redundant computation. Or you can look for mathematical short-cuts.

One benefit of this process is that Step 1 tends to be fast, so you can explore alternative models before investing heavily in any of them.

Another benefit is that if you get to Step 3, you will be starting with a reference implementation that is likely to be correct, so you can use for regression testing (that is, checking that the optimized code yields the same answer, at least approximately).

Code style

Experienced Python programmers will notice that the code in this book does not comply with PEP 8, which is the most common style guide for Python (`http://www.python.org/dev/peps/pep-0008/`).

In particular, PEP 8 calls for lowercase function names with underscores between words, `like_this`. In this book and the accompanying code, function and method names begin with a capital letter and use camel case, `LikeThis`.

The reason I broke this rule is that I developed some of the code while I was a Visiting Scientist at Google, Inc. in 2009-10. So I followed the Google style guide for Python, which deviates from PEP 8 in a few places.

Also, once I got used to function names with capital letters, I found that I liked it. Finally, at this point, it would be too much trouble to change.

Allen B. Downey
Needham MA

Allen B. Downey is a Professor of Computer Science at the Franklin W. Olin College of Engineering.

Contributor List

If you have a suggestion or correction, please send email to `downey@allendowney.com`. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with. Thanks!

- First, I have to acknowledge David MacKay's excellent book, *Information Theory, Inference, and Learning Algorithms* which is where I first came to understand Bayesian methods. With his permission (and with acknowledgment) I use several problems from his book as examples.
- This book has also benefited from my interactions with Sanjoy Mahajan, especially in Fall 2012, when I audited his class on Bayesian Inference at Olin College.
- I wrote parts of this book during project nights with the Boston Python Users' Group, so I would like to thank them for their company, and pizza.
- Jonathan Edwards sent in the first typo.
- George Purkins found a markup error.
- Olivier Yiptong send several helpful suggestions.
- Yuriy Pasichnyk found several errors.

Contents

Chapter 1

Bayes's Theorem

1.1 Conditional probability

The fundamental idea behind all Bayesian statistics is Bayes's Theorem, which is surprisingly easy to derive, provided that you understand conditional probability. So we'll start with probability, then conditional probability, then Bayes's Theorem, and on to Bayesian statistics.

A probability is a number between 0 and 1 (including both) that represents a degree of belief in a fact or prediction. A probability of 1 represents certainty that a fact is true, or that a prediction will come true. A probability of 0 represents equal certainty that the fact is false.

Intermediate values represent degrees of certainty. The value 0.5, often represented as 50%, means that a predicted outcome is as likely to happen as not. For example, the probability that a tossed coin lands face up is very close to 50%.

A conditional probability is a probability based on some background information. For example, I might be interested in the probability that I will have a heart attack in the next year. According to the CDC, "Every year about 785,000 Americans have a first coronary attack. (`http://www.cdc.gov/heartdisease/facts.htm`)"

The U.S. population is about 311 million, so the probability that a randomly-chosen American will have a heart attack in the next year is roughly 0.3%.

But I am not a randomly-chosen American. Epidemiologists have identified many factors affect the risk of heart attacks; depending on those factors, my risk might be higher or lower than average.

I am male, 45 years old, and I have borderline high cholesterol. Those factors increase my chances. However, I have low blood pressure and I don't smoke, and those factors decrease my chances.

Plugging everything into the online calculator at `http://hp2010.nhlbihin.net/atpiii/calculator.asp`, I find that my risk of a heart attack in the next year is about 0.2%, slightly less than the national average. That value is a conditional probability, because it is based on a number of factors that make up my "condition."

The usual notation for conditional probability is $\mathrm{p}(A|B)$, which is the probability of A given that B is true. In this example, A represents the prediction that I will have a heart attack in the next year, and B is the set of conditions I listed.

1.2 Conjoint probability

"Conjoint probability" is a fancy way to say the probability that two things are true. I will write $\mathrm{p}(A \text{ and } B)$ to mean the probability that A and B are both true.

If you learned about probability in the context of coin tosses and dice, you might have learned the formula

$$\mathrm{p}(A \text{ and } B) = \mathrm{p}(A)\,\mathrm{p}(B) \quad \text{WARNING: not always true}$$

For example, if I toss two coins, and A means the first coin lands face up, and B means the second coin lands face up, then $\mathrm{p}(A) = \mathrm{p}(B) = 0.5$, and sure enough, $\mathrm{p}(A \text{ and } B) = \mathrm{p}(A)\,\mathrm{p}(B) = 0.25$.

But this formula only works because in this case A and B are independent; that is, the second event does not depend on the first. If you tell me the first coin is heads, that does not change $\mathrm{p}(B)$.

Here is a different example where the events are not independent. Suppose again that A means the first coin lands face up, but let's add C, which means that *both* coins land face up. What is the probability of both events, $\mathrm{p}(A \text{ and } C)$?

These events are dependent because if we know whether or not A is true, that changes $\mathrm{p}(C)$. Specifically, if A is true then $\mathrm{p}(C) = 0.5$; but if A is false $\mathrm{p}(C) = 0$.

To be more correct, I should not say that $p(C)$ changes, but rather that the conditional probabilities are different: $p(C|A) = 0.5$ and $p(C|\text{not } A) = 0$.

In general, the probability of a conjunction is

$$p(A \text{ and } C) = p(A)\, p(C|A)$$

for any A and C. In this example, $p(A \text{ and } C)$ = (0.5)(0.5) = 0.25.

1.3 The Cookie Problem

We'll get to Bayes's Theorem soon, but I want to motivate it with an example called The Cookie Problem[1]. Suppose there are two bowls of cookies. Bowl 1 contains 30 vanilla cookies and 10 chocolate cookies. Bowl 2 contains 20 of each.

Now suppose you choose one of the bowls at random and, without looking, select a cookie at random. The cookie is vanilla. What is the probability that it came from Bowl 1?

This is a conditional probability; we want $p(\text{Bowl 1}|\text{vanilla})$, but it is not obvious how to compute it. On the other hand, if we flip it around, it's very easy: $p(\text{vanilla}|\text{Bowl 1}) = 3/4$ Unfortunately, $p(A|B)$ is *not* the same as $p(B|A)$. But there is a way to get from one to the other: Bayes's Theorem.

1.4 Bayes's Theorem

At this point we have everything we need to derive Bayes's Theorem. We'll start with the observation that conjunction is commutative; that is

$$p(A \text{ and } B) = p(B \text{ and } A)$$

for any events A and B.

Next, we write the probability of a conjunction:

$$p(A \text{ and } B) = p(A)\, p(B|A)$$

[1] Based on an example from `http://en.wikipedia.org/wiki/Bayes'_theorem` that is no longer there.

Since we have not said anything about what A and B mean, they are interchangeable. Interchanging them yields

$$\mathrm{p}(B \text{ and } A) = \mathrm{p}(B)\ \mathrm{p}(A|B)$$

And that's all we need. Pulling those pieces together, we get

$$\mathrm{p}(B)\ \mathrm{p}(A|B) = \mathrm{p}(A)\ \mathrm{p}(B|A)$$

Which means there are two ways to compute the conjunction. If you have $\mathrm{p}(A)$, you multiply by the conditional probability $\mathrm{p}(B|A)$. Or you can do it the other way around; if you know $\mathrm{p}(B)$, you multiply by $\mathrm{p}(A|B)$. Either way you should get the same thing.

Finally we can divide through by $\mathrm{p}(B)$

$$\mathrm{p}(A|B) = \frac{\mathrm{p}(A)\ \mathrm{p}(B|A)}{\mathrm{p}(B)}$$

And that's Bayes's Theorem! It doesn't look like much, but it is a surprisingly powerful idea.

For example, we can use it to solve the Cookie Problem. I'll write B_1 for the hypothesis that the cookie came from Bowl 1 and V for the vanilla cookie. Plugging in Bayes's Theorem we get

$$\mathrm{p}(B_1|V) = \frac{\mathrm{p}(B_1)\ \mathrm{p}(V|B_1)}{\mathrm{p}(V)}$$

The term on the left is what we want: the probability of Bowl 1, given that we chose a vanilla cookie. The terms on the right are:

- $\mathrm{p}(B_1)$: This is the probability that we chose Bowl 1, unconditioned by what kind of cookie we got. Since the problem says we chose a bowl at random, we can assume $\mathrm{p}(B_1) = 1/2$.

- $\mathrm{p}(V|B_1)$: This is the probability of getting a vanilla cookie from Bowl 1, which is $3/4$.

- $\mathrm{p}(V)$: This is the probability of drawing a vanilla cookie from either bowl. If we combine the two bowls, we get 50 vanilla and 30 chocolate cookies, so $\mathrm{p}(V) = 5/8$.

Putting it together, we have

$$\mathrm{p}(B_1|V) = \frac{(1/2)(3/4)}{5/8}$$

which reduces to 3/5. So the vanilla cookie is evidence in favor of the hypothesis that we chose Bowl 1, because vanilla cookies are more likely to come from Bowl 1.

This example demonstrates one use of Bayes's Theorem: it provides a process to get from $\mathrm{p}(B|A)$ to $\mathrm{p}(A|B)$. This strategy is useful in cases, like the Cookie Problem, where it is easier to compute the terms on the right side of Bayes's Theorem than the term on the left.

1.5 The diachronic interpretation

There is another way to think of Bayes's Theorem: it gives us a way to update the probability of a hypothesis, H, in light of some body of data, D.

This way of thinking about Bayes's Theorem is called the **diachronic interpretation**. "Diachronic" means that something is happening over time; in this case there is an implicit notion that the probability of the hypotheses changes, over time, as we see new data.

Rewriting Bayes's Theorem with H and D yields:

$$\mathrm{p}(H|D) = \frac{\mathrm{p}(H)\,\mathrm{p}(D|H)}{\mathrm{p}(D)}$$

In this interpretation, each term has a name:

- $\mathrm{p}(H)$ is the probability of the hypothesis before we see the data, called the prior probability, or just **prior**.
- $\mathrm{p}(H|D)$ is what we want to compute, the probability of the hypothesis after we see the data, called the **posterior**.
- $\mathrm{p}(D|H)$ is the probability of the data under the hypothesis, called the **likelihood**.
- $\mathrm{p}(D)$ is the probability of the data under any hypothesis, called the **normalizing constant**.

In some cases, we can compute the prior based on background information. For example, the Cookie Problem specifies that we choose a bowl at random (and implies that they have equal probability).

In other cases the prior is subjective; that is, reasonable people might disagree, either because they use different background information or because they interpret the same information differently.

The likelihood is usually the easiest part to compute. In the Cookie Problem, if we know which bowl the cookie came from, we can get the probability of getting a vanilla cookie by counting.

The normalizing constant can be tricky. It is supposed to be the probability of seeing the data under any hypothesis at all, but in the most general case it is hard to see what that means.

Most often we simplify things by specifying a set of hypotheses that are

Mutually exclusive: which means that only one hypothesis in the set can be true, and

Collectively exhaustive: which means that there are no other possibilities; one of the hypotheses has to be true.

I use the word **suite** for a set of hypotheses that has these properties.

In the Cookie Problem, there are only two hypotheses—the cookie came from Bowl 1 or Bowl 2—and they are mutually exclusive and collectively exhaustive.

In that case we can compute $\mathrm{p}(D)$ using the law of total probability, which says that if there are two exclusive ways that something might happen, you can add up the probabilities like this:

$$\mathrm{p}(D) = \mathrm{p}(B_1)\ \mathrm{p}(D|B_1) + \mathrm{p}(B_2)\ \mathrm{p}(D|B_2)$$

Plugging in the values from the Cookie Problem, we have

$$\mathrm{p}(D) = (1/2)\ (3/4) + (1/2)\ (1/2) = 5/8$$

which is what we computed earlier by mentally combining the two bowls.

1.6 The M&M Problem

M&M's are small candy-coated chocolates that come in a variety of colors. Mars, Inc., which makes M&M's, changes the mixture of colors from time to time.

In 1995, they introduced blue M&M's. Before then, the color mix in a bag of plain M&Ms was (30% Brown, 20% Yellow, 20% Red, 10% Green, 10% Orange, 10% Tan). Afterward it was (24% Blue , 20% Green, 16% Orange, 14% Yellow, 13% Red, 13% Brown).

A friend of mine has two bags of M&M's, and he tells me that one is from 1994 and one from 1996. He won't tell me which is which, but he gives me one M&M from each bag. One is yellow and one is green. What is the probability that the yellow one came from the 1994 bag?

This problem is similar to the Cookie Problem, with the twist that I draw one sample from each bowl/bag. This problem also gives me a chance to demonstrate the table method, which is useful for solving problems like this on paper. In the next chapter we will solve them computationally.

The first step is to enumerate the hypotheses. In this case there are only two:

- A: the yellow M&M is from 1994, which implies that green is from 1996.
- B: the yellow M&M is from 1996 and green from 1994.

Now we construct a table with a row for each hypothesis and a column for each term in Bayes's Theorem:

	$\mathrm{p}(H)$	$\mathrm{p}(D\|H)$	$\mathrm{p}(H)\,\mathrm{p}(D\|H)$	$\mathrm{p}(H\|D)$
A	1/2	(20)(20)	200	20/27
B	1/2	(10)(14)	70	7/27

The first column has the priors. Since there is one bag from 1994 and one from 1996, it is reasonable to choose $\mathrm{p}(A) = \mathrm{p}(B) = 1/2$.

The second column has the likelihoods, which follow from the information in the problem. If A is true, the yellow M&M came from 1994 with probability 20%, and the green came from 1996 with probability 20%. Because the selections are independent, we get the conjoint probability by multiplying.

The third column is just the product of the previous two. The sum of this column, 270, is the normalizing constant. To get the last column, which

contains the posteriors, we divide the third column by the normalizing constant.

That's it. Simple, right?

Well, you might be bothered by one detail. I wrote $p(D|H)$ in terms of percentages, not probabilities, which means it is off by a factor of 10,000. But that cancels out when we divide through by the normalizing factor, so it doesn't affect the result.

When the set of hypotheses is mutually exclusive and collectively exhaustive, you can multiply the likelihoods by any factor, if it is convenient, as long as you apply the same factor to the entire column.

1.7 The Monty Hall problem

The Monty Hall problem might be the most contentious question in the history of probability. The scenario is simple, but the correct answer is so counter-intuitive that many people just can't accept it, and many smart people have embarrassed themselves not just by getting it wrong but by arguing the wrong side, aggressively, in public.

Monty Hall was the original host of the game show *Let's Make a Deal*. The Monty Hall problem is based on one of the regular games on the show. If you are on the show, here's what happens:

- Monty shows you three closed doors and tells you that there is a prize behind each door: one prize is a car, the other two are less valuable prizes like peanut butter and fake finger nails. The prizes are arranged at random.
- The object of the game is to guess which door has the car. If you guess right, you get to keep the car.
- So you pick a door, which we will call Door A. We'll call the other doors B and C.
- Before opening the door you chose, Monty increases the suspense by opening either Door B or C, whichever does not have the car. (If the car is actually behind Door A, Monty can safely open B or C, so he chooses one at random).
- Then Monty offers you the option to stick with your original choice or switch to the one remaining unopened door.

The question is, should you "stick" or "switch" or does it make no difference?

Most people have the strong intuition that it makes no difference. There are two doors left, they reason, so the chance that the car is behind Door A is 50%.

But that is wrong. In fact, the chance of winning if you stick with Door A is only 1/3; if you switch, your chances are 2/3.

By applying Bayes's Theorem, we can break this problem into simple pieces, and maybe convince ourselves that the correct answer is, in fact, correct.

To start, we should make a careful statement of the data. In this case D consists of two parts: Monty chooses Door B, and there is no car there.

Next we define three hypotheses: A, B and C represent the hypothesis that the car is behind Door A, Door B or Door C. Again, let's apply the table method:

	$p(H)$	$p(D\|H)$	$p(H)\,p(D\|H)$	$p(H\|D)$
A	1/3	1/2	1/6	1/3
B	1/3	0	0	0
C	1/3	1	1/3	2/3

Filling in the priors is easy because we are told that the prizes are arranged at random, which implies that the car is equally likely to be behind any door.

Figuring out the likelihoods takes some thought, but with reasonable care we can be confident that we have it right:

- If the car is actually behind A, Monty could safely open Doors B or C. So the probability that he chooses B is 1/2. And since the car is actually behind A, the probability that the car is not behind B is 1.
- If the car is actually behind B, Monty does not open B, so the probability of the data we saw is 0.
- Finally, if the car is behind Door C, Monty opens B with probability 1 and finds no car there with probability 1.

Now the hard part is over; the rest is just arithmetic. The sum of the third column is 1/2. Dividing through yields $p(A|D) = 1/3$ and $p(C|D) = 2/3$. So you are better off switching.

There are many variations of the Monty Hall problem. One of the strengths of the Bayesian approach is that it generalizes to handle these variations.

For example, suppose that Monty always chooses B if he can, and only chooses C if he has to (because the car is behind B). In that case the revised table is:

	$\mathrm{p}(H)$	$\mathrm{p}(D\|H)$	$\mathrm{p}(H)\,\mathrm{p}(D\|H)$	$\mathrm{p}(H\|D)$
A	1/3	1	1/3	1/2
B	1/3	0	0	0
C	1/3	1	1/3	1/2

The only change is $\mathrm{p}(D|A)$. If the car is behind A, Monty can chose to open B or C. But in this variation he always chooses B, so $\mathrm{p}(D|A) = 1$.

As a result, the likelihoods are the same for A and C, and the posteriors are the same: $\mathrm{p}(A|D) = \mathrm{p}(C|D) = 1/2$. In this case, the fact that Monty chose B reveals no information about the location of the car, so it doesn't matter whether the contestant sticks or switches.

On the other hand, if he had opened C, we would know $\mathrm{p}(B|D) = 1$.

Chapter 2

Computational statistics

2.1 Distributions

In statistics a **distribution** is a set of values and their corresponding probabilities.

For example, if you roll a six-sided die, the set of possible values is the numbers 1 to 6, and the probability associated with each value is 1/6.

As another example, you might be interested in how many times each word appears in common English usage. You could build a distribution that includes each word and how many time it appears.

In Python, you could represent a distribution with a dictionary that maps from each value to its probability. As an alternative, I have written a class called `Pmf` that represents a distribution. PMF stands for "probability mass function" which is the more specific mathematical term for what I am calling a distribution.

`Pmf` is defined in a Python module I wrote to accompany this book, called `thinkbayes.py`. You can download it from `http://thinkbayes.com/thinkbayes.py`. To use `Pmf` you can import it like this:

```
from thinkbayes import Pmf
```

The following code builds a Pmf to represent the distribution of outcomes for a six-sided die:

```
pmf = Pmf()
for x in [1,2,3,4,5,6]:
    pmf.Set(x, 1/6.0)
```

`Pmf` creates an empty Pmf object with no values. The `Set` method sets the probability associated with each value to 1/6.

Here's another example that counts the number of times each word appears in a sequence:

```
pmf = Pmf()
for word in word_list:
    pmf.Incr(word, 1)
```

`Incr` increases the "probability" associated with each word by 1. If a word is not already in the Pmf, it is added.

I put "probability" in quotes because in this example, the probabilities are not normalized; that is, they do not add up to 1. So they are not true probabilities.

But in this example the word counts are proportional to the probabilities. So after we count all the words, we can compute probabilities by dividing through by the total number of words. `Pmf` provides a method, `Normalize`, that does exactly that:

```
pmf.Normalize()
```

Once you have made a Pmf object, you can ask for the probability associated with any value:

```
print pmf.Prob('the')
```

And that would print the frequency of the word "the" as a fraction of the words in the list.

Pmf uses a Python dictionary to store the values and their probabilities, so the values in the Pmf can be any hashable type. The probabilities can be any numerical type, but they are usually floating point numbers (type `float`).

2.2 The Cookie Problem

In the context of Bayes's Theorem, it is natural to use a Pmf to map from each hypothesis to its probability. In the Cookie Problem, the hypotheses are B_1 and B_2. In Python, I will represent them with strings:

```
pmf = Pmf()
pmf.Set('Bowl 1', 0.5)
pmf.Set('Bowl 2', 0.5)
```

This distribution, which contains the priors for each hypothesis, is called (wait for it) the **prior distribution**.

To update the distribution based on new data (the vanilla cookie), we multiply each prior by the corresponding likelihood. The likelihood of drawing a vanilla cookie from Bowl 1 is 3/4. The likelihood for Bowl 2 is 1/2.

```
pmf.Mult('Bowl 1', 0.75)
pmf.Mult('Bowl 2', 0.5)
```

`Mult` does what you would expect. It gets the probability for the given value and multiplies by the given factor.

After this update, the distribution is no longer normalized, but because these hypotheses are mutually exclusion and collectively exhaustive, we can **renormalize**:

```
pmf.Normalize()
```

The result is a distribution that contains the posterior probability for each hypothesis, which is called (wait now) the **posterior distribution**.

Finally, we can get the posterior for Bowl 1.

```
print pmf.Prob('Bowl 1')
```

And the answer is 0.6. You can download this example from `http://thinkbayes.com/cookie.py`.

2.3 The Bayesian framework

Before we go on to other problems, I want to rewrite the code from the previous section to make it more general. First I'll define a class to encapsulate the code related to this problem:

```
class Cookie(Pmf):
    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()
```

A Cookie object is a Pmf that maps from hypotheses to their probabilities. The `__init__` method gives each hypothesis the same probability. As in the previous section, there are two hypotheses:

```
    hypos = ['Bowl 1', 'Bowl 2']
    pmf = Cookie(hypos)
```

`Cookie` provides an `Update` method that takes data as a parameter and updates the probabilities.

```
def Update(self, data):
    for hypo in self.Values():
        like = self.Likelihood(hypo, data)
        self.Mult(hypo, like)
    self.Normalize()
```

`Update` loops through each hypothesis in the suite and multiplies its probability by the likelihood of the data under the hypothesis, which is computed by `Likelihood`:

```
mixes = {
    'Bowl 1':dict(vanilla=0.75, chocolate=0.25),
    'Bowl 2':dict(vanilla=0.5, chocolate=0.5),
    }

def Likelihood(self, hypo, data):
    mix = self.mixes[hypo]
    like = mix[data]
    return like
```

`Likelihood` uses `mixes`, which is a dictionary that maps from the name of a bowl to the mix of cookies in the bowl.

Here's what the update looks like:

```
pmf.Update('vanilla')
```

And then we can print the posterior probability of each hypothesis:

```
for hypo, prob in pmf.Items():
    print hypo, prob
```

The result is

```
Bowl 1 0.6
Bowl 2 0.4
```

which is the same as what we got before. This code is more complicated than what we saw in the previous section. One advantage is that it generalizes easily to the case where we draw more than one cookie:

```
dataset = ['vanilla', 'chocolate', 'vanilla']
for data in dataset:
    pmf.Update(data)
```

The other advantage is that it provides a framework for solving many similar problems. In the next section we'll solve the Monty Hall problem computationally and then see what parts of the framework are the same.

The code in this section is available from `http://thinkbayes.com/cookie2.py`.

2.4 The Monty Hall problem

To solve the Monty Hall problem, I'll define a new class:

```
class Monty(Pmf):
    def __init__(self, hypos):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, 1)
        self.Normalize()

    def Update(self, data):
        for hypo in self.Values():
            like = self.Likelihood(hypo, data)
            self.Mult(hypo, like)
        self.Normalize()
```

So far `Monty` and `Cookie` are exactly the same. And the code that creates the Pmf is the same, too, except for the names of the hypotheses.

```
    hypos = 'ABC'
    pmf = Monty(hypos)
```

Calling `Update` is pretty much the same:

```
    data = 'B'
    pmf.Update(data)
```

And the implementation of `Update` is exactly the same:

```
    def Update(self, data):
        for hypo in self.Values():
            like = self.Likelihood(hypo, data)
            self.Mult(hypo, like)
        self.Normalize()
```

The only part that requires some work is `Likelihood`.

```
    def Likelihood(self, hypo, data):
        if hypo == data:
            return 0
        elif hypo == 'A':
            return 0.5
        else:
            return 1
```

Finally, printing the results is the same:

```
for hypo, prob in pmf.Items():
    print hypo, prob
```

And the answer is

```
A 0.333333333333
B 0.0
C 0.666666666667
```

In this example, writing `Likelihood` is a little complicated, but the framework of the Bayesian update is simple. The code in this section is available from `http://thinkbayes.com/monty.py`.

2.5 Encapsulating the framework

Now that we see what elements of the framework are the same, we can encapsulate them in an object: a `Suite` is a `Pmf` that provides `__init__`, `Update` and `Print`.

```
class Suite(Pmf):
    """Represents a suite of hypotheses and their probabilities."""

    def __init__(self, hypo=tuple()):
        """Initializes the distribution."""

    def Update(self, data):
        """Updates each hypothesis based on the data."""

    def Print(self):
        """Prints the hypotheses and their probabilities."""
```

The implementation of `Suite` is in `thinkbayes.py`. To use `Suite`, you should write a class that inherits from it and provides `Likelihood`. For example, here is the solution to the Monty Hall problem rewritten to use `Suite`:

```
from thinkbayes import Suite

class Monty(Suite):

    def Likelihood(self, hypo, data):
        if hypo == data:
            return 0
        elif hypo == 'A':
```

```
            return 0.5
        else:
            return 1
```

And here's the code that uses this class:

```
suite = Monty('ABC')
suite.Update('B')
suite.Print()
```

You can download this example at `http://thinkbayes.com/monty2.py`. If you are familiar with design patterns, you might recognize this as an example of the Template method pattern: the parent class, `Suite`, defines an algorithm framework; the child class, `Monty` provides the missing elements of the algorithm, in this case the `Likelihood` function. You can read about this pattern at `http://en.wikipedia.org/wiki/Template_method_pattern`.

2.6 The M&M problem

We can use the `Suite` framework to solve the M&M problem. Writing the `Likelihood` function is tricky, but everything else is straightforward.

First I need to encode the color mixes from before and after 1995.

```
mix94 = dict(brown=30,
             yellow=20,
             red=20,
             green=10,
             orange=10,
             tan=10)

mix96 = dict(blue=24,
             green=20,
             orange=16,
             yellow=14,
             red=13,
             brown=13)
```

Then I have to encode the hypotheses:

```
hypoA = dict(bag1=mix94, bag2=mix96)
hypoB = dict(bag1=mix96, bag2=mix94)
```

`hypoA` represents the hypothesis that Bag 1 is from 1994 and Bag 2 from 1996. `hypoB` is the other way around.

Next I map from the name of the hypothesis to the representation:

```
hypotheses = dict(A=hypoA, B=hypoA)
```

And finally I can write `Likelihood`. In this case the hypothesis, `hypo` is a string, `A` or `B`. The data is a tuple that specifies a bag, `bag1` or `bag2`, and a color.

```
def Likelihood(self, hypo, data):
    bag, color = data
    mix = self.hypotheses[hypo][bag]
    like = mix[color]
    return like
```

Here's the code that creates the suite and updates it:

```
suite = M_and_M('AB')

suite.Update(('bag1', 'yellow'))
suite.Update(('bag2', 'green'))

suite.Print()
```

And here's the result:

```
A 0.740740740741
B 0.259259259259
```

The posterior probability of A is approximately 20/27, which is what we got before.

The code in this section is available from `http://thinkbayes.com/m_and_m.py`.

Chapter 3

Estimation

3.1 Which die was it?

This problem is based on an example I saw in Sanjoy Mahajan's class on Bayesian inference.

Suppose I have a box of dice that contains a 4-sided die, a 6-sided die, an 8-sided die, a 12-sided die and a 20-sided die. If you have ever played Dungeons&Dragons, you know what I am talking about.

Suppose I select a die from the box at random, roll it, and get a 6. What is the probability that I rolled each die?

Let me suggest a three-step strategy for approaching a problem like this.

1. Choose a representation for the hypotheses.
2. Choose a representation for the data.
3. Write the likelihood function.

In previous examples I used strings to represent hypotheses and data, but for the die problem I'll use numbers. Specifically, I'll use the integers 4, 6, 8, 12, and 20 to represent hypotheses:

```
suite = Dice([4, 6, 8, 12, 20])
```

And integers from 1 to 20 for the data:

```
suite.Update(6)
```

I chose these representations because they make it easy to write the likelihood function:

```
class Dice(Suite):
    def Likelihood(self, hypo, data):
        if hypo < data:
            return 0
        else:
            return 1.0/hypo
```

Here's how `Likelihood` works. If `hypo < data`, that means the roll is greater than the number of sides on the die. That can't happen, so the likelihood is 0.

Otherwise the question is, "Given that there are `hypo` sides, what is the chance of rolling `data`?" The answer is `1/hypo`, regardless of `data`.

You can download this example from `http://thinkbayes.com/dice.py`. And here is the result:

```
4 0.0
6 0.392156862745
8 0.294117647059
12 0.196078431373
20 0.117647058824
```

After we roll a 6, the probability for the 4-sided die is 0. The most likely alternative is the 6-sided die, but there is still almost a 12% chance for the 20-sided die.

What if we roll a few more times and get 6, 8, 7, 7, 5, and 4?

```
    for roll in [6, 8, 7, 7, 5, 4]:
        suite.Update(roll)
```

With this data the 6-sided die is eliminated, and the 8-sided die seems quite likely. Here are the results:

```
4 0.0
6 0.0
8 0.943248453672
12 0.0552061280613
20 0.0015454182665
```

Now the probability is 94% that we are rolling the 8-sided die, and less than 1% for the 20-sided die.

3.2 The Locomotive problem

The locomotive problem is a classic also known as the "German tank problem." Here is the version that appears in Mosteller, *Fifty Challenging Prob-*

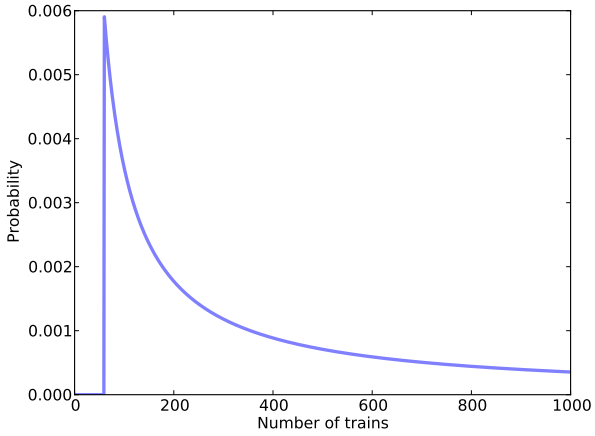

Figure 3.1: Posterior distribution for the Locomotive problem, based on a uniform prior.

lems in Probability:

> "A railroad numbers its locomotives in order 1..N. One day you see a locomotive with the number 60. Estimate how many locomotives the railroad has."

Based on this observation, we know the railroad has 60 or more locomotives. But how many more? To apply Bayesian reasoning, we can break this problem into two steps:

- What did we know about N before we saw the data?
- For any given value of N, what is the likelihood of seeing the data (a locomotive with number 60)?

The answer to the first question is the prior. The answer to the second is the likelihood.

We don't have much basis to choose a prior, but we can start with something simple and then evaluate some alternatives. So let's assume that N is less than 1000, and equally likely to be any value from 1 to 1000.

```
hypos = xrange(1, 1001)
suite = Train(hypos)
```

Now all we need is a likelihood function. With a little thought, you will realize that the likelihood functions for the Locomotive problem and the Dice problem are identical:

```
class Train(Suite):
    def Likelihood(self, hypo, data):
        if hypo < data:
            return 0
        else:
            return 1.0/hypo
```

And here's the update:

```
    suite.Update(60)
```

Since there are 1000 hypothesis, I didn't print the results. Instead I used `myplot.Pmf`:

```
    myplot.Pmf(suite)
    myplot.Show(xlabel='Number of trains',
                ylabel='Probability')
```

The result is in Figure 3.1. Not surprisingly, all values of N below 60 have been eliminated. The most likely value, if you had to guess, is 60. That might not seem like a very good guess; after all, what are the chances that you just happened to see the train with the highest number?

Nevertheless, if you want to maximize the chance of getting the answer exactly right, you should guess 60. But maybe that's not the right goal. An obvious alternative is to compute the mean of the posterior distribution:

```
def Mean(suite):
    total = 0
    for hypo, prob in suite.Items():
        total += hypo * prob
    return total

print Mean(suite)
```

Or you could use the very similar method provided by `Pmf`:

```
    print suite.Mean()
```

The mean of this posterior distribution is 276, so that might be a good guess if you wanted to minimize error. If you played this guessing game over and over, using the mean of the posterior as your estimate would minimize the total squared error over the long run.

You can download this example from `http://thinkbayes.com/train.py`.

3.3 What about that prior?

To make any progress on the Locomotive problem we had to make some assumptions, and some of them were pretty arbitrary. In particular, we chose a uniform prior from 1 to 1000, without much justification for choosing 1000, or for choosing a uniform distribution.

It is not crazy to believe that a railroad company might operate 1000 locomotives, but a reasonable person might guess more or fewer. So we might wonder whether the posterior distribution is sensitive to these assumptions. With so little data (only one observation) it probably is.

Recall that with a uniform prior from 1 to 1000, the mean of the posterior is 276. With an upper bound of 500, we get a posterior mean of 166, and with an upper bound of 2000, the posterior mean is 467.

So that's bad. There are two ways to proceed:

- Get more data.
- Get more background information.

With more data, the posterior distributions based on different priors tend to converge. For example, suppose that in addition to train 60 we also see trains 30 and 90. We can update the distribution like this:

```
for data in [60, 30, 90]:
    suite.Update(data)
```

With these data, the means of the posteriors are

Upper Bound	Posterior Mean
500	152
1000	164
2000	171

So the difference between the posteriors is smaller.

3.4 An alternative prior

If more data are not available, another option is to improve the priors by gathering more background information. It is probably not reasonable to

assume that a railroad company with 1000 locomotives is just as likely as a company with only 1.

With some effort, we could probably find a list of companies that operate locomotives in the area of observation. Or we could interview an expert in rail shipping to gather information about the typical size of companies.

But even without getting into the specifics of railroad economics, we can make some educated guesses. In most fields, there are many small companies, fewer medium-sized companies, and only one or two very large companies. In fact, the distribution of company sizes tends to follow a power law, as Robert Axtell reports in *Science* (see `http://www.sciencemag.org/content/293/5536/1818.full.pdf`).

This law suggests that if there are 1000 companies with fewer than 10 locomotives, there might be 100 companies with 100 locomotives, 10 companies with 1000, and possibly one company with 10,000 locomotives.

Mathematically, a power law means the number of companies with a given size is inversely proportional to the size, or

$$\text{PMF}(x) \propto \left(\frac{1}{x}\right)^{\alpha}$$

where $\text{PMF}(x)$ is the probability mass function of x and α is a parameter that is often near 1.

We can construct a power law prior like this:

```
class Train(Dice):
    def __init__(self, hypos, alpha=1.0):
        Pmf.__init__(self)
        for hypo in hypos:
            self.Set(hypo, hypo**(-alpha))
        self.Normalize()
```

And here's the code that constructs the prior:

```
    hypos = range(1, 1001)
```

Again, the upper bound is arbitrary, but with a power law prior, the posterior is less sensitive to this choice.

If we start with this prior and observe trains 30, 60 and 90, the means of the posteriors are

Upper Bound	Posterior Mean
500	131
1000	133
2000	134

Now the difference between the posteriors is much smaller. In fact, with an arbitrarily large upper bound, the mean converges on 134.

So the power law prior is more realistic, because it is based on general information about the size of companies, and it behaves better in practice.

You can download the examples in this section from `http://thinkbayes.com/train3.py`.

3.5 Credible intervals

Once you have computed a posterior distribution, it is often useful to summarize the results with a single point estimate or an interval. For point estimates it is common to use the mean, median, or the value with maximum likelihood.

For intervals we usually report two values computed so that there is a 90% chance that the unknown value falls between them, or 95%, or some other value. These values define a "credible interval."

A simple way to compute a credible interval is to add up the probabilities in the posterior distribution and record the values that correspond to probabilities 5% and 95%. In other words, the 5th and 95th percentiles.

`thinkbayes` provides a function that computes percentiles:

```
def Percentile(pmf, percentage):
    p = percentage / 100.0
    total = 0
    for val, prob in pmf.Items():
        total += prob
        if total >= p:
            return val
```

And here's the code that uses it:

```
    interval = Percentile(suite, 5), Percentile(suite, 95)
    print interval
```

For the previous example—the locomotive problem with a power law prior and three trains—the 90% credible interval is $(91, 243)$. The width of this range suggests, correctly, that we are still quite uncertain about how many locomotives there are.

3.6 Cumulative distribution function

In the previous section we computed percentiles by iterating through the values and probabilities in a Pmf. If we need to compute more than a few percentiles, it is more efficient to use a cumulative distribution function, or Cdf.

Cdfs and Pmfs are equivalent in the sense that they contain the same information about the distribution, and you can always convert from one to the other. The advantage of the Cdf is that you can compute percentiles more efficiently.

`thinkbayes` provides a `Cdf` class that represents a cumulative distribution function. It also provides functions for converting from a Pmf to a Cdf (and back).

```
import thinkbayes
cdf = thinkbayes.MakeCdfFromPmf(suite)
```

And Cdf provides a function named `Percentile`

```
    interval = cdf.Percentile(5), cdf.Percentile(95)
```

Converting from a Pmf to a Cdf takes time proportional to the number of values. The Cdf stores the values and probabilities in sorted lists, so looking up a probability to get the corresponding value takes "log time," that is, time proportional to the logarithm of the number of values. Looking up a value to get the corresponding probability is also logarithmic, so Cdfs are efficient for many calculations.

The examples in this section are in `http://thinkbayes.com/train.py`.

Chapter 4

More estimation

4.1 Four Urns

James Joyce, the American philosopher not the Irish author, wrote a paper called *How probabilities reflect evidence* that poses the following problem (which I am quoting loosely):

> Jacob and Emily both start out knowing that an urn, U, was randomly chosen from a set of four urns $urn_0, urn_1, urn_2, urn_3$ where urn_i contains three balls, i of which are blue and $3 - i$ of which are green. [...] Jacob and Emily regard random drawing with replacement as an exchangeable process, so that any series of draws that produces m blue balls and n green balls is as likely as any other such series, irrespective of order. Use $B^m G^n$ to denote the generic event in which m blue balls and n green balls are drawn at random and with replacement from U. Against this backdrop of shared evidence, suppose Jacob sees five balls drawn at random and with replacement from U and observes that all are blue, so his evidence is $B^5 G^0$. Emily, who sees Jacob's evidence, looks at fifteen additional draws of which twelve come up blue, so her evidence is $B^{17} G^3$. [If B_{next} is the probability that the next ball drawn from the urn is blue,] what should Emily and Jacob think about B_{next}?

We'll proceed in two steps: the first is to compute the probability that U is urn_i for each i. The second is to compute B_{next}.

Let's use H_i to represent the hypothesis that the urn we chose, U, is urn_i. Since we are told that U was chosen at random, we can infer that the prior probabilities are $H_i = 1/4$ for all i.

And the likelihood function is easy. If H_i is true, U contains i blue balls, so $\mathrm{p}(B|H_i) = i/3$. Or in Python

```
class Urn(Suite):
    def Likelihood(self, hypo, data):
        """
        hypo: integer number of blue balls (out of three)
        data: string 'B' or 'G'
        """
        p = hypo / 3.0
        if data == 'B':
            return p
        else:
            return 1-p
```

And here's the code that creates the prior and updates it with Jacob's data:

```
    suite = Urn([0,1,2,3])

    for data in 'BBBBB':
        suite.Update(data)
```

And here are the results:

```
0 0.0
1 0.0036231884058
2 0.115942028986
3 0.880434782609
```

Since he has seen 5 blue and no green balls, Jacob rules out urn_0 and gives low posterior probabilities to urn_1 and urn_2. Based on this data, Jacob is 88% sure that U is urn_3.

So what should Jacob believe about B_{next}, the chance that the next ball from U is blue. If he H_3 right, and U is urn_3, then B_{next} is 100%. But Jacob is not sure about H_3.

Jacob can compute B_{next} using the law of total probability, which says that the probability that the next ball is blue is the sum of the probabilities for each way it can happen; in this case

$$B_{next} = \sum \frac{1}{3} \mathrm{p}(H_i|D)$$

Pulling out the factor of $1/3$, we have

$$B_{next} = \frac{1}{3} \sum i \mathrm{p}(H_i|D)$$

which we recognize as the mean of the posterior distribution, over 3.

```
print suite.Mean() / 3.0
```

For Jacob, B_{next} is 0.96. For Emily, who sees 17 blue and 3 green balls, the posterior distribution is

```
0 0.0
1 3.05166468308e-05
2 0.999969483353
3 0.0
```

Since Emily has seen a blue and a green ball, she can rule out H_0 and H_3. And since she has seen many more blue than green balls, she is all but certain that U is urn_2, so her value for B_{next} is very close to 2/3.

Joyce constructed this example to point out a difference between the evidence seen by Jacob and Emily. Jacob's evidence has more "force," in one sense of the word, because his posterior belief about B_{next} is more strongly biased. But Emily's evidence has more "weight," because she is more confident about which urn U is.

For Joyce the point of this example is to probe what we mean when we talk about the "weight" or "force," of evidence. We come back to this question, from a Bayesian perspective, in the next chapter.

For us, the point of this example is to provide a warm-up exercise that will help with the next problem.

4.2 The Euro coin problem

In *Information Theory, Inference, and Learning Algorithms*, David MacKay poses this question:

> A statistical statement appeared in "The Guardian" on Friday January 4, 2002:
>
> > When spun on edge 250 times, a Belgian one-euro coin came up heads 140 times and tails 110. 'It looks very suspicious to me,' said Barry Blight, a statistics lecturer at the London School of Economics. 'If the coin were unbiased, the chance of getting a result as extreme as that would be less than 7%.'
>
> But do these data give evidence that the coin is biased rather than fair?

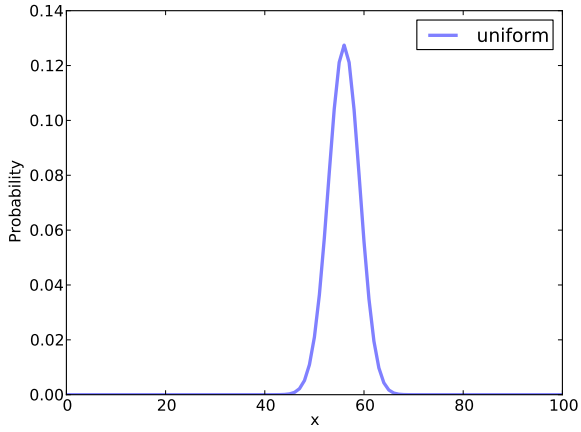

Figure 4.1: Posterior distribution for the Euro problem on a uniform prior.

To answer that question, we'll proceed in two steps. The first is to estimate the probability that the coin lands face up. The second is to evaluate whether the data support the hypothesis that the coin is biased.

Any given coin has some probability, x, of landing heads up when spun on edge. It seems reasonable to believe that the value of x depends on some physical characteristics of the coin, primarily the distribution of weight.

If a coin is perfectly balanced, we expect x to be close to 50%, but for a lopsided coin x might be substantially different. We can use Bayes's Theorem and the observed data to estimate x.

Let's define 101 hypotheses, where H_i is the hypothesis that x is i%, for values of i from 0 to 100. I'll start with a uniform prior where the probability of H_i is the same for all i. We'll come back later to consider other priors.

The likelihood function is easy; in fact, it is the same as in the Four Urn problem. If H_i is true, then the probability of heads is $i/100$; the probability of tails is $1 - i/100$.

```
class Euro(Suite):

    def Likelihood(self, hypo, data):
        x = hypo / 100.0
        if data == 'H':
            return x
        else:
            return 1-x
```

Here's the code that makes the suite and updates it:

```
suite = Euro(xrange(0, 101))
dataset = 'H' * 140 + 'T' * 110

for data in dataset:
    suite.Update(data)
```

And then we can plot the posterior.

```
myplot.Pmf(suite)
myplot.Show()
```

The result is in Figure 4.1.

4.3 Summarizing the posterior

Again, there are several ways to summarize the posterior distribution. One option is to find the most likely value in the posterior distribution. `thinkbayes` provides a function that does that:

```
def MaximumLikelihood(pmf):
    """Returns the value with the highest probability."""
    prob, val = max((prob, val) for val, prob in pmf.Items())
    return val
```

In this case the result is 56, which is also the observed percentage of heads, $140/250 = 0.56\%$. So that suggests (correctly) that the percentage in a sample is the maximum likelihood estimator for the population.

We might also summarize the posterior by computing the mean and median:

```
print 'Mean', suite.Mean()
print 'Median', thinkbayes.Percentile(suite, 50)
```

The mean is 55.95; the median is 56. Finally, we can compute a credible interval:

```
print 'CI', thinkbayes.CredibleInterval(suite, 90)
```

The result is $(51, 61)$.

Now, getting back to the original question, we would like to know whether the coin is fair. We observe that the posterior credible interval does not include 50%, which suggests that the coin is not fair.

But that is not exactly the question we started with. MacKay asked, " Do these data give evidence that the coin is biased rather than fair?" To answer

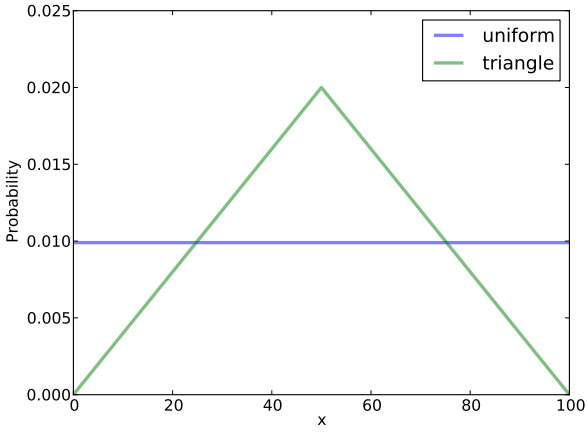

Figure 4.2: Uniform and triangular priors for the Euro problem.

that question, we will have to be more precise about what it means to say that data constitute evidence for a hypothesis. And that is the subject of the next chapter.

But before we go, I want to address one possible source of confusion. Since we want to know whether the coin is fair, it might be tempting to ask for the probability that `x` is 50%:

```
print suite.Prob(50)
```

The result is 0.021, but that value is pretty much meaningless. The decision to evaluate 101 hypotheses was arbitrary; we could have divided the range into more or fewer pieces, and if we had, the probability for any given hypothesis would be greater or less.

4.4 Swamping the priors

Again we started with a uniform prior, and again we have to acknowledge that it was not a very good choice. I can believe that if a coin is lop-sided, x might deviate substantially from 50%, but it seems unlikely that the Belgian Euro coin is so imbalanced that x is 10% or 90%.

So it might be more reasonable to choose a prior that gives higher probability to values of x near 50% and lower probability to extreme values.

As an example, I constructed a triangular prior, shown in Figure 4.2. Here's the code that constructs the prior:

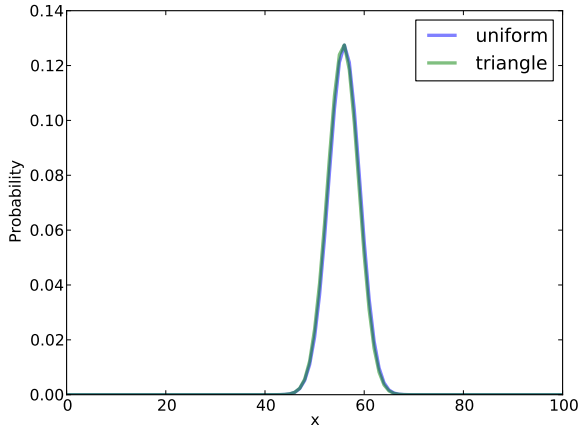

Figure 4.3: Posterior distributions for the Euro problem.

```
suite = Euro(xrange(0, 101))
for x in range(0, 51):
    suite.Set(x, x)
for x in range(51, 101):
    suite.Set(x, 100-x)
suite.Normalize()
```

Figure 4.2 shows the result. Updating this prior with the same dataset yields the posterior distribution shown in Figure 4.3.

The point of this example is that even with substantially different priors, the posterior distributions are very similar. The medians and the credible intervals are identical; the means differ by less than 0.5%.

This is an example of "swamping the priors:" with enough data, people who start with different priors will tend to converge on the same posterior.

4.5 Optimization

The code I have shown so far is meant to be easy to read, but it is not very efficient. In general, I like to develop code that is demonstrably correct, then check whether it is fast enough for my purposes. If so, there is no need to optimize. For this example, if we care about run time, there are several ways we can speed it up.

The first opportunity is to reduce the number of times we normalize the suite. In the original code, we call `Update` once for each spin.

```
dataset = 'H' * heads + 'T' * tails

for data in dataset:
    suite.Update(data)
```

And here's what `Update` looks like:

```
def Update(self, data):
    for hypo in self.Values():
        like = self.Likelihood(hypo, data)
        self.Mult(hypo, like)
    return self.Normalize()
```

Each update iterates through the hypotheses, then calls `Normalize`, which iterates through the hypotheses again. We can save some time by doing all of the updates before normalizing.

`Suite` provides a method called `UpdateSet` that does exactly that. Here it is:

```
def UpdateSet(self, dataset):
    for data in dataset:
        for hypo in self.Values():
            like = self.Likelihood(hypo, data)
            self.Mult(hypo, like)
    return self.Normalize()
```

And here's how we can invoke it:

```
dataset = 'H' * heads + 'T' * tails
suite.UpdateSet(dataset)
```

This optimization speeds things up, but the run time is still proportional to the amount of data. We can speed things up even more by rewriting `Likelihood` to process the entire dataset, rather than one spin at a time.

In the original version, `data` is a string that encodes either heads or tails:

```
def Likelihood(self, hypo, data):
    x = hypo / 100.0
    if data == 'H':
        return x
    else:
        return 1-x
```

As an alternative, we could encode the dataset as a tuple of two integers: the number of heads and tails. In that case `Likelihood` looks like this:

```
def Likelihood(self, hypo, data):
    x = hypo / 100.0
```

```
        heads, tails = data
        like = x**heads * (1-x)**tails
        return like
```

And then we can call `Update` like this:

```
heads, tails = 140, 110
suite.Update((heads, tails))
```

Since we have replaced repeated multiplication with exponentiation, this version takes the same time for any number of spins.

4.6 The beta distribution

There is one more optimization that solves this problem even faster.

So far we have used a Pmf object to represent a discrete set of possible values for `x`. Now we will use a continuous distribution, specifically the beta distribution (see `http://en.wikipedia.org/wiki/Beta_distribution`).

The beta distribution is defined on the interval from 0 to 1 (including both), so it is a natural choice for describing proportions and probabilities. But wait, it gets better.

It turns out that if you do a Bayesian update with a binomial likelihood function, as we did in the previous section, the beta distribution is a "conjugate prior." That means that if the prior distribution for `x` is a beta distribution, the posterior is also a beta distribution. But wait, it gets even better.

The shape of the beta distribution depends on two parameters, written α and β, or `alpha` and `beta`. If the prior is a beta distribution with parameters `alpha` and `beta`, and we see data with `h` heads and `t` tails, the posterior is a beta distribution with parameters `alpha+h` and `beta+t`. In other words, we can do an update with two additions.

So that's great, but it only works if the prior is beta. If beta distributions were unrealistic priors, they would not be very useful. But for many realistic priors there is a beta distribution that is at least a good approximation, and for a uniform prior there is a perfect match. The beta distribution with `alpha=1` and `beta=1` is uniform from 0 to 1.

Let's see how we can take advantage of all this. Here's a class that represents a beta distribution:

```
class Beta(object):

    def __init__(self, alpha=1, beta=1):
        self.alpha = alpha
        self.beta = beta
```

By default `__init__` makes a uniform distribution. `Update` performs a Bayesian update:

```
    def Update(self, data):
        heads, tails = data
        self.alpha += heads
        self.beta += tails
```

`data` is a pair of integers representing the number of heads and tails, or in the language of Bernoulli trials: successes and failures.

The mean of a beta distribution is a simple function of `alpha` and `beta`:

```
    def Mean(self):
        return float(self.alpha) / (self.alpha + self.beta)
```

So we have yet another way to solve the Euro problem:

```
    beta = Beta()
    beta.Update((140, 110))
    print beta.Mean()
```

The posterior mean is 56% (again). To get the probability for any value of `x`, we can evaluate the PDF of the beta distribution:

```
    def EvalPdf(self, x):
        return x**(self.alpha-1) * (1-x)**(self.beta-1)
```

This expression might look familiar. Here's `thinkbayes.EvalBinomialPmf`

```
def EvalBinomialPmf(x, yes, no):
    return x**yes * (1-x)**no
```

It's the same function, but in `EvalPdf`, we think of `x` as a random variable and `alpha` and `beta` as parameters; in `EvalBinomialPmf`, `x` is the parameter, and `yes` and `no` are random variables. Distributions like these that share the same PDF are called "conjugate distributions."

`Beta` is defined in `http://thinkbayes.com/thinkbayes.py`.

Chapter 5

Odds and addends

5.1 Odds

One way to represent a probability is with a number between 0 and 1, but that's not the only way. If you have ever bet on a football game or a horse race, you have probably encountered another representation of probability, called "odds."

You might have heard expressions like "the odds are three to one against," but you might not know what they mean. The odds in favor of an event are the ratio of the probability it will occur to the probability that it will not.

So if I think my team has a 75% chance of winning, I would say that the odds in their favor are three to one, because the chance of winning is three times the chance of losing.

You can write odds in decimal form, but it is most common to write them as a ratio of integers. So "three to one" is written 3 : 1.

When probabilities are low, it is more common to report the "odds against" rather than the odds in favor. For example, if I think my horse has a 10% chance of winning, I would say that the odds against are 9 : 1.

In the context of betting, it can be confusing to talk about odds, because the same format is used to report the payoff. For example, if I offer you a bet with a payoff of "two to one," that means I will pay you $2 for every $1 you bet (if you win). Of course, I would only offer you that bet if I thought the odds against you were more than 2 : 1.

But this is not a book about gambling, so I will only talk about odds and forget about payoffs.

Probabilities and odds are different representations of the same information. Given a probability, you can compute the odds like this:

```
def Odds(p):
    return p / (1-p)
```

Given the odds in favor, in decimal form, you can convert to probability like this:

```
def Probability(o):
    return o / (o+1)
```

If you represent odds with a numerator and denominator, you can convert to probability like this:

```
def Probability2(yes, no):
    return yes / (yes + no)
```

When I work with odds in my head, I find it helpful to picture people at the track. If 20% of them think my horse will win, then 80% of them don't, so the odds in favor are 20 : 80 or 1 : 4.

If the odds are 5 : 1 against my horse, then five out of six people think she will lose, so the probability of winning is 1/6.

5.2 The odds form of Bayes's Theorem

In Chapter 1 I wrote Bayes's Theorem in the "probability form":

$$\mathrm{p}(H|D) = \frac{\mathrm{p}(H)\,\mathrm{p}(D|H)}{\mathrm{p}(D)}$$

If we define $\bar{H}$ as "hypothesis H is false", we can write the odds in favor of H like this:

$$\frac{\mathrm{p}(H|D)}{\mathrm{p}(\bar{H}|D)} = \frac{\mathrm{p}(H)\,\mathrm{p}(D|H)}{\mathrm{p}(\bar{H})\,\mathrm{p}(D|\bar{H})}$$

Or writing $\mathrm{o}(H)$ for odds in favor of H:

$$\mathrm{o}(H|D) = \mathrm{o}(H)\frac{\mathrm{p}(D|H)}{\mathrm{p}(D|\bar{H})}$$

In words, this says that the posterior odds are the prior odds times the likelihood ratio. This is the "odds form" of Bayes's Theorem.

This form is most convenient for computing a Bayesian update on paper, or in your head. For example, let's go back to the cookie problem:

> Suppose there are two bowls of cookies. Bowl 1 contains 30 vanilla cookies and 10 chocolate cookies. Bowl 2 contains 20 of each.
>
> Now suppose you choose one of the bowls at random and, without looking, select a cookie at random. The cookie is vanilla. What is the probability that it came from Bowl 1?

The prior probability is 50%, so the prior odds are 1 : 1, or just 1. The likelihood ratio is $\frac{3}{4} / \frac{1}{2}$, or $3/2$. So the posterior odds are 3 : 2, which corresponds to probability $3/5$.

5.3 Oliver's blood

Here is another problem from MacKay's *Information Theory, Inference, and Learning Algorithms*:

> Two people have left traces of their own blood at the scene of a crime. A suspect, Oliver, is tested and found to have type 'O' blood. The blood groups of the two traces are found to be of type 'O' (a common type in the local population, having frequency 60%) and of type 'AB' (a rare type, with frequency 1%). Do these data [the traces found at the scene] give evidence in favor of the proposition that Oliver was one of the people [who left blood at the scene]?

To answer this question, we need to think about what it means for data to give evidence in favor of (or against) a hypothesis. Intuitively, we might say that data favor a hypothesis if the hypothesis is more likely in light of the data than it was before.

In the cookie problem, the prior odds are 1 : 1, or probability 50%. The posterior odds are 3 : 2, or probability 60%. So we could say that the vanilla cookie is evidence in favor of Bowl 1.

The odds form of Bayes's Theorem provides a way to make this intuition more precise. Again

$$\mathrm{o}(H|D) = \mathrm{o}(H)\frac{\mathrm{p}(D|H)}{\mathrm{p}(D|\bar{H})}$$

Or dividing through by $\mathrm{o}(H)$:

$$\frac{\mathrm{o}(H|D)}{\mathrm{o}(H)} = \frac{\mathrm{p}(D|H)}{\mathrm{p}(D|\bar{H})}$$

The term on the left is the ratio of the posterior and prior odds. The term on the right is the likelihood ratio, also called the "Bayes factor.

If the Bayes factor value is greater than 1, that means that the data were more likely under H than under $\bar{H}$. And since the odds ratio is also greater than 1, that means that the odds are greater, in light of the data, than they were before.

If the Bayes factor is less than 1, that means the data were less likely under H than under $\bar{H}$, so the odds in favor of H go down.

Finally, if the Bayes factor is exactly 1, the data are equally likely under either hypothesis, so the odds do not change.

Now we can get back to the Oliver's blood problem. If Oliver is one of the people who left blood at the crime scene, then he accounts for the 'O' sample, so the probability of the data is just the probability that a random member of the population has type 'AB' blood, which is 1%.

If Oliver did not leave blood at the scene, then we have two samples to account for. If we choose two random people from the population, what is the chance of finding one with type 'O' and one with type 'AB'? Well, there are two ways it might happen: the first person we choose might have type 'O' and the second 'AB', or the other way around. So the total probability is $2(0.6)(0.01) = 1.2\%$.

The likelihood of the data is slightly higher if Oliver is *not* one of the people who left blood at the scene, so the blood data is actually evidence against Oliver's guilt.

This example is a little contrived, but it is an example of the counterintuitive result that data *consistent* with a hypothesis are not necessarily *in favor of* the hypothesis.

If this result is so counterintuitive that it bothers you, this way of thinking might help: the data consist of a common event, type 'O' blood, and a rare event, type 'AB' blood. If Oliver accounts for the common event, that leaves the rare event still unexplained. If Oliver doesn't account for the 'O' blood, then we have two chances to find someone in the population with 'AB' blood. And that factor of two makes the difference.

5.4 Addends

Dungeons and Dragons is a role-playing game where the results of players' decisions are usually determined by rolling dice. In fact, before game

play starts, players generate each attribute of their characters—strength, intelligence, wisdom, dexterity, constitution, and charisma—by rolling three six-sided dice and adding them up.

So you might be curious to know the distribution of this sum. There are two ways you might compute it:

Simulation: Given a Pmf that represents the distribution for a single die, you can draw random samples, add them up, and accumulate the distribution of simulated sums.

Convolution: Given two Pmfs, you can enumerate all possible pairs of values and compute the distribution of the sums.

`thinkbayes` provides functions for both. Here's an example of the first approach. First, I'll define a class to represent a single die as a Pmf:

```
class Die(thinkbayes.Pmf):
    def __init__(self, sides):
        d = dict((i, 1) for i in xrange(1, sides+1))
        thinkbayes.Pmf.__init__(self, d)
        self.Normalize()
```

Now I can create a six-sided die:

```
    d6 = Die(6)
```

And use `SampleSum` to generate a sample of 1000 rolls.

```
    dice = [d6] * 3
    three = thinkbayes.SampleSum(dice, 1000)
```

`SampleSum` is based on another function, `RandomSum`, also in `thinkbayes.py`:

```
def RandomSum(dists):
    total = sum(dist.Random() for dist in dists)
    return total


def SampleSum(dists, n):
    pmf = MakePmfFromList(RandomSum(dists) for i in xrange(n))
    return pmf
```

`dists` is a list of objects that can be either Pmfs or Cdfs (or anything else that provides an appropriate `Random` method); `n` is the number of rolls to simulate.

The drawback of estimating this distribution by simulation is that it is only approximately correct. As `n` gets larger, it gets more accurate, but of course the run time increases as well.

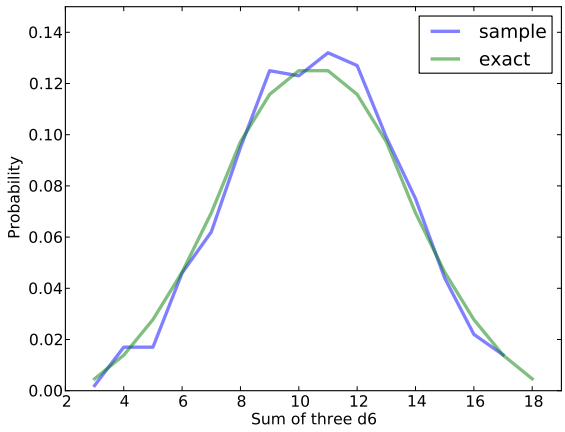

Figure 5.1: Approximate and exact distributions for the sum of three six-sided dice.

The other approach is to enumerate all pairs of values and compute the sum and probability of each pair. This is implemented in the `Pmf.__add__` method:

```
def __add__(self, other):
    pmf = Pmf()
    for v1, p1 in self.Items():
        for v2, p2 in other.Items():
            pmf.Incr(v1+v2, p1*p2)
    return pmf
```

And here's how it's used:

```
three_exact = d6 + d6 + d6
```

The approximate and exact distributions are shown in Figure 5.1.

`Pmf.__add__` is based on the assumption that the random selections from each Pmf are independent. In the example of rolling several dice, this assumption is pretty good. In other cases, we might have to extend this method to use conditional probabilities.

The code from this section is available from `http://thinkbayes.com/dungeons.py`.

5.5 Maxima

Having generated a Dungeons and Dragons character, I would be particularly interested in the character's best attributes, so I might wonder what

is the chance of getting an 18 in one or more attributes, or more generally what is the distribution of the best attribute.

There are three ways to compute the distribution of a maximum:

Simulation: Given a Pmf that represents the distribution for a single selection, you can a random samples, find the maximum, and accumulate the distribution of simulated maxima.

Convolution: Given two Pmfs, you can enumerate all possible pairs of values and compute the distribution of the maxima.

Exponentiation: If we convert a Pmf to a Cdf, there is a simple and efficient algorithm for finding the Cdf of the maxima.

The code to simulate maxima is almost identical to the code for simulating sums:

```
def RandomMax(dists):
    total = max(dist.Random() for dist in dists)
    return total


def SampleMax(dists, n):
    pmf = MakePmfFromList(RandomMax(dists) for i in xrange(n))
    return pmf
```

All I did was replace "sum" with "max". And the code for convolution is almost identical, too:

```
def PmfMax(pmf1, pmf2):
    res = thinkbayes.Pmf()
    for v1, p1 in pmf1.Items():
        for v2, p2 in pmf2.Items():
            res.Incr(max(v1, v2), p1*p2)
    return res
```

In fact, you could generalize this function by taking the appropriate operator as a parameter.

The only problem with this algorithm is that if each Pmf has n values, the run time is proportional to n^2.

If we convert the Pmfs to Cdfs, we can do the same calculation in linear time! The key is to remember the definition of the cumulative distribution function:

$$CDF(x) = \mathrm{p}(X \leq x)$$

where X is a random variable that means "a value chosen randomly from this distribution." So, for example, $CDF(5)$ is the probability that a value from this distribution is less than or equal to 5.

If I draw X from CDF_1 and Y from CDF_2, and compute the maximum $Z = max(X, Y)$, what is the chance that Z is less than or equal to 5? Well, in that case both X and Y must be less than or equal to 5.

If the selections of X and Y are independent,

$$CDF_3(5) = CDF_1(5)CDF_2(5)$$

where CDF_3 is the distribution of Z. I chose the value 5 because I think it makes the formulas easy to read, but we can generalize for any value of z:

$$CDF_3(z) = CDF_1(z)CDF_2(z)$$

In the special case where we draw n values from the same distribution,

$$CDF_n(z) = CDF_1(z)^n$$

And `Cdf` provides a method that does just that:

```
def Max(self, n):
    cdf = self.Copy()
    cdf.ps = [p**n for p in cdf.ps]
    return cdf
```

`Max` takes the sample size, `n`, and returns a new Cdf that represents the distribution of the maximum of `n` values.

`Pmf.Max` does the same thing for Pmfs. It has to do a little more work to convert the Pmf to a Cdf, so the run time is proportional to $n \log n$, but that's still better than quadratic.

Finally, here's an example that computes the distribution of your character's best attribute:

```
best_attr_cdf = three_exact.Max(6)
best_attr_pmf = thinkbayes.MakePmfFromCdf(best_attr_cdf)
```

Where `three_exact` is defined in the previous section. If we print the results, we see that the chance of generating a character with at least one attribute of 18 is about 3%.

[[[Include the figure.]]]

Chapter 6

Prediction

6.1 The Boston Bruins problem

In the 2010-11 National Hockey League (NHL) Finals, my beloved Boston Bruins played a best-of-seven championship series against the despised Vancouver Canucks. Boston lost the first two games 0-1 and 2-3, then won the next two games 8-1 and 4-0. At this point in the series, what is the probability that Boston will win the next game, and what is their probability of winning the championship?

As always, to answer a question like this, we need to make some assumptions. First, it is reasonable to believe that goal scoring in hockey is at least approximately a Poisson process, which means that it is equally likely for a goal to be scored at any time during a game. Second, we can assume that when team A plays team B, each team has some long-term average goals per game, denoted λ.

Given these assumptions, my strategy for answering this question is

1. Use statistics from previous games to choose a prior distribution for λ.

2. Use the score from the first four games to estimate λ for each team.

3. Use the posterior distributions of λ to compute distribution of goals for each team, the distribution of the goal differential, and the probability that each team wins.

4. Simulate the rest of the series to estimate the probability of each possible outcome.

To choose a prior distribution, I got some statistics from `http://www.nhl.com`, specifically the average goals per game for each team in the 2010-11 season. The distribution is roughly normal (Gaussian) with mean 2.8 and standard deviation 0.3.

The Gaussian distribution is continuous, but we'll approximate it with a discrete Pmf. `thinkbayes` provides `MakeGaussianPmf` to do exactly that:

```
def MakeGaussianPmf(mu, sigma, num_sigmas, n=101):
    pmf = Pmf()
    low = mu - num_sigmas*sigma
    high = mu + num_sigmas*sigma

    for x in numpy.linspace(low, high, n):
        p = EvalGaussianPdf(mu, sigma, x)
        pmf.Set(x, p)
    pmf.Normalize()
    return pmf
```

`mu` and `sigma` are the mean and standard deviation of the Gaussian distribution. `num_sigmas` is the number of standard deviations above and below the mean that the Pmf will span, and `n` is the number of values in the Pmf.

`EvalGaussianPdf` does what it says; it evaluates the Gaussian probability density function (PDF).

```
def EvalGaussianPdf(mu, sigma, x):
    z = (x - mu) / sigma
    p = math.exp(-z**2/2)
    return p
```

If you look up the Gaussian PDF, you'll see that I left out a normalizing constant, so this function actually computes a likelihood that is proportional to the Gaussian PDF. But that's fine, because I have to normalize the Pmf anyway.

Getting back to the hockey problem, here's the definition for a suite of hypotheses about the value of λ.

```
class Hockey(thinkbayes.Suite):

    def __init__(self):
        thinkbayes.Suite.__init__(self)

        pmf = thinkbayes.MakeGaussianPmf(2.7, 0.3, 6)
        for x, p in pmf.Items():
            self.Set(x, p)
```

So the prior distribution is Gaussian with mean 2.7, standard deviation 0.3, and it spans six sigmas above and below the mean.

As always, we have to decide how to represent each hypothesis; in this case I represent the hypothesis that $\lambda = x$ with the floating-point value `x`.

6.2 Poisson processes

In mathematical statistics, a "process" is a stochastic, or random, model of a physical system. For example, a Bernoulli process is a model of a sequence of events, called trials, in which each trial has two possible outcomes, like success and failure. So a Bernoulli process is a natural model for a series of coin flips, or a series of shots on goal.

A Poisson process is the continuous version of a Bernoulli process, where an event can occur at any point in time with equal probability. Poisson processes can be used to model customers arriving in a store, buses arriving at a bus stop, or goals scored in a hockey game.

In many real systems the probability of an event changes over time. Customers are more likely to go to a store at certain times of day, buses are supposed to arrive at fixed intervals, and goals are more or less likely at different times during a game.

But all models are based on simplifications, and in this case modeling a hockey game with a Poisson process is a reasonable choice. Heuer et al analyze scoring in a German soccer league and come to the same conclusion; see `http://www.cimat.mx/Eventos/vpec10/img/poisson.pdf`.

The benefit of using this model is that we can compute the distribution of goals per game efficiently, as well as the distribution of time between goals. Specifically, if the average number of goals in a game is `lam`, the distribution of goals per game is given by the Poisson PMF:

```
def EvalPoissonPmf(lam, k):
    return (lam)**k * math.exp(-lam) / math.factorial(k)
```

And the distribution of time between goals is given by the exponential PDF:

```
def EvalExponentialPdf(lam, x):
    return lam * math.exp(-lam * x)
```

I use the variable `lam` because `lambda` is a reserved keyword in Python. Both of these functions are in `thinkbayes.py`.

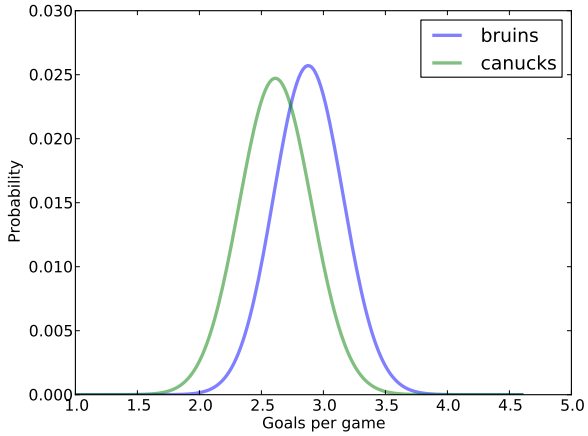

Figure 6.1: Posterior distribution of the number of goals per game.

6.3 The posteriors

Now we can compute the likelihood of seeing a team with a hypothetical value of `lam` score `k` goals in a game:

```
# class Hockey

    def Likelihood(self, hypo, data):
        lam = hypo
        k = data
        like = thinkbayes.EvalPoissonPmf(lam, k)
        return like
```

Each hypothesis is a possible value of λ; `data` is the observed number of goals, `k`.

With the likelihood function in place, we can make a suite for each team and update them with the scores from the first four games.

```
    suite1 = Hockey('bruins')
    suite1.UpdateSet([0, 2, 8, 4])

    suite2 = Hockey('canucks')
    suite2.UpdateSet([1, 3, 1, 0])
```

Figure 6.1 shows the resulting posterior distributions for `lam`. Based on the first four games, the most likely values for `lam` are 2.6 for the Canucks and 2.9 for the Bruins.

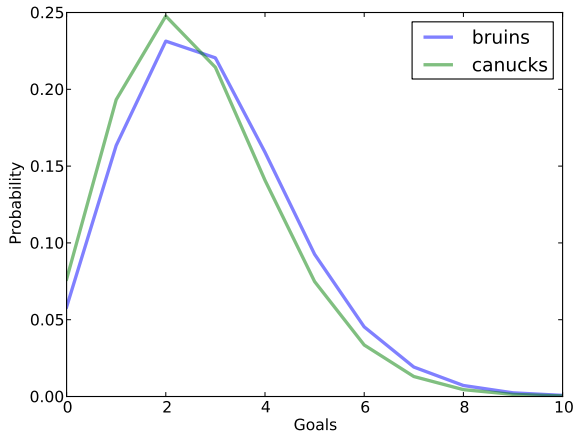

Figure 6.2: Distribution of goals in a single game.

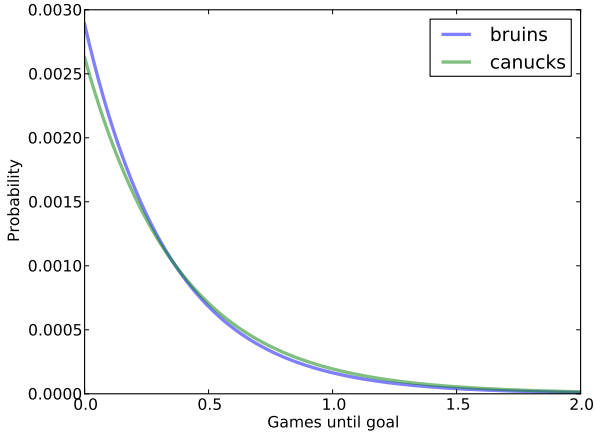

Figure 6.3: Distribution of time between goals.

6.4 The Meta-Pmf

To compute the probability that each team wins the next game, we need to compute the distribution of goals for each team.

If we knew the value of `lam` exactly, we could use the Poisson distribution again. `thinkbayes` provides a method that computes a truncated approximation of a Poisson distribution:

```
def MakePoissonPmf(lam, high):
    pmf = Pmf()
    for k in xrange(0, high+1):
        p = EvalPoissonPmf(lam, k)
        pmf.Set(k, p)
    pmf.Normalize()
    return pmf
```

The range of values in the computed Pmf is from 0 to `high`. So if the value of `lam` were exactly 3.4, we would compute:

```
lam = 3.4
goal_dist = thinkbayes.MakePoissonPmf(lam, 10)
```

I chose the upper bound, 10, because the probability of scoring more than 10 goals in a game is quite low.

That's simple enough so far; the problem is that we don't know the value of `lam` exactly. Instead, we have a distribution of possible values for `lam`.

For each value of `lam`, the distribution of goals is Poisson. So the overall distribution of goals is a mixture of these Poisson distributions, weighted according to the probabilities in the distribution of `lam`.

`thinkbayes` provides a method that computes mixtures of distributions:

```
def MakeMixture(metapmf):
    mix = Pmf()
    for pmf, p1 in metapmf.Items():
        for x, p2 in pmf.Items():
            mix.Incr(x, p1 * p2)
    return mix
```

`metapmf` is a Meta-Pmf, which means that it is a Pmf that contains Pmfs. If you find the Meta-Pmf confusing; it might help to think about how you would generate a random value from a Meta-Pmf. First you would choose a Pmf according to the probabilities in the Meta-Pmf. Then you would choose a value according to the probabilities in the chosen Pmf.

`MakeMixture` computes the distribution of the resulting values.

Given the posterior distribution of `lam`, here's the code that makes the distribution of goals:

```
def MakeGoalPmf(suite):
    metapmf = thinkbayes.Pmf()
    high = 10

    for lam, prob in suite.Items():
        pmf = thinkbayes.MakePoissonPmf(lam, high)
        metapmf.Set(pmf, prob)

    mix = thinkbayes.MakeMixture(metapmf)
    return mix
```

For each value of `lam` we make a Poisson Pmf and add it to the Meta-Pmf. Then we use `MakeMixture` to compute the mixture. If the Meta-Pmf makes your head hurt, don't panic. We are past the hard part of the problem now; everything else is easy.

Figure 6.2 shows the resulting distribution of goals for the Bruins and Canucks. As expected, the Bruins are less likely to score 3 goals or fewer in the next game, and more likely to score 4 or more.

6.5 The probability of winning

To get the probability of winning, we can convolve the distributions from the previous section to get the distribution of the goal differential:

```
goal_dist1 = MakeGoalPmf(suite1)
goal_dist2 = MakeGoalPmf(suite2)
diff = goal_dist1 - goal_dist2
```

If this differential is positive, the Bruins win; if negative, the Canucks win; if 0, it's a tie:

```
p_win = diff.ProbGreater(0)
p_loss = diff.ProbLess(0)
p_tie = diff.Prob(0)
```

With the distributions from the previous section, `p_win` is 46%, `p_loss` is 37%, and `p_tie` is 17%.

In the event of a tie at the end of regulation play, the teams play overtime periods until one team scores. Since the game ends immediately when the first goal is scored, this overtime format is known as "sudden death."

6.6 Sudden death

To compute the probability of winning in a sudden death overtime, the important statistic is not goals per game, but time until the first goal. The assumption that goal-scoring is a Poisson process is helpful here, because it implies that the time between goals is exponentially distributed.

Given `lam`, we can compute the time between goals like this:

```
lam = 3.4
time_dist = thinkbayes.MakeExponentialPmf(lam, high=2, n=101)
```

`high` is the upper bound of the distribution. In this case I chose 2, because the probability of going more than two games without scoring is small. `n` is the number of values in the Pmf.

If we know `lam` exactly, that's all there is to it. But we don't; instead we have a posterior distribution of possible values. So as we did with the distribution of goals, we have to make a Meta-Pmf and compute a mixture of Pmfs.

```
def MakeGoalTimePmf(suite):
    metapmf = thinkbayes.Pmf()

    for lam, prob in suite.Items():
        pmf = thinkbayes.MakeExponentialPmf(lam, high=2, n=2001)
        metapmf.Set(pmf, prob)

    mix = thinkbayes.MakeMixture(metapmf)
    return mix
```

Figure 6.2 shows the resulting distributions. For time values less than one period (one third of a game), the Bruins are more likely to score. The time until the Canucks score is more likely to be longer.

I set the number of values, `n`, fairly high in order to minimize the number of ties, since it is not actually possible for both teams to score simultaneously.

Now we can compute the probability that the Bruins score first:

```
    time_dist1 = MakeGoalTimePmf(suite1)
    time_dist2 = MakeGoalTimePmf(suite2)
    p_overtime = thinkbayes.PmfProbLess(time_dist1, time_dist2)
```

For the Bruins, the probability of winning in overtime is 52%. Finally, we can compute the total probability of winning:

```
p_tie = diff.Prob(0)
p_overtime = thinkbayes.PmfProbLess(time_dist1, time_dist2)

p_win = diff.ProbGreater(0) + p_overtime * p_tie
```

So the overall chance of winning the next game is 55%. To win the series, the Bruins can either win the next two games or split the next two and win the third. Again, we can compute the total probability:

```
# win the next two
p_series = p_win**2

# split the next two, win the third
p_series += 2 * p_win * (1-p_win) * p_win
```

The Bruins chance of winning the series is 57%. And in 2011, they did.

6.7 Opportunities for improvement

Solving problems like this is almost always iterative. In general, you want to start with something simple that yields an approximate answer, then look for opportunities to improve.

In this example, I would consider these options:

- I chose a prior based on the average goals per game for each team. But this statistic is averaged across all opponents. Against a particular opponent, we might expect more variability. For example, if the team with the best offense plays the team with the worst defense, the expected goals per game might be several standard deviations above the mean.

- For data I used only the first four games of the championship series. Assuming that the same teams played each other during the regular season, I could use the results from those games as well. One complication is that the composition of teams changes during the season due to trades and injuries. So it might be best to give more weight to recent games.

- To take advantage of all available information, we could use results from all regular season games to estimate each team's goal scoring rate, possibly adjusted by estimating an additional factor for each pairwise match-up. This approach would be more complicated, but it is still feasible. [[Forward reference to chess example]]

For the first option, we could use the results from the regular season to estimate the variability across all pairwise match-ups. Thanks to Dirk Hoag at `http://forechecker.blogspot.com/`, I was able to get the number of goals scored during regulation play (not overtime) for each game in the regular season.

Teams in different conferences only play each other one or two times in the regular season, so I focused on pairs that played each other 4–6 times. For each pair, I computed the average goals per game, which is an estimate of λ, then plotted the distribution of these estimates.

The mean of these estimates is 2.8, again, but the standard deviation is 0.85, substantially higher than what we got computing one estimate for each team.

If we run the analysis again with the higher-variance prior, the probability that the Bruins win the series is 80%, substantially higher than the result with the low-variance prior, 57%.

So this is another example where the results are sensitive to the prior. In general, if the variability in the prior is too high, the extreme values will be contradicted by data. But if the variability is too low, the effective range of the prior is too narrow, and the data may not be able to correct the error. So it is usually a good idea to err on the side of a more generous prior.

The code and data for this chapter are available from `http://thinkbayes.com/hockey.py` and `http://thinkbayes.com/hockey_data.csv`.

Chapter 7

Approximate Bayesian Computation

7.1 The Variability Hypothesis

I have a soft spot for crank science. Recently I visited Norumbega Tower, which is an enduring monument to the crackpot theories of Eben Norton Horsford, inventor of double-acting baking powder and fake history. But that's not what this chapter is about.

This article is about the Variability Hypothesis, which

> "originated in the early nineteenth century with Johann Meckel, who argued that males have a greater range of ability than females, especially in intelligence. In other words, he believed that most geniuses and most mentally retarded people are men. Because he considered males to be the 'superior animal,' Meckel concluded that females' lack of variation was a sign of inferiority."
>
> From `http://en.wikipedia.org/wiki/Variability_hypothesis`.

I particularly like that last part, because I suspect that if it turns out that women are actually more variable, Meckel would take that as a sign of inferiority, too. Anyway, you will not be surprised to hear that evidence for the Variability Hypothesis is mixed at best.

Nevertheless, it came up in my class recently when we looked at data from the CDC's Behavioral Risk Factor Surveillance System (BRFSS), specifically

the self-reported heights of adult American men and women. The dataset includes responses from 154407 men and 254722 women. Here's what we found:

- The average height for men is 178 cm; the average height for women is 163 cm. So men are taller, on average. No surprises so far.
- For men the standard deviation is 7.7 cm; for women is it 7.3 cm. So in absolute terms, men's heights are more variable.
- But to compare variability between groups, it is more meaningful to use the coefficient of variation (CV), which is the standard deviation divided by the mean. It is a dimensionless measure of variability relative to scale. For men CV is 0.0433; for women it is 0.0444.

That's very close, so we could conclude that this dataset provides little support for the Variability Hypothesis. But at the risk of overthinking the question, I want to use this problem to demonstrate Bayesian estimation in 2 dimensions.

So far we have worked with random processes that can be characterized by a single parameter: in the Euro problem it is the probability that the coin lands face up; in the Boston Bruin problem it is the goal scoring rate.

The techniques we used generalize easily to systems with more than one parameter. However, because the dataset is so big, we will run into some performance problems. I will proceed in a few steps:

1. We'll start with the simplest implementation, but it only works for datasets smaller than 1000 values.
2. By computing probabilities under a log transform, we can scale up to xxx of values, but the computation gets slow.
3. Finally, we speed things up with Approximate Bayesian Computation, also known as ABC.

7.2 Estimation in 2-D

Basic estimation in multiple dimensions is easy: we just represent each hypothesis with a tuple of parameters. For the normal distribution, the obvious tuple is `mu, sigma`.

For this problem, I defined a Suite called `Height` that represents a map from each `mu, sigma` pair to its probability:

```
class Height(thinkbayes.Suite):

    def __init__(self, mus, sigmas):
        thinkbayes.Suite.__init__(self)

        self.mus = mus
        self.sigmas = sigmas

        for mu in self.mus:
            for sigma in self.sigmas:
                self.Set((mu, sigma), 1)
```

`mus` is a sequence of possible values for `mu`; `sigmas` is a sequence of values for `sigma`. The prior distribution is uniform over all `mu, sigma` pairs.

The likelihood function is easy. Given hypothetical values of `mu` and `sigma`, we compute the likelihood of a particular value, `x`. That's what `EvalGaussianPdf` does, so all we have to do is use it:

```
# class Height
    def Likelihood(self, hypo, data):
        x = data
        mu, sigma = hypo

        like = thinkbayes.EvalGaussianPdf(mu, sigma, x)
        return like
```

If you have studied statistics from a mathematical perspective, you know that when you evaluate a PDF, you get a probability density. In order to get a probability, you have to integrate probability densities over some range.

But for our purposes, we don't need a probability; we just need something proportional to the probability we want. A probability density does that job nicely.

The hardest part of this problem turns out to be choosing appropriate ranges for `mus` and `sigmas`. If the range is too small, we omit some possibilities with non-negligible probability and get the wrong answer. If the range is too big, we get the right answer, but waste computational power.

So this is an opportunity to use classical estimation techniques to make Bayesian techniques more efficient. Specifically, we can use the sample to choose central values for `mu` and `sigma`, and use the standard errors of those estimates to choose the range.

If the true parameters of the distribution are μ and σ, and we take a sample of n values, the classical estimator of μ is

$$\hat{\mu} = \bar{x} = \frac{1}{n}\sum_i x_i$$

and the estimator of σ^2 is

$$\hat{\sigma}^2 = S^2 = \frac{1}{n-1}\sum_i (x_i - \bar{x})^2$$

The standard error of $\hat{\mu}$ is

$$S/\sqrt{n}$$

and the standard error of $\hat{\sigma}$ is

$$S/\sqrt{2(n-1)}$$

Here's the code to compute all that:

```
def FindPriorRanges(xs, num_points, num_stderrs=3.0):

    def MakeRange(estimate, stderr):
        spread = stderr * num_stderrs
        array = numpy.linspace(estimate-spread, estimate+spread, num_points)
        return array

    # estimate mean and stddev of xs
    n = len(xs)
    xbar, S2 = thinkstats.MeanVar(xs)
    sighat = math.sqrt(S2)

    print 'classical estimators', xbar, sighat

    # compute ranges for xbar and sighat
    stderr_xbar = sighat / math.sqrt(n)
    mus = MakeRange(xbar, stderr_xbar)

    stderr_sighat = sighat / math.sqrt(2 * (n-1))
    sigmas = MakeRange(sighat, stderr_sighat)

    return mus, sigmas
```

`xs` is the dataset. `num_points` is the desired number of values in the range.

`num_stderrs` is the width of the range on each side of the estimate, in number of standard errors. The return value is a pair of sequences, `mus` and `sigmas`.

And here's the code to make and update the suite:

```
mus, sigmas = FindPriorRanges(xs, num_points)
suite = Height(mus, sigmas, name)
suite.UpdateSet(xs)
print thinkbayes.MaximumLikelihood(suite)
```

This process might seem bogus, because we use the data to choose the range of the prior distribution, and then use the data again to do the update. In general, using the same data twice is, in fact, bogus.

But in this case it is ok. Really. We use the data to choose the range for the prior, but only to avoid computing a lot of probabilities that would have been very small anyway. I chose a value for `num_stderrs` big enough to cover all values with non-negligible likelihood. After that, making it bigger has no effect on the results.

In effect, we are approximating a prior that is uniform over all values of `mu` and `sigma`, but for computational efficiency we ignore all the values that don't matter.

Let's look at some results. Using the first 100 values from the BRFSS dataset—which includes 27 men, 71 women, and two people who did not report their heights— the maximum likelihood estimates for men are $(179, 8.7)$ cm, and for women $(163, 6.2)$ cm.

7.3 The posterior distribution of CV

These values are identical to the classical estimators, so if that's all we care about, we haven't accomplished much. The advantage of the Bayesian approach is that with the posterior distribution (not just a point estimate or credible interval), we can compute the distribution of CV for men and women, and the probability that one exceeds the other.

To compute the distribution of CV, we enumerate pairs of `mu` and `sigma`:

```
def ComputeCoefVariation(suite):
    pmf = thinkbayes.Pmf()
    for (mu, sigma), p in suite.Items():
        pmf.Incr(sigma/mu, p)
    return pmf
```

Given the distribution of CV for men and women, we can use `PmfProbGreater` to compute the probability that men are more variable.

Using the first 100 values from the BRFSS dataset, we would conclude that the probability is 96%, but there are two problems with this conclusion:

1. When we collapse the distributions to a single probability, we lose information about uncertainty. For example, if we report "The probability that men are more variable is 96%," that report would not capture the likelihood of the Variability Hypothesis before we saw the data or the weight of the evidence. We can address these limitations using Bayesian hypothesis testing, which is the topic of the next chapter.

2. The first 100 values are not a representative sample from the dataset. Of course, we should use the entire dataset. But that will turn out to be a little tricky.

7.4 Big data, small likelihood

If we select the first 1000 values from the dataset and run the program again, we get an error in `Pmf.Normalize`:

```
ValueError: total probability is zero.
```

The problem is that the likelihood function computes probability densities, and densities from continuous distributions tend to be small.

If you take a large number of small values and multiply them together, the result is very small. In this case it is so small it can't be represented by a floating-point number, so it gets rounded down to zero, which is called "underflow." And if all of the probabilities in the distribution are 0, it's not a distribution any more.

A possible solution is to renormalize the Pmf after each update, or after each batch of 100. But that would be slow.

A better alternative is to compute likelihoods under a log transform. That way, instead of multiplying small values, we can add up log likelihoods. `Pmf` provides methods `Log`, `LogUpdateSet` and `Exp` to make this process easy.

Here's what the update looks like:

```
suite.Log()
suite.LogUpdateSet(xs)
suite.Exp()
suite.Normalize()
```

It's important not to call `Pmf.Normalize` while the Pmf is under the log transform; the result would be nonsensical.

Here's the implementation of `Log`:

```
# class Pmf
    def Log(self):
        m = self.MaxLike()
        for x, p in self.d.iteritems():
            if p:
                self.Set(x, math.log(p/m))
            else:
                self.Remove(x)
```

`MaxLike` finds the highest probability in the Pmf. Since we divide all probabilities by `m`, the highest probability gets normalized to 1, which yields a log of 0.

If there are any values in the Pmf with probability 0, they are removed.

`LogUpdateSet` calls `LogUpdate`:

```
# class Suite
    def LogUpdateSet(self, dataset):
        for data in dataset:
            self.LogUpdate(data)
```

`LogUpdate` is just like `Update` except that it calls `LogLikelihood` instead of `Likelihood`, and `Incr` instead of `Mult`:

```
# class Suite
    def LogUpdate(self, data):
        for hypo in self.Values():
            like = self.LogLikelihood(hypo, data)
            self.Incr(hypo, like)
```

To compute the log-likelihood, we look up the Gaussian PDF again:

$$\frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right]$$

compute the log (dropping the constant term):

$$-\log\sigma - \frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$$

and translate into Python:

```
# class Height
    def LogLikelihood(self, hypo, data):
        x = data
        mu, sigma = hypo
        loglike = GaussianLogLikelihood(mu, sigma, x)
        return loglike


def GaussianLogLikelihood(mu, sigma, x):
    z = (x-mu) / sigma
    loglike = -math.log(sigma) - z**2 / 2
    return loglike
```

One advantage of this approach is that for many continuous distributions, it is easier to compute the log-likelihood than the likelihood.

Finally, here's how `Exp` inverts the transform:

```
# class Pmf
    def Exp(self):
        m = self.MaxLike()
        for x, p in self.d.iteritems():
            self.Set(x, math.exp(p-m))
```

Before applying `math.exp`, we find the maximum log-likelihood and shift so that the largest value in the distribution is 0, which yields likelihood 1. The result is a set of values we can add up with minimal loss of precision.

Using the log transform, we can scale the program up to a dataset of 10000 values, but the run time is in the range of minutes on my computer. To process the entire dataset might take an hour.

We can do better.

7.5 A little optimization

We can speed things up by providing a new version of `LogUpdateSet` that computes the log-likelihood of the entire dataset at once (rather than looping through the data). Given a sequence of values, x_i, the total log-likelihood is

$$\sum_i -\log\sigma - \frac{1}{2}\left(\frac{x_i - \mu}{\sigma}\right)^2$$

Pulling out the terms that don't depend on i, we get

$$-n \log \sigma - \frac{1}{2\sigma^2} \sum_i (x_i - \mu)^2$$

and translate into Python:

```
    def LogUpdateSetFast(self, data):
        xs = tuple(data)
        n = len(xs)

        for hypo in self.Values():
            mu, sigma = hypo
            total = Summation(xs, mu)
            loglike = -n * math.log(sigma) - total / 2 / sigma**2
            self.Incr(hypo, loglike)
```

By itself, this optimization would be a small improvement, but it creates an opportunity for a bigger improvement. Notice that the summation only depends on `mu`, not `sigma`, so we only have to evaluate it once for each value of `mu`. We could hoist it out of the loop, but a simpler alternative is to memoize `Summation` like this:

```
def Summation(xs, mu, cache={}):
    try:
        return cache[xs, mu]
    except KeyError:
        ds = [(x-mu)**2 for x in xs]
        total = sum(ds)
        cache[xs, mu] = total
        return total
```

`cache` stores previously-computed sums. The `try` statement returns a result from the cache if possible; otherwise it computes the summation, then caches and returns the result.

The only catch is that we can't use a list as a key in the cache, because it is not a hashable type. That's why `LogUpdateSetFast` converts the dataset to a tuple.

Together, these optimizations speed up the computation by about a factor of 100, processing the entire dataset (154 407 men and 254 722 women) in less than a minute on my not-very-fast computer.

7.6 ABC

But maybe you don't have that kind of time. In that case, Approximate Bayesian Computation (ABC) might be the way to go. The motivation behind ABC is that the likelihood of a particular dataset is:

- Very small, especially for large datasets, which is why we had to use the log transform,
- Expensive to compute, which is why we had to do so much optimization, and
- Not really what we want anyway.

We don't really care about the likelihood of seeing the exact dataset we saw. Especially for continuous variables, we care about the likelihood of seeing any dataset like the one we saw.

For example, in the Euro problem, we don't care about the order of the coin flips, only the total number of heads and tails. And in the train problem, we don't care about which particular trains were seen, only the number of trains and the maximum of the serial numbers.

Similarly, in the BRFSS sample, we don't really want to know the probability of seeing one particular set of values (especially since there are hundreds of thousands of them). It is more relevant to ask, "If we sample 100,000 people from a population with hypothetical values of μ and *sigma*, what would be the chance of collecting a sample with the observed mean and variance?"

For samples from a normal distribution, we can answer this question efficiently because we can find the distribution of the sample statistics analytically. In fact, we already did it when we computed the range of the prior.

If you draw n values from a normal distribution with parameters μ and *sigma*, and compute the sample mean, $\bar{x}$, you can think of $\bar{x}$ as a random variable, and its distribution is normal with parameters μ and $\sigma/\sqrt{n}$.

Similarly, the distribution of the sample standard deviation, S, is normal with parameters σ and $\sigma/\sqrt{2(n-1)}$.

We can use these sample distributions to compute the likelihood of any particular statistics, $\bar{x}$ and S, given hypothetical values for μ and *sigma*. Here's a new version of `LogUpdateSet` that does it:

```
    def LogUpdateSetABC(self, data):
        xs = data
        n = len(xs)

        # compute summary stats
        xbar, S2 = thinkstats.MeanVar(xs)
        sighat = math.sqrt(S2)

        for hypo in sorted(self.Values()):
            mu, sigma = hypo

            # compute log likelihood of xbar, given hypo
            sample_mu = mu
            sample_sigma = sigma / math.sqrt(n)
            loglike = GaussianLogLikelihood(sample_mu, sample_sigma, xbar)

            #compute log likelihood of sighat, given hypo
            sample_mu = sigma
            sample_sigma = sigma / math.sqrt(2 * (n-1))
            loglike += GaussianLogLikelihood(sample_mu, sample_sigma, sighat)

            self.Incr(hypo, loglike)
```

On my not-very-fast computer, this computation takes only a few seconds, and the result agrees with the "exact" result with about 5 digits of precision.

I put "exact" in quotes because in cases like this I think the ABC result is not an approximation of *the* correct answer, but a correct answer to a slightly different, and possibly more appropriate, question.

7.7 Who is more variable?

Finally we are ready to answer the question we started with: is the coefficient of variation greater for men than for women?

TO BE CONTINUED.

Chapter 8

Hypothesis testing

8.1 Back to the Euro problem

In Section 4.2 I presented a question from MacKay's *Information Theory, Inference, and Learning Algorithms*:

> A statistical statement appeared in "The Guardian" on Friday January 4, 2002:
>
> > When spun on edge 250 times, a Belgian one-euro coin came up heads 140 times and tails 110. 'It looks very suspicious to me,' said Barry Blight, a statistics lecturer at the London School of Economics. 'If the coin were unbiased, the chance of getting a result as extreme as that would be less than 7%.'
>
> But do these data give evidence that the coin is biased rather than fair?

We used this data to estimate the probability that the coin would land face up, but we didn't really answer MacKay's question. Let's get back to that now.

8.2 Bayesian hypothesis testing

In Section 5.3 I proposed that data are in favor of a hypothesis if the data are more likely under the hypothesis than under the alternative or, equivalently, if the Bayes factor is greater than 1.

In the Euro example, we have two hypotheses to consider: I'll use F for the hypothesis that the coin is fair and B for the hypothesis that it is biased.

If the coin is fair, it is easy to compute the likelihood of the data, $\mathrm{p}(D|F)$. In fact, we already wrote the `Likelihood` that does it.

```
def Likelihood(self, hypo, data):
    x = hypo / 100.0
    head, tails = data
    like = x**heads * (1-x)**tails
    return like
```

All we have to do is create a `Euro` suite and use it's `Likelihood` method:

```
suite = Euro()
likelihood = suite.Likelihood(50, data)
```

$\mathrm{p}(D|F)$ is $5.5 \cdot 10^{-76}$, which doesn't tell us much except that the probability of seeing any particular dataset is very small. As always, it takes two likelihoods to make a ratio, so we have to compute $\mathrm{p}(D|B)$.

It is not obvious how to compute the likelihood of B, because it's not obvious what "biased" means.

One possibility is to cheat and look at the data before we define the hypothesis. In that case we would say that "biased" means that the probability of heads is 140/250.

```
actual_percent = 100.0 * 140 / 250
likelihood = suite.Likelihood(actual_percent, data)
```

This version of B I call `B_cheat`; the likelihood of `b_cheat` is $34 \cdot 10^{-76}$ and the likelihood ratio is 6.1. So we would say that the data are evidence in favor of this version of B.

But using the data to formulate the hypothesis is obviously bogus. By that definition, any dataset would be evidence in favor of B, unless the observed percentage of heads is exactly 50%.

8.3 Making a fair comparison

To make a legitimate comparison, we have to define B without looking at the data. So let's try a different definition. If you inspect a Belgian Euro coin, you might notice that the "heads" side is more prominent than the "tails" side. You might expect that to effect x, but be unsure whether it makes

heads more or less likely. So you might say "I think the coin is biased so that x is either 0.6 or 0.4, but I am not sure which."

We can think of this version, which I'll call `B_two` as a hypothesis made up of two sub-hypotheses. We can compute the likelihood for each sub-hypothesis and then compute the average likelihood.

```
like40 = suite.Likelihood(40, data)
like60 = suite.Likelihood(60, data)
likelihood = 0.5 * like40 + 0.5 * like60
```

The likelihood ratio (or Bayes factor) for `b_two` is 1.3, which means the data provide weak evidence in favor of `b_two`.

More generally, suppose you suspect that the coin is biased, but you have no clue about the value of x. In that case you might build a Suite to represent sub-hypotheses from 0 to 100.

```
b_uniform = Euro(xrange(0, 101))
b_uniform.Remove(50)
b_uniform.Normalize()
```

I removed the sub-hypothesis that x is 50%, but it doesn't really matter if you include it. To compute the likelihood of `b_uniform` we compute the likelihood of the sub-hypotheses and compute a weighted average.

```
def SuiteLikelihood(suite, data):
    total = 0
    for hypo, prob in suite.Items():
        like = suite.Likelihood(hypo, data)
        total += prob * like
    return total
```

The likelihood ratio for `b_uniform` is 0.47, which means that the data are weak evidence against `b_uniform` at least compared to F.

If you think about the computation performed by `SuiteLikelihood`, you might notice that it is similar to an update. To refresh your memory, here's the `Update` function:

```
def Update(self, data):
    for hypo in self.Values():
        like = self.Likelihood(hypo, data)
        self.Mult(hypo, like)
    return self.Normalize()
```

And here's `Normalize`:

```
    def Normalize(self):
        total = self.Total()

        factor = 1.0 / total
        for x in self.d:
            self.d[x] *= factor

        return total
```

The return value from `Normalize` is the total of the probabilities in the Suite, which is the average of the likelihoods for the sub-hypotheses, weighted by the prior probabilities. And `Update` passes this value along, so we could compute the likelihood of `b_uniform` like this:

```
likelihood = b_uniform.Update(data)
```

8.4 The triangle prior

In Chapter 4.2 we also considered a triangle-shaped prior that gives higher probability to values of x near 50%. If we think of this prior as a suite of sub-hypotheses, we can compute its likelihood like this:

```
b_triangle = TrianglePrior()
likelihood = b_triangle.Update(data)
```

The likelihood ratio for `b_triangle` is 0.84, compared to F, so again we would say that the data are weak evidence against B.

The following table shows the different priors we have considered, the likelihood of each, and the likelihood ratio (or Bayes factor) relative to F.

Hypothesis	Likelihood $\times 10^{-76}$	Bayes Factor
F	5.5	–
`B_cheat`	34	6.1
`B_two`	7.4	1.3
`B_uniform`	2.6	0.47
`B_triangle`	4.6	0.84

In summary, we can use Bayesian hypothesis testing to compare the likelihood of F and B, but we have to do some work to specify precisely what B means. This specification depends on background information about coins and their behavior when spun, so people could reasonably disagree about the "right" definition.

Depending on which definition we choose, the data might provide evidence for or against the hypothesis that the coin is biased, but in either case it is relatively weak evidence.

I should acknowledge that my presentation of this example follows David MacKay's discussion, and comes to the same conclusion.

You can download the code I used in this chapter from `http://thinkbayes.com/euro3.py`.

Chapter 9

Evidence

9.1 Interpreting SAT scores

Suppose you are the Dean of Admission at a small engineering college in Massachusetts, and you are considering two candidates, Alice and Bob, whose qualifications are similar in many ways, with the exception that Alice got a higher score on the Math portion of the SAT, a standardized test intended to measure preparation for college-level work in mathematics.

If Alice got 780 and Bob got a 740 (out of a possible 800), you might want to know whether that difference is evidence that Alice is better prepared than Bob, and what the strength of that evidence is.

Now in reality, both scores are very good, and both candidates are probably well prepared for college math. So the real Dean of Admission would probably suggest that we choose the candidate who best demonstrates the other skills and attitudes we look for in students. But as an example of Bayesian hypothesis testing, let's stick with a narrower question: "How strong is the evidence that Alice is better prepared than Bob?"

To answer that question, we need to make some modeling decisions. I'll start with a simplification I know is wrong; then we'll come back and improve the model. Specifically, I will temporarily pretend that all SAT questions are equally difficult. In reality, the designers of the SAT choose questions with a range of difficulty, because that improves the ability to measure statistical differences between test-takers.

But if we choose a model where all questions are equally difficult, we can define a characteristic, `p_correct`, for each test-taker, which is the probabil-

ity of answering any question correctly. This simplification makes it easy to compute the likelihood of a given score.

9.2 The scale

In order to understand SAT scores, we have to understand the scoring and scaling process. Each test-taker gets a raw score based on the number of correct and incorrect questions. The raw score is converted to a scaled score in the range 200–800.

In 2009, there were 54 questions on the math SAT. The raw score for each test-taker is the number of questions answered correctly minus a penalty of 1/4 point for each question answered incorrectly.

The College Board, which administers the SAT, publishes the mapping from raw scores to scaled scores. I have downloaded that data and wrapped it in an Interpolater object that provides a forward lookup (from raw score to scaled) and a reverse lookup (from scaled score to raw).

You can download the code for this example from `http://thinkbayes.com/sat.py`.

9.3 The prior

The College Board also publishes the distribution of scaled scores for all test-takers. If we convert them to raw scores, we can compute the fraction of questions each test-taker got right, which is an estimate `p_correct`.

Then we can use the distribution of raw scores to model the prior distribution of `p_correct`. Here is the code that reads and processes the data:

```
class Exam(object):

    def __init__(self):
        self.scale = ReadScale()

        scores = ReadRanks()
        hist = Pmf.MakeHistFromDict(dict(scores))
        score_dist = Pmf.MakePmfFromHist(hist)

        raw = self.ReverseScale(score_dist)
        self.prior = DivideValues(raw, 54)
```

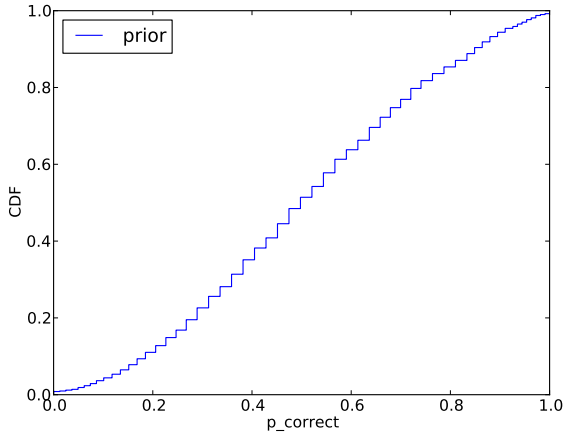

Figure 9.1: Prior distribution of `p_correct` for SAT test-takers.

`Exam` encapsulates the information we have about the offering of an exam. `self.scale` is the `Interpolater` that converts from raw to scaled scores and back. `self.prior` is a `Pmf` that maps from each raw score to a fraction of test-takers.

Figure 9.1 shows the prior distribution of `p_correct`. This distribution is approximately normal, but it is compressed at the extremes. By design, the SAT has the most power to discriminate between test-takers within two standard deviations of the mean, and less power outside that range.

For each test-taker, I define a Suite called `Sat` that represents the distribution of `p_correct`. Here's the definition:

```
class Sat(thinkbayes.Suite):

    def __init__(self, exam, score):
        thinkbayes.Suite.__init__(self)

        self.exam = exam
        self.score = score

        # start with the prior distribution
        for p_correct, prob in exam.prior.Items():
            self.Set(p_correct, prob)

        # update based on an exam score
        self.Update(score)
```

`__init__` takes the Exam object that encapsulates data about the exam, and

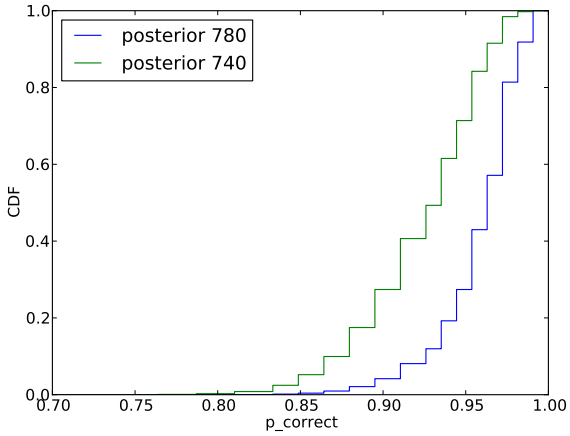

Figure 9.2: Posterior distributions of `p_correct` for Alice and Bob.

a scale score. It makes a copy of the prior distribution and then updates itself based on the exam score.

As usual, we inherit `Update` from `Suite` and provide `Likelihood`:

```
def Likelihood(self, hypo, data):
    p_correct = hypo
    score = data

    raw = self.exam.Reverse(score)
    yes, no = raw, 54 - raw

    like = thinkbayes.EvalBinomialPmf(p_correct, yes, no)
    return like
```

And as usual, `Likelihood` computes the likelihood of the data under a given hypothesis. In this case `hypo` is a hypothetical value of `p_correct`. And `data` is a scaled score.

To keep things simple, I interpret the raw score as the number of correct answers, ignoring the penalty for wrong answers. With this simplification, the likelihood is given by the binomial distribution, which computes the probability of `yes` correct answers and `no` incorrect.

9.4 Posterior

Figure 9.2 shows the posterior distributions of `p_correct` for Alice and Bob based on their exam scores. We can see that they overlap, so it is possible

that p_correct is actually higher for Bob, but it looks unlikely.

Which brings us back to the original question, "How strong is the evidence that Alice is better prepared than Bob?" We can use the posterior distributions of p_correct to answer this question.

To formulate the question in terms of Bayesian hypothesis testing, I define two hypotheses:

- A: p_correct is higher for Alice than for Bob.
- B: p_correct is higher for Bob than for Alice.

To compute the likelihood of A, we can enumerate all pairs of values from the posterior distributions, and add up the total probability of the cases where p_correct is higher for Alice than for Bob. And we already have a function, thinkbayes.PmfProbGreater that does that.

So we can define a Suite that computes the posterior probabilities of *A* and *B*:

```
class TopLevel(thinkbayes.Suite):

    def Update(self, data):
        a_sat, b_sat = data

        a_like = thinkbayes.PmfProbGreater(a_sat, b_sat)
        b_like = thinkbayes.PmfProbLess(a_sat, b_sat)
        c_like = thinkbayes.PmfProbEqual(a_sat, b_sat)

        a_like += c_like / 2
        b_like += c_like / 2

        self.Mult('A', a_like)
        self.Mult('B', b_like)

        self.Normalize()
```

Usually when we define a new Suite, we inherit Update and provide Likelihood. In this case I override Update, because it is easier to evaluate the likelihood of both hypotheses at the same time.

The data passed to Update are Sat objects that represent the posterior distributions of p_correct.

`a_like` is the total probability that `p_correct` is higher for Alice; `b_like` is that probability that it is higher for Bob.

`c_like` is the probability that they are "equal," but this equality is an artifact of the decision to model `p_correct` with a set of discrete values. If we chose more values, `c_like` would be smaller, and in the extreme, if `p_correct` were continuous, `c_like` would be zero. So I treat `c_like` as a kind of round-off error, and split it evenly between `a_like` and `b_like`.

Here is the code that creates `TopLevel` and updates it:

```
exam = Exam()
a_sat = Sat(exam, 780)
b_sat = Sat(exam, 740)

top = TopLevel('AB')
top.Update((a_sat, b_sat))
top.Print()
```

The likelihood of A is 0.79 and the likelihood of B is 0.21. The likelihood ratio (or Bayes factor) is 3.8, which means that these test scores are evidence that Alice is better than Bob at answering SAT questions. If we believed, before seeing the test scores, that A and B were equally likely, then after seeing the scores we should believe that the probability of A is 79%, which means there is still an 21% chance that Bob is actually better prepared.

9.5 A better model

Remember that the analysis we have done so far is based on the simplification that all SAT questions are equally difficult. In reality, some are easier than others, which means that the difference between Alice and Bob might be even smaller.

To see why, consider a test where most questions are so easy that Alice and Bob are nearly certain to answer them correctly. In that case, the difference between their scores is determined by a small number of difficult questions, so it is more likely to be due to chance.

But how big is the difference? If it is small, we would conclude that the first model—based on the simplification that all questions are equally difficult—is probably good enough. If it's large, we need a better model.

In the next few sections, I develop a better model and we will discover (spoiler alert!) that the difference is small. So if you are satisfied with the

simpler model, you can skip to the next chapter. If you want to see how the more realistic model works, read on...

- First, let's assume that each test-taker has some degree of `efficacy`, that measures their ability to answer SAT questions.
- Second, assume that each question has some level of `difficulty`.
- Finally, assume that the chance that a test-taker answers a question correctly is related to `efficacy` and `difficulty` according to this function:

```
def ProbCorrect(efficacy, difficulty, a=1):
    return 1 / (1 + math.exp(-a * (efficacy - difficulty)))
```

This function is a simplified version of the curve used in item response theory, which you can read about at `http://en.wikipedia.org/wiki/Item_response_theory`. `efficacy` and `difficulty` are considered to be on the same scale, and the probability of getting a question right depends only on the difference between them.

When `efficacy` and `difficulty` are equal, the probability of getting the question right is 50%. As `efficacy` increases, this probability approaches 100%. As it decreases (or as `difficulty` increases), the probability approaches 0%.

Given the distribution of `efficacy` across test-takers and the distribution of `difficulty` across questions, we can compute the expected distribution of raw scores. We'll do that in two steps. First, for a person with given `efficacy`, we'll compute the distribution of raw scores.

```
def PmfCorrect(efficacy, difficulties):
    pmf0 = thinkbayes.Suite([0])

    ps = [ProbCorrect(efficacy, diff) for diff in difficulties]
    pmfs = [BinaryPmf(p) for p in ps]
    dist = sum(pmfs, pmf0)
    return dist
```

`difficulties` is a list of difficulties, one for each question. `ps` is a list of probabilities, and `pmfs` is a list of two-valued Pmf objects; here's the function that makes them:

```
def BinaryPmf(p):
    pmf = thinkbayes.Pmf()
```

```
    pmf.Set(1, p)
    pmf.Set(0, 1-p)
    return pmf
```

`dist` is the sum of these Pmfs. Remember from Section 5.4 that when we add up Pmf objects, the result is the distribution of the sums. In order to use Python's `sum` to add up Pmfs, we have to provide `pmf0` which is the identity for Pmfs, so `pmf + pmf0` is always `pmf`.

So if we know a person's efficacy, we can compute their distribution of raw scores. For a group of people with a different efficacies, the resulting distribution of raw scores is a mixture. Here's the code that computes the mixture:

```
class Exam:

    def MakeRawScoreDist(self, efficacies):
        pmfs = thinkbayes.Pmf()
        for efficacy, prob in efficacies.Items():
            scores = PmfCorrect(efficacy, self.difficulties)
            pmfs.Set(scores, prob)

        mix = thinkbayes.MakeMixture(pmfs)
        return mix
```

`efficacies` is a Pmf that represents the distribution of efficacy across test-takers. I assume that it is Gaussian with mean 0 and standard deviation 1.5. This choice is mostly arbitrary. The probability of getting a question correct depends on the difference between efficacy and difficulty, so we can choose the units of efficacy and then calibrate the units of difficulty accordingly.

`pmfs` is a Meta-Pmf that contains one Pmf for each level of efficacy, and maps to the fraction of test-takers at that level. `thinkbayes.MakeMixture` takes the meta-pmf and compute the distribution of the mixture (see Section 6.4).

9.6 Calibration

If we were given the distribution of difficulty, we could use `MakeRawScoreDist` to compute the distribution of raw scores. But for us the problem is the other way around: we are given the distribution of raw scores and we want to infer the distribution of difficulty.

I assume that the distribution of difficulty is uniform with parameters `center` and `width`. `MakeDifficulties` makes a list of difficulties with these parameters.

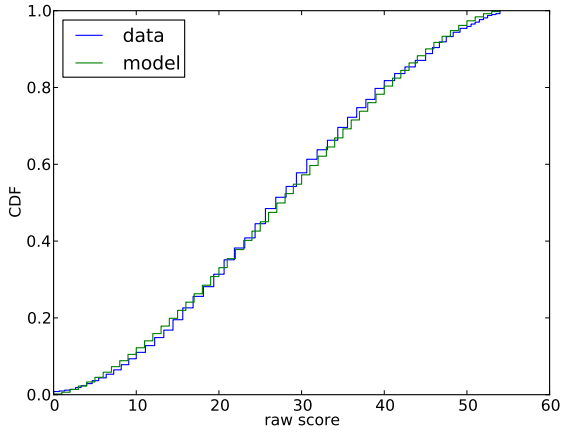

Figure 9.3: Actual distribution of raw scores and a model to fit it.

```
def MakeDifficulties(center, width, n):
    low, high = center-width, center+width
    return numpy.linspace(low, high, n)
```

By trying out a few combinations, I found that the values `center=-0.05` and `width=1.8` yield a distribution of raw scores similar to the actual data. Figure 9.3 shows the actual data and the results from this model.

So, assuming that the distribution of difficulty is uniform, the range is approximately `-1.85` to `1.75`, on a scale with a distribution of efficacy that is Gaussian with mean 0 and standard deviation 1.5.

The following table shows the range of `ProbCorrect` for test-takers at different levels of efficacy:

	Difficulty		
Efficacy	-1.85	-0.05	1.75
3.00	0.99	0.95	0.78
1.50	0.97	0.82	0.44
0.00	0.86	0.51	0.15
-1.50	0.59	0.19	0.04
-3.00	0.24	0.05	0.01

Someone with efficacy 3 (two standard deviations above the mean) has a 99% chance of answering the easiest questions on the exam, and a 78% chance of answering the hardest. On the other end of the range, someone two standard deviations below the mean has only a 24% chance of answering the easiest questions.

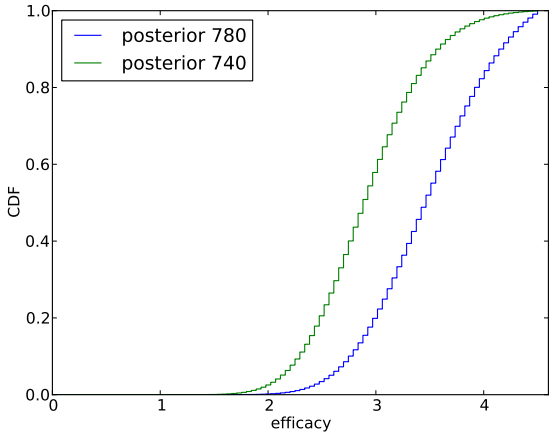

Figure 9.4: Posterior distributions of efficacy for Alice and Bob.

9.7 Posterior distribution of efficacy

Now that the model is calibrated, we can compute the posterior distribution of efficacy for Alice and Bob. Here is a version of the Sat class that works with the new model:

```
class Sat2(thinkbayes.Suite):

    def __init__(self, exam, score):
        thinkbayes.Suite.__init__(self)

        self.exam = exam
        self.score = score

        # start with the prior distribution
        efficacies = thinkbayes.MakeGaussianPmf(0, 1.5, 3)
        for efficacy, prob in efficacies.Items():
            self.Set(efficacy, prob)

        # update based on an exam score
        self.Update(score)
```

The prior distribution of efficacy is Gaussian with mean 0 and standard deviation 1.5 (and the suite of hypotheses spans 3 standard deviations in each direction).

`Sat2.Likelihood` computes the likelihood of a given test score for a hypothetical level of efficacy.

```
def Likelihood(self, hypo, data):
    efficacy = hypo
    score = data
    raw = self.exam.Reverse(score)

    pmf = self.exam.PmfCorrect(efficacy)
    like = pmf.Prob(raw)
    return like
```

`pmf` is the distribution of raw scores for a test-taker with the given efficacy; `like` is the probability of the observed score.

Figure 9.4 shows the posterior distributions of efficacy for Alice and Bob. As expected, the location of Alice's distribution is farther to the right, but again there is some overlap.

Using `TopLevel` again, we compare A, the hypothesis that Alice's efficacy is higher, and B, the hypothesis that Bob's is higher. The likelihood ratio is 3.4, a bit smaller than what we got from the simple model (3.8). So this model indicates that the data are evidence in favor of A, but a little weaker than the previous estimate.

If our prior belief is that A and B are equally likely, then in light of this evidence we would give A a posterior probability of 77%, leaving a 23% chance that Bob's efficacy is higher.

9.8 Predictive distribution

The analysis we have done so far generates estimates for Alice and Bob's efficacy, but since efficacy is not directly observable, it is hard to validate the results.

To give the model predictive power, we can use it to answer another related question: "If Alice and Bob take the math SAT again, what is the chance that Alice will do better again?"

We'll answer this question in two steps:

- We'll use the posterior distribution of efficacy to generate a predictive distribution of raw score for each test-taker.
- We'll compare the two predictive distributions to compute the probability that Alice gets a higher score again.

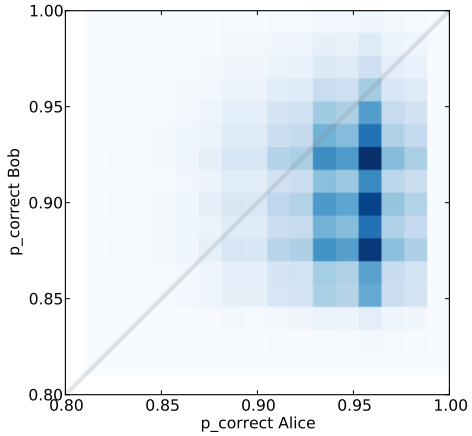

Figure 9.5: Joint posterior distribution of `p_correct` for Alice and Bob.

We already have most of the code we need. To compute the predictive distributions, we can use `MakeRawScoreDist` again:

```
exam = Exam()
a_sat = Sat(exam, 780)
b_sat = Sat(exam, 740)

a_pred = exam.MakeRawScoreDist(a_sat)
b_pred = exam.MakeRawScoreDist(b_sat)
```

Then we can find the likelihood that Alice does better on the second test, Bob does better, or they tie:

```
a_like = thinkbayes.PmfProbGreater(a_pred, b_pred)
b_like = thinkbayes.PmfProbLess(a_pred, b_pred)
c_like = thinkbayes.PmfProbEqual(a_pred, b_pred)
```

The probability that Alice does better on the second exam is 63%, which means that Bob has a 37% chance of doing as well or better.

Notice that we have more confidence about Alice's efficacy than we do about the outcome of the next test. The posterior odds are 3:1 that Alice's efficacy is higher, but only 2:1 that Alice will do better on the next exam.

9.9 Nuisance parameters

In this chapter we started with the question, "How strong is the evidence that Alice is better prepared than Bob?" On the face of it, that sounds like we want to test two hypotheses: either Alice is more prepared or Bob is.

But in order to compute likelihoods for these hypotheses, we have to solve an estimation problem. For each test-taker we have to find the posterior distribution of either `p_correct` or `efficacy`.

We don't really care about these values, but we have to estimate them to solve the problem we care about. Which is why values like this are called "nuisance parameters."

One way to visualize the analysis we did in this chapter is to plot the space of these parameters. `thinkbayes.JointPmf` takes two Pmfs, computes their joint distribution, and returns a Pmf of each possible pair of values and its probability.

```
def JointPmf(pmf1, pmf2):
    pmf = Pmf()
    for v1, p1 in pmf1.Items():
        for v2, p2 in pmf2.Items():
            pmf.Set((v1, v2), p1 * p2)
    return pmf
```

This function assumes that the two distributions are independent.

Figure 9.5 shows the joint posterior distribution of `p_correct` for Alice and Bob. The diagonal like indicates the part of the space where `p_correct` is the same for Alice and Bob. To the right of this line, Alice is more prepared; to the left Bob is more prepared.

In `TopLevel.Update`, when we computed the likelihoods of A and B, we added up the probability mass each side of this line. For the cells that fall on the line, we added up the total mass and split it between A and B.

The process we used in this chapter—estimating a nuisance variable in order to evaluate the likelihood of competing hypotheses—is a common Bayesian approach to problems like this.

Chapter 10

Simulation

In this chapter I describe my solution to a problem posed by a patient with a kidney tumor. I think the problem is interesting, and important, because it is relevant to both the patient and doctors treating other patients with these tumors.

And I think the solution is interesting because, although it is a Bayesian approach to the problem, the use of Bayes's Theorem is implicit. I present the solution and my code; at the end of the chapter I will explain the Bayesian part.

If you want more technical detail than I present here, you can read my paper on this work at `http://arxiv.org/abs/1203.6890`.

10.1 The Kidney Tumor problem

I am a frequent reader and occasional contributor to the online statistics forum at `http://reddit.com/r/statistics`. In November 2011, I read the following message:

> "I have Stage IV Kidney Cancer and am trying to determine if the cancer formed before I retired from the military. ... Given the dates of retirement and detection is it possible to determine when there was a 50/50 chance that I developed the disease? Is it possible to determine the probability on the retirement date? My tumor was 15.5 cm x 15 cm at detection. Grade II."

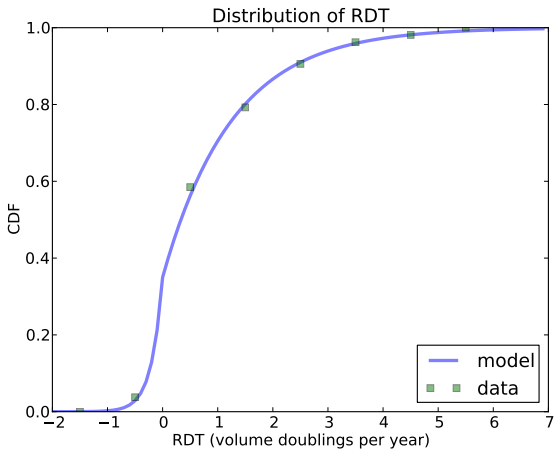

Figure 10.1: CDF of RDT in doublings per year.

I contacted the original poster and got more information; I learned that veterans get different benefits if it is "more likely than not" that a tumor formed while they were in military service (among other considerations).

Because renal tumors grow slowly, and often do not cause symptoms, they are sometimes left untreated. As a result, doctors can observe the rate of growth for untreated tumors by comparing scans from the same patient at different times. Several papers have reported these growth rates.

I collected data from a paper by Zhang et al [1]. I contacted the authors to see if I could get raw data, but they refused on grounds of medical privacy. Nevertheless, I was able to extract the data I needed by printing one of their graphs and measuring it with a ruler.

They report growth rates in reciprocal doubling time (RDT), which is in units of doublings per year. So a tumor with $RDT = 1$ doubles in volume each year; with $RDT = 2$ it quadruples in the same time, and with $RDT = -1$, it halves. Figure 10.1 shows the distribution of RDT for 53 patients.

The squares are the data points from the paper; the line is a model I fit to the data. The positive tail fits an exponential distribution well, so I used a mixture of two exponentials.

[1] Zhang et al, Distribution of Renal Tumor Growth Rates Determined by Using Serial Volumetric CT Measurements, January 2009 *Radiology*, 250, 137-144.

10.2 A simple model

It is usually a good idea to start with a simple model before trying something more challenging. Sometimes the simple model is sufficient for the problem at hand, and if not, you can use it to validate the more complex model.

For my simple model, I assume that tumors grow with a constant doubling time, and that they are three-dimensional in the sense that if the maximum linear measurement doubles, the volume is multiplied by eight.

I learned from my correspondent that the time between his discharge from the military and his diagnosis was 3291 days (about 9 years). So my first calculation was, "If this tumor grew at the median rate, how big would it have been at the date of discharge?"

The median volume doubling time reported by Zhang et al is 811 days. Assuming 3-dimensional geometry, the doubling time for a linear measure is three times longer.

```
# time between discharge and diagnosis, in days
interval = 3291.0

# doubling time in linear measure is doubling time in volume * 3
dt = 811.0 * 3

# number of doublings since discharge
doublings = interval / dt

# how big was the tumor at time of discharge (diameter in cm)
d1 = 15.5
d0 = d1 / 2.0 ** doublings
```

You can download the code in this chapter from `http://thinkbayes.com/kidney.py`.

The result, `d0`, is about 6 cm. So if this tumor formed after the date of discharge, it must have grown substantially faster than the median rate. Therefore I concluded that it is "more likely than not" that this tumor formed before the date of discharge.

In addition, I computed the growth rate that would be implied if this tumor had formed after the date of discharge. If we assume an initial size of 0.1 cm, we can compute the number of doublings to get to a final size of 15.5 cm:

```
# assume an initial linear measure of 0.1 cm
d0 = 0.1
d1 = 15.5

# how many doublings would it take to get from d0 to d1
doublings = log2(d1 / d0)

# what linear doubling time does that imply?
dt = interval / doublings

# compute the volumetric doubling time and RDT
vdt = dt / 3
rdt = 365 / vdt
```

The number of doublings, in linear measure, is 7.3, which implies an RDT of 2.4. In the data from Zhang et al, only 20% of tumors grew this fast during a period of observation. So again, I concluded that is "more likely than not" that the tumor formed prior to the date of discharge.

These calculations are sufficient to answer the question as posed, and on behalf of my correspondent, I wrote a letter explaining my conclusions to the Veterans' Benefit Administration.

Later I told a friend, who is an oncologist, about my results. He was surprised by the growth rates observed by Zhang et al, and by what they imply about the ages of these tumors. He suggested that the results might be interesting to researchers and doctors.

But in order to make them useful, I wanted a more general model of the relationship between age and size.

10.3 A more general model

Given the size of a tumor at time of diagnosis, it would be most useful to know the probability that the tumor formed before any given date; in other words, the distribution of ages.

To find it, I will run simulation of tumor growth to get the distribution of size, conditioned on age. Then we can get the distribution of age, conditioned on size.

The simulation starts with a small tumor and runs these steps:

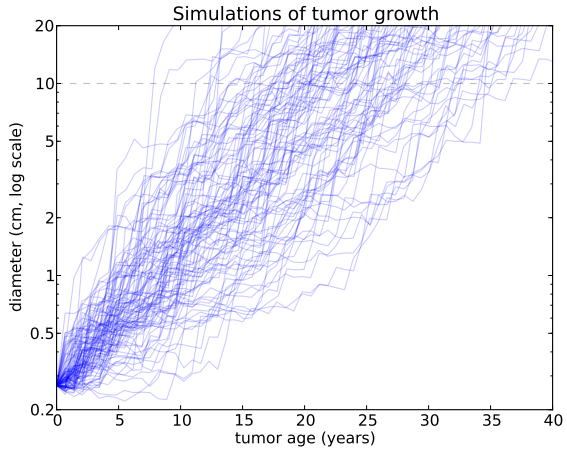

Figure 10.2: Simulations of tumor growth, size vs. time.

- Choose a growth rate from the distribution of RDT.
- Compute the size of the tumor at the end of an interval.
- Record the size of the tumor at each interval
- Repeat until the tumor exceeds the maximum relevant size.

For the initial size I chose 0.3 cm, because carcinomas smaller than that are less likely to be invasive and less likely to have the blood supply needed for rapid growth (see `http://en.wikipedia.org/wiki/Carcinoma_in_situ`).

I chose an interval of 245 days (about 8 months) because that is the median time between measurements in the data source.

For the maximum size I chose 20 cm. In the data source, the range of observed sizes is 1.0 to 12.0 cm, so we are extrapolating beyond the observed range at each end, but not by far, and not in a way likely to have a strong effect on the results.

The simulation is based on one big simplification: the growth rate is chosen independently during each interval, so it does not depend on age, size, or growth rate during previous intervals.

In Section 10.7 I will review these assumptions and consider more detailed models. But first let's look at some examples.

Figure 10.2 shows the size of simulated tumors as a function of age. The dashed line at 10 cm shows the range of ages for tumors at that size: the fastest-growing tumor gets there in 8 years; the slowest takes more than 35.

I am presenting results in terms of linear measurements, but the calculations are in terms of volume. To convert from one to the other, again, I use the volume of a sphere with the given diameter.

10.4 Implementation

Here is the kernel of the simulation:

```
def MakeSequence(iterator, v0=0.01, interval=0.67, vmax=Volume(20.0)):
    seq = v0,
    age = 0

    for rdt in iterator:
        age += interval
        final, seq = ExtendSequence(age, seq, rdt, interval)
        if final > vmax:
            break

    return seq
```

`iterator` is an iterator object that yields a series of random values from the CDF of growth rate. `MakeSequence` takes the iterator as an argument so we can run it with different random number generators, as you will see soon.

`v0` is the initial volume in mL. `interval` is the time step in years. `vmax` is the final volume corresponding to a linear measurement of 20 cm.

`Volume` converts from linear measurement in cm to volume in mL, based on the simplification that the tumor is a sphere:

```
def Volume(diameter, factor=4*math.pi/3):
    return factor * (diameter/2.0)**3
```

`ExtendSequence` computes the volume of the tumor at the end of the interval.

```
def ExtendSequence(age, seq, rdt, interval):
    initial = seq[-1]
    doublings = rdt * interval
    final = initial * 2**doublings
    new_seq = seq + (final,)
    cache.Add(age, new_seq, rdt)

    return final, new_seq
```

`age` is the age of the tumor at the end of the interval. `seq` is a tuple that contains the volumes so far. `rdt` is the growth rate during the interval, in doublings per year. `interval` is the size of the time step in years.

The return values are `final`, the volume of the tumor at the end of the interval, and `new_seq` a new tuple containing the volumes in `seq` plus the new volume `final`.

I am using a tuple so I can store it in the cache without worrying about it being modified later. The purpose of the cache is to record the size and age of each tumor at the end of each interval.

10.5 Caching the joint distribution

Here's how the cache works.

```
class Cache(object):

    def __init__(self):
        self.joint = thinkbayes.Pmf()
```

`joint` is a Pmf that records frequency of each age-size pair, which approximates the joint distribution of age and size.

At the end of each simulated interval, `ExtendSequence` calls `Cache.Add`:

```
    def Add(self, age, seq):
        final = seq[-1]
        cm = Diameter(final)
        bucket = round(CmToBucket(cm))
        self.joint.Incr((age, bucket))
```

Again, `age` is the age of the tumor, and `seq` is the sequence of volumes so far.

Before adding the new data to the joint distribution, we use `Diameter` to convert from volume to diameter in centimeters, and `CmToBucket` to convert from centimeters to a discrete bucket number:

```
def CmToBucket(x, factor=10):
    return factor * math.log(x)
```

The buckets are equally spaced on a log scale. I chose the factor 10 to yield a reasonable number of buckets. For example, 1 cm maps to bucket 0; 10 cm maps to bucket 23.

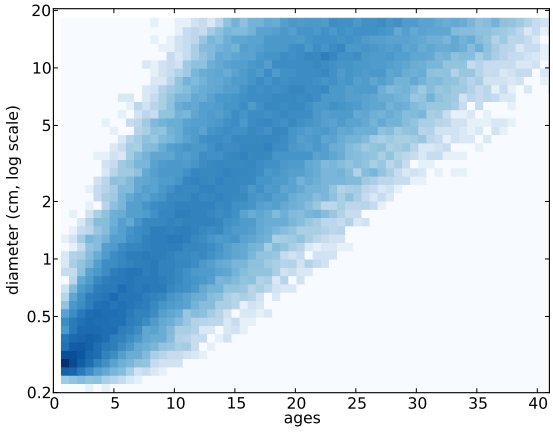

Figure 10.3: Joint distribution of age and tumor size.

After running the simulations, we can plot the joint distribution as a pseudocolor plot, where each cell represents the number of tumors observed at a given size-age pair.

Figure 10.3 shows the joint distribution after 1000 simulations. As expected, the size of tumors typically increases with age.

10.6 Conditional distributions

By taking a vertical slice from the joint distribution, we can get the distribution of sizes for any given age. By taking a horizontal slice, we can get the distribution of ages conditioned on size.

Here's the code that reads the joint distribution and builds the conditional distribution for a given size bucket.

```
def ConditionalCdf(size_bucket):
    pmf = thinkbayes.Pmf(name=name)
    for val, prob in cache.GetItems():
        age, bucket = val
        if bucket == size_bucket:
            pmf.Set(age, prob)

    pmf.Normalize()
    cdf = thinkbayes.MakeCdfFromPmf(pmf)
    return cdf
```

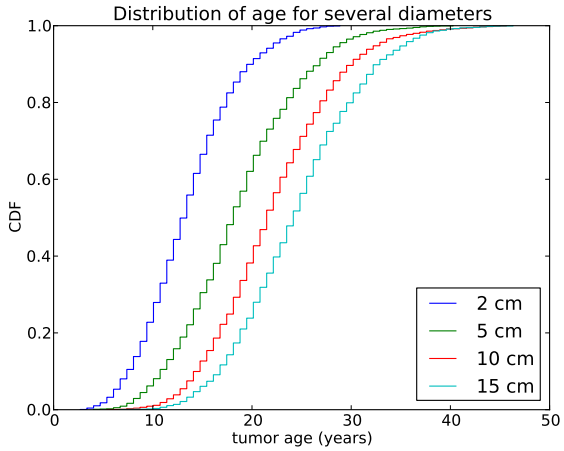

Figure 10.4: Distributions of age, conditioned on size.

`size_bucket` is the integer bucket number corresponding to a particular size. The return value is the CDF of age, conditioned on the given size.

Figure 10.4 shows several of these CDFs, for a range of sizes. To summarize these distributions, we can compute percentiles as a function of size.

```
percentiles = [95, 75, 50, 25, 5]

for bucket in cache.GetBuckets():
    cdf = ConditionalCdf(bucket)
    ps = [cdf.Percentile(p) for p in percentiles]
```

Figure 10.5 shows these percentiles for each size bucket. The data points are computed from the estimated joint distribution. There is some variability caused by making time and size discrete, so I also show a least squares fit for each sequence of percentiles.

10.7 Serial Correlation

The results so far are based on a number of modeling decisions; let's review them and consider which ones are the most likely sources of error:

- To convert from linear measure to volume, we assume that tumors are approximately spherical. This assumption is probably fine for tumors up to a few centimeters, but not for very large tumors.
- The distribution of growth rates in the simulations are based on a continuous model we chose to fit the data reported by Zhang et al, which

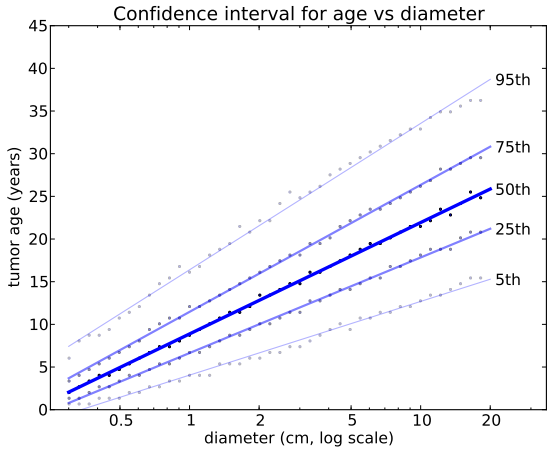

Figure 10.5: Percentiles of tumor age as a function of size.

is based on 53 patients. The fit is only approximate and, more importantly, a larger sample would yield a different distribution.

- The growth model does not take into account tumor subtype or grade, which is consistent with the conclusion of Zhang et al: "Growth rates in renal tumors of different sizes, subtypes and grades represent a wide range and overlap substantially." But again, with a larger sample, difference might become apparent.

- The distribution of growth rate does not depend on the size of the tumor. This assumption is not realistic for very small and very large tumors, whose growth is limited by blood supply.

  But tumors observed by Zhang et al ranged from 1 to 12 cm, and they found no statistically significant relationship between size and growth rate. So if there is a relationship, it is likely to be weak, at least in this size range.

- In the simulations, growth rate during each interval is independent of previous growth rates. It is plausible that, in reality, tumors that have grown quickly in the past are more likely to grow quickly. In other words, there is probably a serial correlation in growth rate.

Of these, the first and last seem the most problematic. I'll investigate serial correlation first, then come back to spherical geometry.

To simulate correlated growth, I wrote a generator that yields a correlated series from a given CDF. Here's how the algorithm works:

1. Generate correlated variates from a Gaussian distribution. This is easy to do because we can compute the distribution of the next value conditioned on the previous value.

2. Transform each value to its cumulative probability, using the Gaussian CDF.

3. Transform each cumulative probability to a random variate by inverting the given CDF.

Here's what that looks like in code:

```
def CorrelatedGenerator(cdf, rho):
    x = random.gauss(0, 1)
    yield Transform(x)

    sigma = math.sqrt(1 - rho**2);
    while True:
        x = random.gauss(x * rho, sigma)
        yield Transform(x)
```

`cdf` is the desired CDF; `rho` is the desired correlation.

The first value is Gaussian with mean 0 and standard deviation 1.

For subsequent values, the mean and standard deviation depend on the previous value. Given the previous x, the mean of the next value is ρx, and the variance is $1 - \rho^2$.

`Transform` is the function that maps from a Gaussian variate to a variate with the given CDF.

```
    def Transform(x):
        p = thinkbayes.GaussianCdf(x)
        y = cdf.Value(p)
        return y
```

`GaussianCdf` computes the CDF of the standard Gaussian distribution at `x`, returning a cumulative probability. `Cdf.Value` maps from a cumulative probability to the corresponding value in `cdf`.

Depending on the shape of `cdf`, information can be lost in transformation, so the actual correlation might be lower than the target. For example, when I generate 10000 values from the distribution of growth rates with target correlation 0.4, the actual correlation is 0.37.

Remember that `MakeSequence` takes an iterator as an argument. That interface allows it to work with different generators:

Serial Correlation	Diameter (cm)	Percentiles of age				
		5th	25th	50th	75th	95th
0.0	6.0	10.7	15.4	19.5	23.5	30.2
0.4	6.0	9.4	15.4	20.8	26.2	36.9

Table 10.1: Percentiles of tumor age conditioned on size.

```
iterator = UncorrelatedGenerator(cdf, rho)
seq1 = MakeSequence(iterator)

iterator = CorrelatedGenerator(cdf, rho)
seq2 = MakeSequence(iterator)
```

Now we can see what effect serial correlation has on the results; the following table shows percentiles of age for a 6 cm tumor, using the uncorrelated generator and a correlated generator with target $\rho = 0.4$.

Correlation makes the fastest growing tumors faster and the slowest slower, so the range of ages is wider. The difference is modest for low percentiles, but for the 95th percentile it is more than 6 years. To compute these percentiles precisely, we would need a better estimate of the actual serial correlation.

However, this model is sufficient to answer the question we started with: given a tumor with a linear dimension of 15.5 cm, what is the probability that it formed more than 8 years ago?

Here's the code:

```
def ProbOlder(cm, age):
    bucket = CmToBucket(cm)
    cdf = ConditionalCdf(bucket)
    p = cdf.Prob(age)
    return 1-p
```

`cm` is the size of the tumor; `age` is the age threshold in years. With no serial correlation, the probability that a 15.5 cm tumor is older than 8 years is 0.999, or almost certain. With correlation 0.4, faster-growing tumors are more likely, but the probability is still 0.995. Even with correlation 0.8, the probability is 0.978.

Another likely source of error is the assumption that tumors are approximately spherical. For a tumor with linear dimensions 15.5 x 15 cm, this assumption is probably not valid. If, as seems likely, it were relatively flat, it might have the same volume as a 6 cm sphere. But with this smaller volume and correlation 0.8, the probability of age greater than 8 is still 95%.

So even taking into account modeling errors, it is unlikely that such a large tumor could have formed less than 8 years prior to the date of diagnosis.

10.8 Where's Bayes?

Well, we got through a whole chapter without using Bayes's Theorem or the `Suite` class that encapsulates Bayesian updates. What happened?

One way to think about Bayes's Theorem is as an algorithm for inverting conditional probabilities. Given $\mathrm{p}(B|A)$, we can compute $\mathrm{p}(A|B)$, provided we know $\mathrm{p}(A)$ and $\mathrm{p}(B)$. Of course this algorithm is only useful if, for some reason, it is easier to compute $\mathrm{p}(B|A)$ than $\mathrm{p}(A|B)$.

In this example, it is. By running simulations, we can estimate the distribution of size conditioned on age, or $\mathrm{p}(size|age)$. But it is harder to get the distribution of age conditioned on size, or $\mathrm{p}(age|size)$. So this seems like a perfect opportunity to use Bayes's Theorem.

The reason I didn't is computational efficiency. To estimate $\mathrm{p}(size|age)$ for any given size, you have to run a lot of simulations. Along the way, you end up computing $\mathrm{p}(size|age)$ for a lot of sizes. In fact, you end up computing the entire joint distribution of size and age, $\mathrm{p}(size, age)$.

And once you have the joint distribution, you don't really need Bayes's Theorem, you can extract $\mathrm{p}(age|size)$ by taking slices from the joint distribution, as demonstrated in `ConditionalCdf`.

So we side-stepped Bayes, but he was with us in spirit.

Chapter 11

A hierarchical model

11.1 Belly button bacteria

Belly Button Biodiversity 2.0 (BBB2) is a nation-wide citizen science project with the goal of identifying bacterial species that can be found in human navels (`http://bbdata.yourwildlife.org`). The project might seem whimsical, but it is part of an increasing interest in the human microbiome, the set of microorganisms that live on human skin and other surfaces that contact the environment.

In their pilot study, BBB2 researchers collected swabs from the navels of 60 volunteers, used multiplex pyrosequencing to extract and sequence fragments of 16S rDNA, then identified the species or genus the fragments came from. Each identified fragment is called a "read."

We can use these data to answer several related questions:

- Based on the number of species we have observed, can we estimate the total number of species in the environment?
- Can we estimate the prevalence of each species; that is, the fraction of the total population belonging to each species?
- If we are planning to collect additional samples, can we predict how many new species we are likely to discover?
- How many additional reads are needed to increase the fraction of observed species to a given threshold?

These questions make up what is called the "unseen species problem."

11.2 Lions and tigers and bears

I'll start with a simplified version of the problem where we know that there are exactly three species. Let's call them lions, tigers and bears. Suppose we visit a wild animal preserve and see 3 lions, 2 tigers and one bear.

If we have an equal chance of observing any animal in the preserve then the number of each species we see is governed by the multinomial distribution. If the prevalence of lions and tigers and bears is `p_lion` and `p_tiger` and `p_bear`, the likelihood of seeing 3 lions, 2 tigers and one bear is

```
p_lion**3 * p_tiger**2 * p_bear**1
```

An approach that is tempting, but not correct, is to use beta distributions to describe the prevalence of each species separately. For example, we saw 3 lions and 3 non-lions; if we think of that as 3 "heads" and 3 "tails," then the posterior distribution of `p_lion` is:

```
beta = thinkbayes.Beta()
beta.Update((3, 3))
print beta.MaximumLikelihood()
```

The maximum likelihood estimate for `p_lion` is the observed rate, 50%. Similarly the MLEs for `p_tiger` and `p_bear` are 33% and 17%.

But there are two problems:

- We have implicitly used a prior for each species that is uniform from 0 to 1, but since we know that there are three species, that prior is not correct. The right prior should have a mean of 1/3, and there should be zero likelihood that any species has a prevalence of 100%.
- The distributions for each species are not independent, because the prevalences have to add up to 1. To capture this dependence, we need a joint distribution for the three prevalences.

We can use a Dirichlet distribution to solve both of these problems (see `http://en.wikipedia.org/wiki/Dirichlet_distribution`). In the same way we use the beta distribution to describe the distribution of `x`, we can use a Dirichlet distribution to describe the joint distribution of `p_lion` and `p_tiger` and `p_bear`.

The Dirichlet distribution is the multi-dimensional generalization of the beta distribution. Instead of two possible outcomes, like heads and tails, the Dirichlet distribution handles any number of outcomes: in this example, three species.

If there are n outcomes, the Dirichlet distribution is described by n parameters, written α_i.

Here's the definition, from thinkbayes.py, of a class that represents a Dirichlet distribution:

```
class Dirichlet(object):

    def __init__(self, n):
        self.n = n
        self.params = numpy.ones(n, dtype=numpy.int)
```

n is the number of dimensions; initially the parameters are all 1. I use a numpy array to store the parameters so I can take advantage of array operations.

Given a Dirichlet distribution, the marginal distribution for each prevalence is a beta distribution, which we can compute like this:

```
    def MarginalBeta(self, i):
        alpha0 = self.params.sum()
        alpha = self.params[i]
        return Beta(alpha, alpha0-alpha)
```

i is the index of the marginal distribution we want. alpha0 is the sum of the parameters; alpha is the parameter for the given species.

In the example, the prior marginal distribution for each species is Beta(1, 2). We can compute the prior means like this:

```
    dirichlet = thinkbayes.Dirichlet(3)
    for i in range(3):
        beta = dirichlet.MarginalBeta(i)
        print beta.Mean()
```

As expected, the mean prior for each species is 1/3.

To update the Dirichlet distribution, we add the number of observations to each parameter, like this:

```
    def Update(self, data):
        m = len(data)
        self.params[:m] += data
```

Here data is a sequence of counts in the same order as params, so in this example, it should be the number of lions and tigers and bears.

But the data can be shorter than params; in that case there are some hypothetical species that have not been observed.

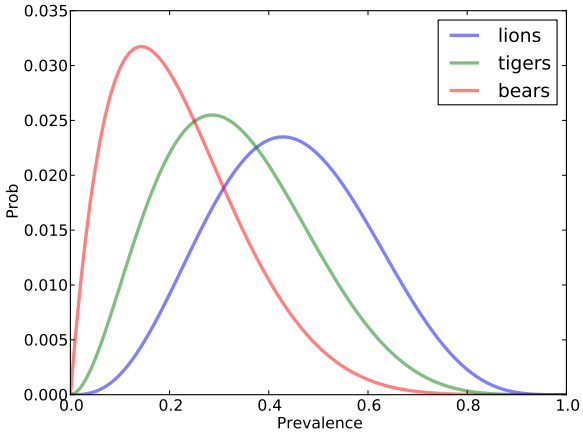

Figure 11.1: Distribution of prevalences for three species.

Here's code that updates the Dirichlet with the observed data, and plots the posterior marginal distributions.

```
data = [3, 2, 1]
dirichlet.Update(data)

for i in range(3):
    beta = dirichlet.MarginalBeta(i)
    pmf = beta.MakePmf()
    print pmf.MaximumLikelihood()
```

Figure 11.1 shows the results. The posterior mean prevalences are 44%, 33% and 22%.

11.3 A hierarchical model

We have solved a simplified version of the problem: if we know how many species there are, we can estimate the prevalence of each.

Now let's get back to the original problem, estimating the total number of species. To solve this problem I'll define a metasuite, which is a Suite that contains other Suites as hypotheses. In this case, the top-level Suite contains hypotheses about the number of species; the bottom level contains hypotheses about prevalences. A multi-level model like this is called "hierarchical."

Here's the class definition:

```
class Species(thinkbayes.Suite):

    def __init__(self, ns):
        hypos = [thinkbayes.Dirichlet(n) for n in ns]
        thinkbayes.Suite.__init__(self, hypos)
```

`__init__` takes a list of possible values for `n` and makes a list of Dirichlet objects.

Here's the code that creates the top-level suite:

```
    ns = range(3, 30)
    suite = Species(ns)
```

`ns` is the list of possible values for `n`. We have seen 3 species, so there have to be at least that many. I chose an upper bound that seemed reasonable, but we will have to check later that the probability of exceeding this bound is low. And at least initially we assume that any value in this range is equally likely.

To update a hierarchical model, you have to update all levels. Sometimes it is necessary or more efficient to update the bottom level first and work up. In this case it doesn't matter, so I update the top level first:

```
#class Species

    def Update(self, data):
        thinkbayes.Suite.Update(self, data)
        for hypo in self.Values():
            hypo.Update(data)
```

`Species.Update` invokes `Update` in the parent class, then loops through the sub-hypotheses and updates them.

Now all we need is a likelihood function. As usual, `Likelihood` gets a hypothesis and a dataset as arguments:

```
# class Species

    def Likelihood(self, hypo, data):
        dirichlet = hypo
        like = 0
        for i in range(1000):
            like += dirichlet.Likelihood(data)

        return like
```

`hypo` is a Dirichlet object; `data` is a sequence of observed counts. `Species.Likelihood` calls `Dirichlet.Likelihood` 1000 times and returns the total.

Why do we have to call it 1000 times? Because `Dirichlet.Likelihood` doesn't actually compute the likelihood of the data under the whole Dirichlet distribution. Instead, it draws one sample from the hypothetical distribution and computes the likelihood of the data under the sampled set of prevalences.

Here's what it looks like:

```
# class Dirichlet

    def Likelihood(self, data):
        m = len(data)
        if self.n < m:
            return 0

        x = data
        p = self.Random()
        q = p[:m]**x
        return q.prod()
```

The length of `data` is the number of species observed. If we see more species than we thought existed, the likelihood is 0.

Otherwise we select a random set of prevalences, `p`, and compute the multinomial PDF, which is

$$c(x) \prod_i p_i^{x_i}$$

p_i is the prevalence of the ith species, and x_i is the observed number. The first term, $c(x)$, is the multinomial coefficient; I left it out of the computation because it is a multiplicative factor that depends only on the data, not the hypothesis, so it gets normalized away (see `http://en.wikipedia.org/wiki/Multinomial_distribution`).

Also, I truncated `p` at `m`, which is the number of observed species. For the unseen species, x_i is 0, so $p_i^{x_i}$ is 1, so we can leave them out of the product.

11.4 Random sampling

There are two ways to generate a random sample from a Dirichlet distribution. One is to use the marginal beta distributions, but in that case

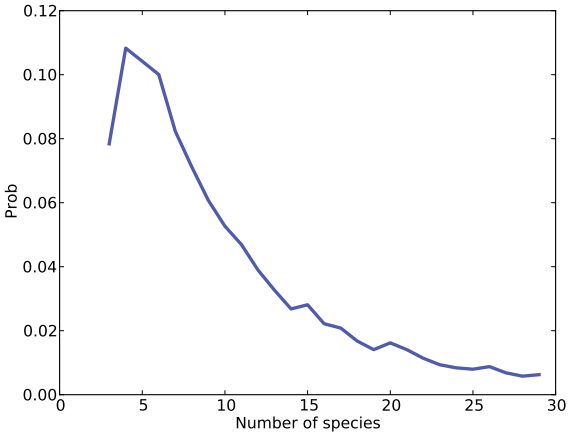

Figure 11.2: Distribution of prevalences for three species.

you have to select one at a time and scale the rest so they add up to 1 (see `http://en.wikipedia.org/wiki/Dirichlet_distribution#Random_number_generation`).

A less obvious, but faster, way is to select values from `n` gamma distributions, then normalize by dividing through by the total. Here's the code:

```
# class Dirichlet

    def Random(self):
        p = numpy.random.gamma(self.params)
        return p / p.sum()
```

Now we're ready to look at some results. Here is the code that updates the top-level suite and extracts the posterior PMF of `n`:

```
data = [3, 2, 1]
suite.Update(data)
pmf = suite.DistOfN()
```

To get the posterior distribution of `n`, `DistOfN` iterates through the top-level hypotheses:

```
def DistOfN(self):
    pmf = thinkbayes.Pmf()
    for hypo, prob in self.Items():
        pmf.Set(hypo.n, prob)
    return pmf
```

Figure 11.2 shows the result. The most likely value is 4. Values from 3 to 8 are all likely; after that the probabilities drop off quickly. The probability

that there are 29 species is low enough to be negligible; if we chose a higher bound, we would get the same result.

But remember that we started with a uniform prior for `n`. If we have background information about the number of species in the environment, we might choose a different prior.

11.5 Optimization

I have to admit that I am proud of this example. The unseen species problem is not easy, and I think this solution is simple and clear, and takes surprisingly few lines of code (about 50 so far).

The only problem is that it is slow. It's good enough for the example with only 3 observed species, but not good enough for the belly button data, with more than 100 species in some samples.

The next few sections present a series of optimizations we need to make this solution scale. Before we get into the details, here's a road map.

- The first step is to recognize that if we update the Dirichlet distributions with the same data, the first `m` parameters are the same for all of them. The only difference is the number of hypothetical unseen species. So we don't really need `n` Dirichlet objects; we can store the parameters in the top level of the hierarchy. `Species2` implements this optimization.

- `Species2` also uses the same set of random values for all of the hypotheses. This saves time generating random values, but it has a second benefit that turns out to be more important: by giving all hypothesis the same selection from the sample space, we make the comparison between the hypotheses more fair, so it takes fewer iterations to converge.

- But there is still a major performance problem. As the number of observed species increases, the array of random prevalences gets bigger, and the chance of choosing one that is approximately right becomes small. So the vast majority of iterations yield small likelihoods that don't contribute much to the total, and don't discriminate between hypotheses.

  The solution is to do the updates one species at a time. `Species4` is a simple implementation of this strategy using Dirichlet objects to represent the sub-hypotheses.

- Finally, `Species5` combines the sub-hypothesis into the top level and uses `numpy` array operations to speed things up.

If you are not interested in the details, feel free to skip to Section 11.9 where we look at some results from the belly button data.

11.6 Collapsing the hierarchy

As I said, all of the bottom-level Dirichlet distributions are updated with the same data, so the first `m` parameters are the same for all of them.

Since the bottom-level Dirichlet distributions all contain the same information, we can eliminate them and merge the parameters into the top-level suite. `Species2` implements this optimization:

```
class Species2(object):

    def __init__(self, ns):
        self.ns = ns
        self.probs = numpy.ones(len(ns), dtype=numpy.double)
        self.params = numpy.ones(self.high, dtype=numpy.int)
```

`ns` is the list of hypothetical values for `n`; `probs` is the list of corresponding probabilities. And `params` is the sequence of Dirichlet parameters, initially all 1.

Here's the `Update` function that updates both levels of the hierarchy: first the probability for each value of `n`, then the Dirichlet parameters:

```
# class Species2

    def Update(self, data):
        like = numpy.zeros(len(self.ns), dtype=numpy.double)
        for i in range(1000):
            like += self.SampleLikelihood(data)

        self.probs *= like
        self.probs /= self.probs.sum()

        m = len(data)
        self.params[:m] += data
```

`SampleLikelihood` returns an array of likelihoods, one for each value of `n`. `like` accumulates the total likelihood for 1000 samples. `self.probs` is

multiplied by the total likelihood, then normalized. Finally, updating the parameters is identical to what we saw in `Dirichlet.Update`.

Now let's look at `SampleLikelihood`. There are two opportunities for optimization here:

- When the hypothetical number of species, `n`, exceeds the observed number, `m`, we only need the first `m` terms of the multinomial PDF; the rest are 1.
- If the number of species is large, the likelihood of the data might be too small for floating-point. So it is safer to compute log-likelihoods. For the multinomial PDF, it is also faster.

Again, the multinomial PDF is

$$c(x) \prod_i p_i^{x_i}$$

So the log-likelihood is

$$log(c(x)) + \sum_i \log(p_i) * x_i$$

which is fast and easy to compute. Here's the code:

```
# class Species2

    def SampleLikelihood(self, data):
        gammas = numpy.random.gamma(self.params)

        m = len(data)
        row = gammas[:m]
        col = numpy.cumsum(gammas)

        log_likes = []
        for n in self.ns:
            ps = row / col[n-1]
            terms = numpy.log(ps) * data
            log_like = terms.sum()
            log_likes.append(log_like)

        log_likes -= numpy.max(log_likes)
        likes = numpy.exp(log_likes)
```

```
        coefs = [thinkbayes.BinomialCoef(n, m) for n in self.ns]
        likes *= coefs

        return likes
```

`gammas` is an array of gamma variates; its length is the largest hypothetical value of `n`. `row` is just the first `m` elements of `gammas`; since these are the only elements that depend on the data, they are the only ones we need.

For each value of `n` we need to divide `row` by the total of the first `n` values from `gamma`. `cumsum` computes these cumulative sums and stores them in `col`.

The loop iterates through the values of `n` and accumulates a list of log-likelihoods.

Inside the loop, `ps` contains the row of probabilities, normalized with the appropriate cumulative sum. `terms` contains the terms of the summation, $\log(p_i)x_i$, and `log_like` contains their sum.

After the loop, we want to convert the log-likelihoods to linear likelihoods, but first it's a good idea to shift them so the largest log-likelihood is 0; that way the linear likelihoods are not too small.

Finally, before we return the likelihood, we have to apply a correction factor, which is the number of ways we could have observed these `m` species, if the total number of species is `n`. The binomial coefficient is the fancy name for "n choose m", which is written $\binom{n}{m}$.

As often happens, the optimized version is less readable and more error-prone than the original. But that's one reason I think it is a good idea to start with the simple version; we can use it for regression testing. I plotted results from both versions and confirmed that they are approximately equal, and that they converge as the sample size increases.

11.7 One more problem

There's more we could do to optimize this version, but there's a problem we need to work on first. As the number of observed species increases, this version gets noisier and takes more iterations to converge on a good answer.

The problem is that if the probabilities we choose from the Dirichlet distribution, the `ps`, are not at least approximately right, the likelihood of the

observed data is close to zero and almost equally bad for all values of `n`. So most iterations don't provide any useful contribution to the total likelihood. And as the number of observed species, `m`, gets large, the probability of choosing `ps` with non-legigible likelihood gets small. Really small.

Fortunately, there is a solution. Remember that if you observe a set of data, you can update the prior distribution with the entire dataset, or you can break it up into a series of updates with subsets of the data, and the result is the same either way.

For this example, the key is to perform the updates one species at a time. That way when we generate a random set of `ps`, only one of them affects the computed likelihood, so the chance of choosing a good one is much better.

Here's a new version that updates one species at a time:

```
class Species4(Species):

    def Update(self, data):
        m = len(data)

        for i in range(m):
            one = numpy.zeros(i+1)
            one[i] = data[i]
            Species.Update(self, one)
```

This version inherits `__init__` from `Species`, so it represents the hypotheses as a list of Dirichlet objects (unlike `Species2`.

`Update` loops through the observed species and makes an array, `one`, with all zeros and one species count. Then is calls `Update` in the parent class, which actually computes the likelihoods and updates the sub-hypotheses.

So in the running example, we do three updates. The first is something like "I have seen three lions." The second is "I have see two tigers and no additional lions." And the third is "I have seen one bear and no more lions and tigers."

So here's the new version of `Likelihood`:

```
# class Species4

    def Likelihood(self, hypo, data):
        dirichlet = hypo
        like = 0
```

```
        for i in range(self.iterations):
            like += dirichlet.Likelihood(data)

        # correct for the number of unseen species the new one
        # could have been
        m = len(data)
        num_unseen = dirichlet.n - m + 1
        like *= num_unseen

        return like
```

This is almost the same as `Species.Likelihood`. The difference is the factor, `num_unseen`. This correction is necessary because each time we see a species for the first time, we have to consider that there were some number of other unseen species that we might have seen. For larger values of `n` there are more unseen species that we could have seen, which increases the likelihood of the data.

This is a subtle point and I have to admit that I did not get it right the first time. But again I was able to validate this version by comparing it to the previous versions.

11.8 We're not done yet

Performing the updates one species at a time solves one problem, but it creates another. Each update takes time proportional to nm, so if we do m updates, the algorithm is $\mathcal{O}(nm^2)$.

But we can speed things up using the same trick we used in Section 11.6: we'll get rid of the Dirichlet objects and collapse the two levels of the hierarchy into a single object. So here's yet another version of `Species`:

```
class Species5(Species2):

    def Update(self, data):
        m = len(data)
        for i in range(m):
            self.UpdateOne(i+1, data[i])
            self.params[i] += data[i]
```

This version inherits `__init__` from `Species2`, so it uses `ns` and `probs` to represent the distribution of `n`, and `params` to represent the parameters of the Dirichlet distribution.

`Update` is similar to what we saw in the previous section. It loops through the observed species and calls `UpdateOne`. One difference is that `UpdateOne` doesn't get the whole list of species counts; it just gets the index (which species was observed) and the count (how many of them there were).

Now here's `UpdateOne`:

```
# class Species5

    def UpdateOne(self, m, count):
        likes = numpy.zeros(len(self.ns), dtype=numpy.double)
        for i in range(self.iterations):
            likes += self.SampleLikelihood(m, count)

        unseen_species = [n-m+1 for n in self.ns]
        likes *= unseen_species

        self.probs *= likes
        self.probs /= self.probs.sum()
```

This function is similar to `Species2.Update`, with two changes:

- The interface is different. Instead of the whole dataset, we get `m`, the number of species we have seen, and `count`, how many of that species we've seen.
- We have to apply a correction factor for the number of unseen species, as in `Species4.Likelihood`. The difference here is that we update all of the likelihoods at once with array multiplication.

Finally, here's `SampleLikelihood`:

```
# class Species5

    def SampleLikelihood(self, m, count):
        gammas = numpy.random.gamma(self.params)

        sums = numpy.cumsum(gammas)[self.ns[0]-1:]

        ps = gammas[m-1] / sums
        log_likes = numpy.log(ps) * count

        log_likes -= numpy.max(log_likes)
        likes = numpy.exp(log_likes)

        return likes
```

This is similar to `Species2.SampleLikelihood`; the difference is that each update only includes a single species, so we don't need a loop: `log_likes` only has a single term.

So the runtime of this function is linear in `n`, but does not depend on `m`. `Update` is linear in `m`, so the whole update process is $\mathcal{O}(()nm)$. And the number of iterations we need to get an accurate result is usually small.

11.9 The belly button data

That's enough about lions and tigers and bears. Let's get back to belly buttons. To get a sense of what the data look like, consider subject B1242, whose sample of 400 reads yielded 61 species with the following counts:

```
92, 53, 47, 38, 15, 14, 12, 10, 8, 7, 7, 5, 5,
4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

There are a few dominant species that make up a substantial fraction of the whole, but many species that yielded only a single read. The number of these "singletons" suggests that there are likely to be at least a few unseen species.

In the example with lions and tigers, we assume that each animal in the preserve is equally likely to be observed. Similarly, for the belly button data, we assume that each bacterium is equally likely to yield a read.

In reality, it is possible that each step in the data-collection process might introduce consistent biases. Some species might be more likely to be picked up by a swab, or to yield identifiable amplicons. So when we talk about the prevalence of each species, we should remember this source of error.

I should also acknowledge that I am using the term "species" loosely. First, bacterial species are not well-defined. Second, some reads identify a particular species, others only identify a genus. To be more precise, I should say "operational taxonomic unit, or OTU.

Now let's process some of the belly button data. I defined a class called `Subject` to represent information about each subject in the study:

```
class Subject(object):

    def __init__(self, code):
```

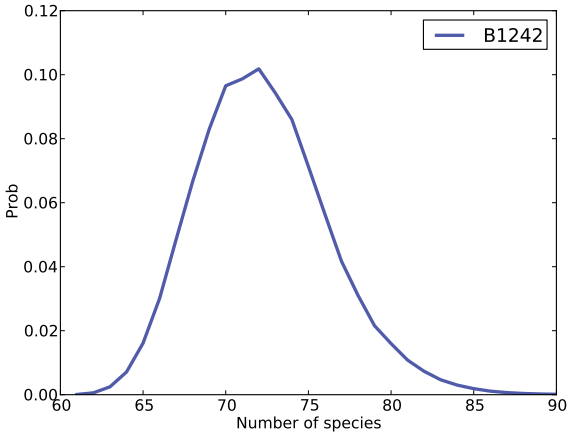

Figure 11.3: Distribution of `n` for subject B1242.

```
        self.code = code
        self.species = []
```

Each subject has a string code, like "B1242", and a list of (count, species name) pairs, sorted in increasing order by count. `Subject` provides several methods to make it easy to these counts and species names. You can see the details in `http://thinkbayes.com/species.py`.

In addition, `Subject.Process` creates a suite, specifically a suite of type `Species5`, which represents the distribution of `n` and the prevalences after processing the data.

It also provides `PlotDistOfN`, which plots the posterior distribution of `n`. Figure 11.3 shows this distribution for subject B1242. The probability that there are exactly 61 species, and no unseen species, is nearly zero. The most likely value is 72, with 90% credible interval 66 to 79. At the high end, it is unlikely that there are as many as 87 species.

Next we compute the posterior distribution of prevalence for each species. `Species2` provides `DistOfPrevalence`:

```
# class Species2

    def DistOfPrevalence(self, index):
        pmfs = thinkbayes.Pmf()

        for n, prob in zip(self.ns, self.probs):
            beta = self.MarginalBeta(n, index)
            pmf = beta.MakePmf()
```

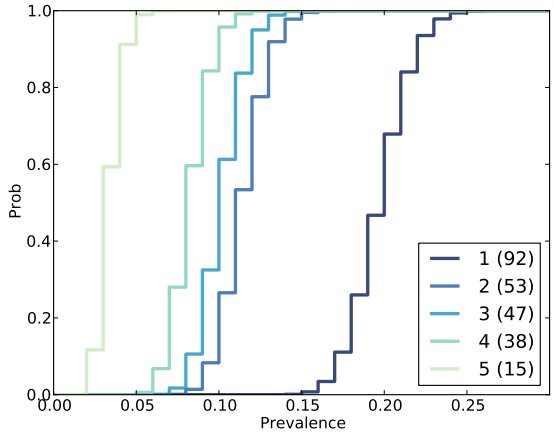

Figure 11.4: Distribution of prevalences for subject B1242.

```
    pmfs.Set(pmf, prob)

mix = thinkbayes.MakeMixture(pmfs)
return pmfs, mix
```

`index` indicates which species we want. For each value of `n`, we have a different posterior distribution of prevalence.

So the loop iterates through the possible values of `n` and their probabilities. For each value of `n` it gets a Beta object representing the marginal distribution for the indicated species. Remember that Beta objects contain the parameters `alpha` and `beta`; they don't have values and probabilities like a Pmf, but they provide `MakePmf` which generates a discrete approximation to the continuous beta distribution.

`pmfs` is a MetaPmf that contains the distributions of prevalence, conditioned on `n`. `MakeMixture` combines the MetaPmf into `mix`, which combines the conditional distributions into the answer, a single distribution of prevalence.

Figure 11.4 shows these distributions for the five species with the most reads. The most prevalent species accounts for 23% of the 400 reads, but since there are almost certainly unseen species, the most likely estimate for its prevalence is 20%, with 90% credible interval between 17% and 23%.

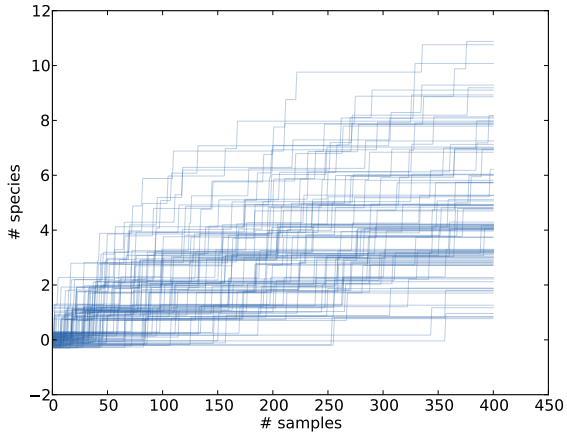

Figure 11.5: Simulated rarefaction curves for subject B1242.

11.10 Predictive distributions

I introduced the hidden species problem in the form of four related questions. We have answered the first two by computing the posterior distribution for `n` and the prevalence of each species.

The other two questions are:

- If we are planning to collect additional samples, can we predict how many new species we are likely to discover?
- How many additional reads are needed to increase the fraction of observed species to a given threshold?

To answer predictive questions like this we can use the posterior distributions to simulate possible future events and compute predictive distributions for the number of species, and fraction of the total, we are likely to see.

The kernel of these simulations looks like this:

1. Choose `n` from its posterior distribution.
2. Choose a prevalence for each species, including possible unseen species, using the Dirichlet distribution.
3. Generate a random sequence of future observations.

4. Compute the number of new species, num_new, as a function of the number of additional samples, k.

5. Repeat the previous steps and accumulate the joint distribution of num_new and k.

And here's the code. RunSimulation runs a single simulation:

```
# class Subject

    def RunSimulation(self, num_samples):
        m, seen = self.GetSeenSpecies()
        n, observations = self.GenerateObservations(num_samples)

        curve = []
        for k, obs in enumerate(observations):
            seen.add(obs)

            num_new = len(seen) - m
            curve.append((k+1, num_new))

        return curve
```

num_samples is the number of additional samples to simulate. m is the number of seen species, and seen is a set of strings with a unique name for each species. n is a random value from the posterior distribution, and observations is a random sequence of species names.

The result of RunSimulation is a "rarefaction curve", represented as a list of pairs with the number of samples and the number of new species seen.

Before we see the results, let's look at GetSeenSpecies and GenerateObservations.

```
#class Subject

    def GetSeenSpecies(self):
        names = self.GetNames()
        m = len(names)
        seen = set(SpeciesGenerator(names, m))
        return m, seen
```

GetNames returns the list of species names that appear in the data files, but for many subjects these names are not unique. So I use SpeciesGenerator to extend each name with a serial number:

```
def SpeciesGenerator(names, num):
    i = 0
    for name in names:
        yield '%s-%d' % (name, i)
        i += 1

    while i < num:
        yield 'unseen-%d' % i
        i += 1
```

Given a name like `Corynebacterium`, `SpeciesGenerator` yields `Corynebacterium-1`. When the list of names is exhausted, it yields names like `unseen-62`.

Here is `GenerateObservations`:

```
# class Subject

    def GenerateObservations(self, num_samples):
        n, prevalences = self.suite.Sample()

        names = self.GetNames()
        name_iter = SpeciesGenerator(names, n)

        d = dict(zip(name_iter, prevalences))
        cdf = thinkbayes.MakeCdfFromDict(d)
        observations = cdf.Sample(num_samples)

        return n, observations
```

Again, `num_samples` is the number of additional samples to generate. `n` and `prevalances` are samples from the posterior distribution.

`cdf` is a Cdf object that maps species names, including the unseen, to cumulative probabilities. Using a Cdf makes it efficient to generate a random sequence of species names.

Finally, here is `Species2.Sample`:

```
    def Sample(self):
        pmf = self.DistOfN()
        n = pmf.Random()
        prevalences = self.SampleConditional(n)
        return n, prevalences
```

And `SampleConditional`, which generates a sample of prevalances conditioned on `n`:

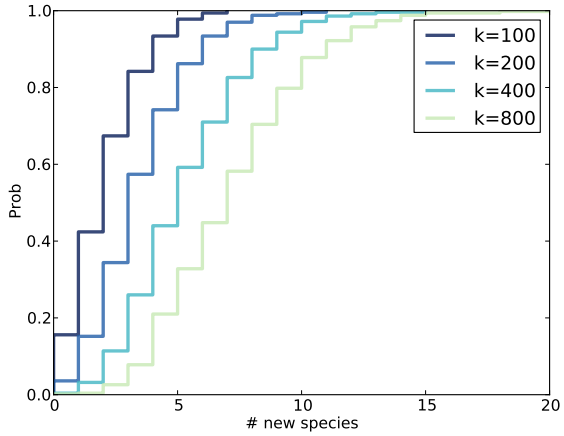

Figure 11.6: Distributions of the number of new species conditioned on the number of additional samples.

```
# class Species2

    def SampleConditional(self, n):
        params = self.params[:n]
        gammas = numpy.random.gamma(params)
        gammas /= gammas.sum()
        return gammas
```

We saw this algorithm for generating prevalences previously in `Species2.SampleLikelihood`.

Figure 11.5 shows 100 simulated rarefaction curves for subject B1242. I shifted each curve by a random offset so they would not all overlap. By inspection we can estimate that after 400 more samples we are likely to find $2 - -6$ new species.

11.11 Joint posterior

To be more precise, we can use the simulations to estimate the joint distribution of `num_new` and `k`, and from that we can get the distribution of `num_new` conditioned on any value of `k`.

```
# class Subject

    def MakeJointPredictive(self, curves):
        joint = thinkbayes.Joint()
```

```
        for curve in curves:
            for k, num_new in curve:
                joint.Incr((k, num_new))
        joint.Normalize()
        return joint
```

`MakeJointPredictive` makes a Joint object, which is a `Pmf` whose values are tuples.

`curves` is a list of rarefaction curves created by `RunSimulation`. Each curve contains a list of pairs of `k` and `num_new`.

The resulting joint distribution is a map from each pair to its probability of occuring. Given the joint distribution, we can get the distribution of `num_new` conditioned on `k`:

```
# class Joint

    def Conditional(self, i, j, val):
        pmf = Pmf()
        for vs, prob in self.Items():
            if vs[j] != val: continue
            pmf.Incr(vs[i], prob)

        pmf.Normalize()
        return pmf
```

`i` is the index of the variable whose distribution we want; `j` is the index of the conditional variables, and `val` is the value the `j`th variable has to have. You can think of this operation as taking vertical slices out of Figure 11.5.

`Subject.MakeConditionals` takes a list of `ks` and computes the conditional distribution of `num_new` for each `k`. The result is a list of Cdf objects.

```
# class Subject

    def MakeConditionals(self, curves, ks):
        joint = self.MakeJointPredictive(curves)

        cdfs = []
        for k in ks:
            pmf = joint.Conditional(1, 0, k)
            pmf.name = 'k=%d' % k
            cdf = thinkbayes.MakeCdfFromPmf(pmf)
            cdfs.append(cdf)
```

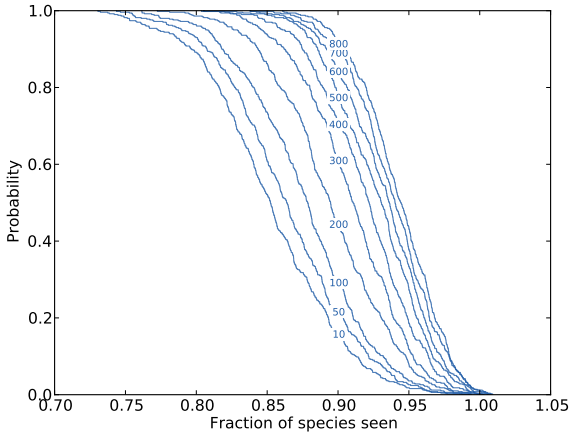

Figure 11.7: Complementary CDF of coverage for a range of additional samples.

```
        return cdfs
```

Figure 11.6 shows the results. After 100 samples, the median predicted number of new species is 2; the 90% credible interval is 0 to 5. After 800 samples, we expect to see 3 to 12 new species.

11.12 Coverage

The last question we want to answer is, "How many additional reads are needed to increase the fraction of observed species to a given threshold?"

To answer this question, we'll need a version of `RunSimulation` that computes the fraction of observed species rather than the number of new species.

```
# class Subject

    def RunSimulation(self, num_samples):
        m, seen = self.GetSeenSpecies()
        n, observations = self.GenerateObservations(num_samples)

        curve = []
        for k, obs in enumerate(observations):
            seen.add(obs)
```

```
            frac_seen = len(seen) / float(n)
            curve.append((k+1, frac_seen))

        return curve
```

Next we loop through each curve and make a dictionary, `d`, that maps from the number of additional samples, `k`, to a list of `fracs`; that is, a list of values for the coverage achieved after `k` samples.

```
    def MakeFracCdfs(self, curves):
        d = {}
        for curve in curves:
            for k, frac in curve:
                d.setdefault(k, []).append(frac)

        cdfs = {}
        for k, fracs in d.iteritems():
            cdf = thinkbayes.MakeCdfFromList(fracs)
            cdfs[k] = cdf

        return cdfs
```

Then for each value of `k` we make a Cdf of `fracs`; this Cdf represents the distribution of coverage after `k` samples.

Remember that the CDF tells you the probability of falling below a given threshold, so the *complementary* CDF tells you the probability of exceeding it. Figure 11.7 shows complementary CDFs for a range of values of `k`.

To read this figure, select the level of coverage you want to achieve along the x-axis. As an example, choose 90%.

Now you can read up the chart to find the probability of achieving 90% coverage after `k` samples. For example, with 300 samples, you have about a 60% of getting 90% coverage. With 700 samples, you have a 90% chance of getting 90% coverage.

With that, we have answered the four questions that make up the unseen species problem. This has been a longer chapter than most, but I think there is a lot to it. So if you made it this far, congratulations!

Chapter 12

Future chapters

Bayesian regression (hybrid version with resampling?)

Change point detection:

Deconvolution: Estimating round trip times