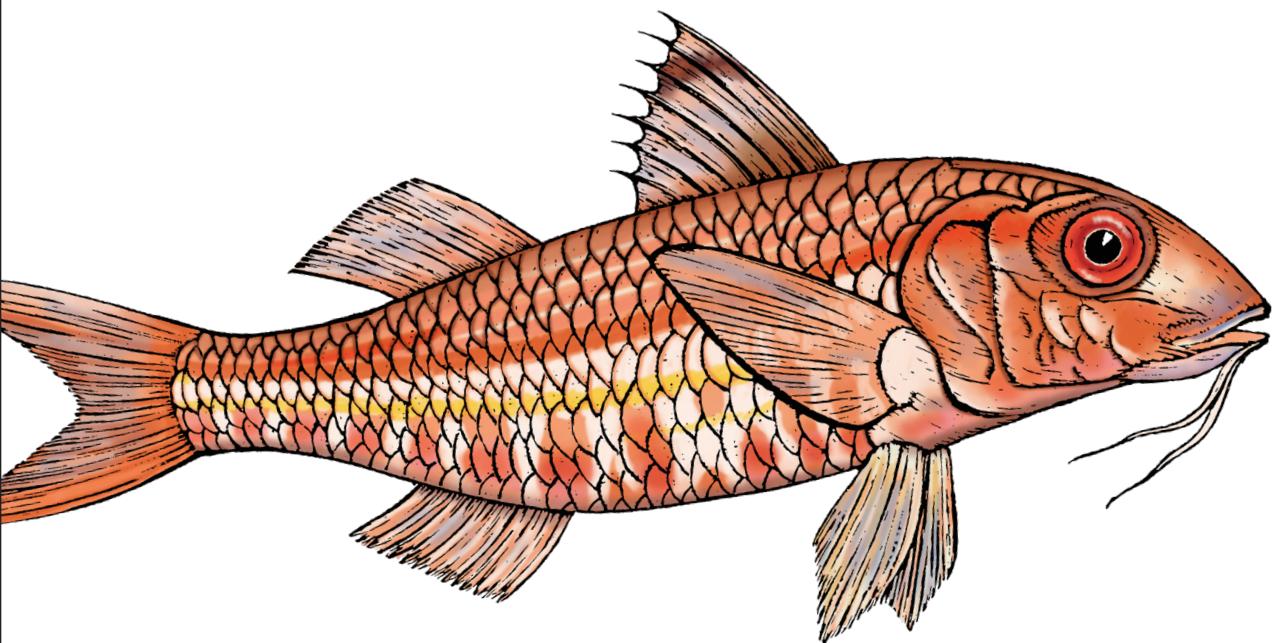


O'REILLY®

Second
Edition

Think Bayes

Bayesian Statistics in Python



Allen B. Downey

Think Bayes

If you know how to program, you're ready to tackle Bayesian statistics. With this book, you'll learn how to solve statistical problems with Python code instead of mathematical formulas, using discrete probability distributions rather than continuous mathematics. Once you get the math out of the way, the Bayesian fundamentals will become clearer and you'll begin to apply these techniques to real-world problems.

Bayesian statistical methods are becoming more common and more important, but there aren't many resources available to help beginners. Based on undergraduate classes taught by author Allen B. Downey, this book's computational approach helps you get a solid start.

- Use your programming skills to learn and understand Bayesian statistics
- Work with problems involving estimation, prediction, decision analysis, evidence, and Bayesian hypothesis testing
- Get started with simple examples, using coins, dice, and a bowl of cookies
- Learn computational methods for solving real-world problems

Allen B. Downey is a professor of computer science at Olin College of Engineering. He's taught at Wellesley College, Colby College, and UC Berkeley. Allen has a PhD in computer science from UC Berkeley and master's and bachelor's degrees from MIT. He's the author of *Think Python*, *Think Stats*, and *Think DSP* (all from O'Reilly), and a blog, *Probably Overthinking It*.

"Allen B. Downey once again turns a 200-year-old math theorem into an interesting book combining informative reading with practical examples. This is one of the best applied introductions to Bayes's theorem that you will find."

—Ravin Kumar
Data Scientist

"Allen takes an effective approach by curating unique examples and diverse applications. Bayesian ideas are no longer a niche interest, but a growing area for problem-solving."

—Thomas Nield
Founder of Nield Consulting Group and author of *Getting Started with SQL*

MATH

US \$49.99 CAN \$65.99

ISBN: 978-1-492-08946-9



Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

Think Bayes

Bayesian Statistics in Python

Allen B. Downey

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Think Bayes

by Allen B. Downey

Copyright © 2021 Allen B. Downey. All rights reserved.

Printed in the United States of America.

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

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

Acquisitions Editor: Jessica Haberman

Indexer: Sue Klefstad

Development Editor: Michele Cronin

Interior Designer: David Futato

Production Editor: Kristen Brown

Cover Designer: Karen Montgomery

Copyeditor: O'Reilly Production Services

Illustrator: Allen B. Downey

Proofreader: Stephanie English

September 2013: First Edition

May 2021: Second Edition

Revision History for the Second Edition

2021-05-18: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492089469> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Think Bayes*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

Think Bayes is available under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. The author maintains an online version at <https://greenteapress.com/wp/think-bayes>.

978-1-492-08946-9

[LSI]

Table of Contents

Preface.....	ix
1. Probability.....	1
Linda the Banker	1
Probability	2
Fraction of Bankers	3
The Probability Function	4
Political Views and Parties	4
Conjunction	5
Conditional Probability	6
Conditional Probability Is Not Commutative	7
Condition and Conjunction	8
Laws of Probability	8
Theorem 1	9
Theorem 2	10
Theorem 3	10
The Law of Total Probability	11
Summary	13
Exercises	14
2. Bayes's Theorem.....	17
The Cookie Problem	17
Diachronic Bayes	19
Bayes Tables	20
The Dice Problem	22
The Monty Hall Problem	23
Summary	25
Exercises	26

3. Distributions.....	29
Distributions	29
Probability Mass Functions	29
The Cookie Problem Revisited	32
101 Bowls	34
The Dice Problem	38
Updating Dice	39
Summary	40
Exercises	41
4. Estimating Proportions.....	43
The Euro Problem	43
The Binomial Distribution	44
Bayesian Estimation	47
Triangle Prior	49
The Binomial Likelihood Function	51
Bayesian Statistics	52
Summary	53
Exercises	54
5. Estimating Counts.....	57
The Train Problem	57
Sensitivity to the Prior	60
Power Law Prior	61
Credible Intervals	63
The German Tank Problem	64
Informative Priors	65
Summary	66
Exercises	66
6. Odds and Addends.....	69
Odds	69
Bayes's Rule	70
Oliver's Blood	71
Addends	73
Gluten Sensitivity	76
The Forward Problem	77
The Inverse Problem	78
Summary	80
More Exercises	81

7. Minimum, Maximum, and Mixture.....	83
Cumulative Distribution Functions	83
Best Three of Four	86
Maximum	88
Minimum	89
Mixture	90
General Mixtures	93
Summary	96
Exercises	97
8. Poisson Processes.....	99
The World Cup Problem	99
The Poisson Distribution	100
The Gamma Distribution	101
The Update	103
Probability of Superiority	105
Predicting the Rematch	106
The Exponential Distribution	108
Summary	110
Exercises	110
9. Decision Analysis.....	113
The Price Is Right Problem	113
The Prior	114
Kernel Density Estimation	115
Distribution of Error	116
Update	118
Probability of Winning	120
Decision Analysis	122
Maximizing Expected Gain	124
Summary	126
Discussion	126
More Exercises	127
10. Testing.....	129
Estimation	129
Evidence	131
Uniformly Distributed Bias	132
Bayesian Hypothesis Testing	134
Bayesian Bandits	134
Prior Beliefs	135
The Update	136

Multiple Bandits	137
Explore and Exploit	138
The Strategy	140
Summary	142
More Exercises	142
11. Comparison.....	145
Outer Operations	145
How Tall Is A?	147
Joint Distribution	148
Visualizing the Joint Distribution	149
Likelihood	151
The Update	152
Marginal Distributions	153
Conditional Posteriors	156
Dependence and Independence	157
Summary	158
Exercises	158
12. Classification.....	161
Penguin Data	161
Normal Models	163
The Update	164
Naive Bayesian Classification	166
Joint Distributions	168
Multivariate Normal Distribution	170
A Less Naive Classifier	172
Summary	173
Exercises	173
13. Inference.....	175
Improving Reading Ability	175
Estimating Parameters	177
Likelihood	178
Posterior Marginal Distributions	180
Distribution of Differences	181
Using Summary Statistics	184
Update with Summary Statistics	186
Comparing Marginals	187
Summary	188
Exercises	189

14. Survival Analysis.....	191
The Weibull Distribution	191
Incomplete Data	194
Using Incomplete Data	196
Light Bulbs	199
Posterior Means	201
Posterior Predictive Distribution	202
Summary	204
Exercises	204
15. Mark and Recapture.....	207
The Grizzly Bear Problem	207
The Update	209
Two-Parameter Model	211
The Prior	212
The Update	213
The Lincoln Index Problem	215
Three-Parameter Model	217
Summary	220
Exercises	221
16. Logistic Regression.....	223
Log Odds	223
The Space Shuttle Problem	226
Prior Distribution	229
Likelihood	230
The Update	231
Marginal Distributions	232
Transforming Distributions	233
Predictive Distributions	235
Empirical Bayes	237
Summary	238
More Exercises	238
17. Regression.....	241
More Snow?	241
Regression Model	243
Least Squares Regression	244
Priors	245
Likelihood	246
The Update	247
Marathon World Record	250

The Priors	252
Prediction	254
Summary	255
Exercises	255
18. Conjugate Priors.....	257
The World Cup Problem Revisited	257
The Conjugate Prior	258
What the Actual?	260
Binomial Likelihood	261
Lions and Tigers and Bears	263
The Dirichlet Distribution	264
Summary	266
Exercises	267
19. MCMC.....	269
The World Cup Problem	269
Grid Approximation	270
Prior Predictive Distribution	270
Introducing PyMC3	271
Sampling the Prior	272
When Do We Get to Inference?	274
Posterior Predictive Distribution	275
Happiness	276
Simple Regression	277
Multiple Regression	280
Summary	282
Exercises	283
20. Approximate Bayesian Computation.....	287
The Kidney Tumor Problem	287
A Simple Growth Model	288
A More General Model	289
Simulation	291
Approximate Bayesian Computation	294
Counting Cells	295
Cell Counting with ABC	298
When Do We Get to the Approximate Part?	299
Summary	302
Exercises	303
Index.....	305

Preface

The premise of this book, and the other books in the *Think X* series, is that if you know how to program, you can use that skill to learn other topics.

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

I think 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 worrying too much about whether the model lends itself to mathematical analysis.

Also, it provides a smooth path from simple examples to real-world problems.

Who Is This Book For?

To start this book, you should be comfortable with Python. If you are familiar with NumPy and pandas, that will help, but I'll explain what you need as we go. You don't need to know calculus or linear algebra. You don't need any prior knowledge of statistics.

In [Chapter 1](#), I define probability and introduce conditional probability, which is the foundation of Bayes's theorem. [Chapter 3](#) introduces the probability distribution, which is the foundation of Bayesian statistics.

In later chapters, we use a variety of discrete and continuous distributions, including the binomial, exponential, Poisson, beta, gamma, and normal distributions. I will explain each distribution when it is introduced, and we will use SciPy to compute them, so you don't need to know about their mathematical properties.

Modeling

Most chapters in this book are motivated by a real-world problem, so they involve some degree of modeling. Before we can apply Bayesian methods (or any other analysis), we have to make decisions about which parts of the real-world system to include in the model and which details we can abstract away.

For example, in [Chapter 8](#), the motivating problem is to predict the winner of a soccer (football) game. I model goal-scoring as a Poisson process, which implies that a goal is equally likely at any point in the game. That is not exactly true, but it is probably a good enough model for most purposes.

I think it is important to include modeling as an explicit part of problem solving because it reminds us to think about modeling errors (that is, errors due to simplifications and assumptions of the model).

Many of the methods in this book are based on discrete distributions, which makes some people worry about numerical errors. But for real-world problems, numerical errors are almost always smaller than modeling errors.

Furthermore, the discrete approach often allows better modeling decisions, and I would rather have an approximate solution to a good model than an exact solution to a bad model.

Working with the Code

Reading this book will only get you so far; to really understand it, you have to work with the code. The original form of this book is a series of Jupyter notebooks. After you read each chapter, I encourage you to run the notebook and work on the exercises. If you need help, my solutions are available.

There are several ways to run the notebooks:

- If you have Python and Jupyter installed, you can download the notebooks and run them on your computer.
- If you don't have a programming environment where you can run Jupyter notebooks, you can use Colab, which lets you run Jupyter notebooks in a browser without installing anything.

To run the notebooks on Colab, start from [this landing page](#), which has links to all of the notebooks.

If you already have Python and Jupyter, you can [download the notebooks as a ZIP file](#).

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Think Bayes*, Second Edition, by Allen B. Downey (O'Reilly). Copyright 2021 Allen B. Downey, 978-1-492-08946-9.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact O'Reilly Media at permissions@oreilly.com.

Installing Jupyter

If you don't have Python and Jupyter already, I recommend you install Anaconda, which is a free Python distribution that includes all the packages you'll need. I found Anaconda easy to install. By default it installs files in your home directory, so you don't need administrator privileges. You can download Anaconda from [this site](#).

Anaconda includes most of the packages you need to run the code in this book. But there are a few additional packages you need to install.

To make sure you have everything you need (and the right versions), the best option is to create a Conda environment. [Download this Conda environment file](#) and run the following commands to create and activate an environment called ThinkBayes2:

```
conda env create -f environment.yml  
conda activate ThinkBayes2
```

If you don't want to create an environment just for this book, you can install what you need using Conda. The following commands should get everything you need:

```
conda install python jupyter pandas scipy matplotlib  
pip install empiricaldist
```

If you don't want to use Anaconda, you will need the following packages:

- Jupyter to run the notebooks, <https://jupyter.org>;
- NumPy for basic numerical computation, <https://numpy.org>;
- SciPy for scientific computation, <https://scipy.org>;

- pandas for working with data, <https://pandas.pydata.org>;
- matplotlib for visualization, <https://matplotlib.org>;
- empiricaldist for representing distributions, <https://pypi.org/project/empiricaldist>.

Although these are commonly used packages, they are not included with all Python installations, and they can be hard to install in some environments. If you have trouble installing them, I recommend using Anaconda or one of the other Python distributions that include these packages.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates URLs, email addresses, filenames, and file extensions.

Bold

Indicates new and key terms.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

O'Reilly Online Learning

O'REILLY® For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <http://oreilly.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://oreil.ly/thinkBayes2e>.

Email bookquestions@oreilly.com to comment or ask technical questions about this book.

For news and information about our books and courses, visit <http://oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://youtube.com/oreillymedia>

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 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, I use several problems from his book as examples.
- Several examples and exercises in the second edition are borrowed, with permission, from Cameron Davidson-Pilon and one exercise from Rasmus Bååth.
- This book also benefited from my interactions with Sanjoy Mahajan, especially in Fall 2012, when I audited his class on Bayesian Inference at Olin College.

- Many examples in this book were developed in collaboration with students in my Bayesian Statistics classes at Olin College. In particular, the Red Line example started as a class project by Brendan Ritter and Kai Austin.
- I wrote parts of this book during project nights with the Boston Python User Group, so I would like to thank them for their company and pizza.
- Jasmine Kwityn and Dan Fauxsmith at O'Reilly Media proofread the first edition and found many opportunities for improvement.
- Linda Pescatore found a typo and made some helpful suggestions.
- Tomasz Miasko sent many excellent corrections and suggestions.
- For the second edition, I want to thank Michele Cronin and Kristen Brown at O'Reilly Media and the technical reviewers Ravin Kumar, Thomas Nield, Josh Starmer, and Junpeng Lao.
- I am grateful to the developers and contributors of the software libraries this book is based on, especially Jupyter, NumPy, SciPy, pandas, PyMC, ArviZ, and Matplotlib.

Other people who spotted typos and errors include Greg Marra, Matt Aasted, Marcus Ogren, Tom Pollard, Paul A. Giannaros, Jonathan Edwards, George Purkins, Robert Marcus, Ram Limbu, James Lawry, Ben Kahle, Jeffrey Law, Alvaro Sanchez, Olivier Yiptong, Yuriy Pasichnyk, Kristopher Overholt, Max Hailperin, Markus Dobler, Brad Minch, Allen Minch, Nathan Yee, Michael Mera, Chris Krenn, and Daniel Vianna.

CHAPTER 1

Probability

The foundation of Bayesian statistics is Bayes's theorem, and the foundation of Bayes's theorem is conditional probability.

In this chapter, we'll start with conditional probability, derive Bayes's theorem, and demonstrate it using a real dataset. In the next chapter, we'll use Bayes's theorem to solve problems related to conditional probability. In the chapters that follow, we'll make the transition from Bayes's theorem to Bayesian statistics, and I'll explain the difference.

Linda the Banker

To introduce conditional probability, I'll use an example from a [famous experiment by Tversky and Kahneman](#), who posed the following question:

Linda is 31 years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in anti-nuclear demonstrations. Which is more probable?

1. Linda is a bank teller.
2. Linda is a bank teller and is active in the feminist movement.

Many people choose the second answer, presumably because it seems more consistent with the description. It seems uncharacteristic if Linda is *just* a bank teller; it seems more consistent if she is also a feminist.

But the second answer cannot be “more probable”, as the question asks. Suppose we find 1,000 people who fit Linda's description and 10 of them work as bank tellers. How many of them are also feminists? At most, all 10 of them are; in that case, the two options are *equally* probable. If fewer than 10 are, the second option is *less* probable. But there is no way the second option can be *more* probable.

If you were inclined to choose the second option, you are in good company. The biologist **Stephen J. Gould** wrote:

I am particularly fond of this example because I know that the [second] statement is least probable, yet a little **homunculus** in my head continues to jump up and down, shouting at me, “but she can’t just be a bank teller; read the description.”

If the little person in your head is still unhappy, maybe this chapter will help.

Probability

At this point I should provide a definition of “probability”, but that **turns out to be surprisingly difficult**. To avoid getting stuck before we start, we will use a simple definition for now and refine it later: A **probability** is a fraction of a finite set.

For example, if we survey 1,000 people, and 20 of them are bank tellers, the fraction that work as bank tellers is 0.02 or 2%. If we choose a person from this population at random, the probability that they are a bank teller is 2%. By “at random” I mean that every person in the dataset has the same chance of being chosen.

With this definition and an appropriate dataset, we can compute probabilities by counting. To demonstrate, I’ll use data from the **General Social Survey** (GSS).

I’ll use pandas to read the data and store it in a `DataFrame`.

```
import pandas as pd

gss = pd.read_csv('gss_bayes.csv', index_col=0)
gss.head()
```

	year	age	sex	polviews	partyid	indus10
caseid						
1	1974	21.0	1	4.0	2.0	4970.0
2	1974	41.0	1	5.0	0.0	9160.0
5	1974	58.0	2	6.0	1.0	2670.0
6	1974	30.0	1	5.0	4.0	6870.0
7	1974	48.0	1	5.0	4.0	7860.0

The `DataFrame` has one row for each person surveyed and one column for each variable I selected.

The columns are

- `caseid`: Respondent id (which is the index of the table).
- `year`: Year when the respondent was surveyed.
- `age`: Respondent's age when surveyed.
- `sex`: Male or female.
- `polviews`: Political views on a range from liberal to conservative.
- `partyid`: Political party affiliation: Democratic, Republican, or independent.
- `indus10`: Code for the industry the respondent works in.

Let's look at these variables in more detail, starting with `indus10`.

Fraction of Bankers

The code for “Banking and related activities” is 6870, so we can select bankers like this:

```
banker = (gss['indus10'] == 6870)
banker.head()

caseid
1    False
2    False
5    False
6    True
7    False
Name: indus10, dtype: bool
```

The result is a pandas `Series` that contains the Boolean values `True` and `False`.

If we use the `sum` function on this `Series`, it treats `True` as 1 and `False` as 0, so the total is the number of bankers:

```
banker.sum()
```

```
728
```

In this dataset, there are 728 bankers.

To compute the *fraction* of bankers, we can use the `mean` function, which computes the fraction of `True` values in the `Series`:

```
banker.mean()
```

```
0.014769730168391155
```

About 1.5% of the respondents work in banking, so if we choose a random person from the dataset, the probability they are a banker is about 1.5%.

The Probability Function

I'll put the code from the previous section in a function that takes a Boolean Series and returns a probability:

```
def prob(A):
    """Computes the probability of a proposition, A."""
    return A.mean()
```

So we can compute the fraction of bankers like this:

```
prob(banker)
0.014769730168391155
```

Now let's look at another variable in this dataset. The values of the column `sex` are encoded like this:

```
1  Male
2  Female
```

So we can make a Boolean Series that is `True` for female respondents and `False` otherwise:

```
female = (gss['sex'] == 2)
```

And use it to compute the fraction of respondents who are women:

```
prob(female)
0.5378575776019476
```

The fraction of women in this dataset is higher than in the adult US population because the GSS doesn't include people living in institutions like prisons and military housing, and those populations are more likely to be male.

Political Views and Parties

The other variables we'll consider are `polviews`, which describes the political views of the respondents, and `partyid`, which describes their affiliation with a political party.

The values of `polviews` are on a seven-point scale:

```
1  Extremely liberal
2  Liberal
3  Slightly liberal
4  Moderate
5  Slightly conservative
6  Conservative
7  Extremely conservative
```

I'll define `liberal` to be `True` for anyone whose response is "Extremely liberal", "Liberal", or "Slightly liberal":

```
liberal = (gss['polviews'] <= 3)
```

Here's the fraction of respondents who are liberal by this definition:

```
prob(liberal)
```

```
0.27374721038750255
```

If we choose a random person in this dataset, the probability they are liberal is about 27%.

The values of `partyid` are encoded like this:

0	Strong democrat
1	Not strong democrat
2	Independent, near democrat
3	Independent
4	Independent, near republican
5	Not strong republican
6	Strong republican
7	Other party

I'll define `democrat` to include respondents who chose "Strong democrat" or "Not strong democrat":

```
democrat = (gss['partyid'] <= 1)
```

And here's the fraction of respondents who are Democrats, by this definition:

```
prob(democrat)
```

```
0.3662609048488537
```

Conjunction

Now that we have a definition of probability and a function that computes it, let's move on to conjunction.

"Conjunction" is another name for the logical `and` operation. If you have two **propositions**, A and B, the conjunction A `and` B is True if both A and B are True, and False otherwise.

If we have two Boolean **Series**, we can use the `&` operator to compute their conjunction. For example, we have already computed the probability that a respondent is a banker:

```
prob(banker)
```

```
0.014769730168391155
```

And the probability that they are a Democrat:

```
prob(democrat)
```

```
0.3662609048488537
```

Now we can compute the probability that a respondent is a banker *and* a Democrat:

```
prob(banker & democrat)
```

```
0.004686548995739501
```

As we should expect, `prob(banker & democrat)` is less than `prob(banker)`, because not all bankers are Democrats.

We expect conjunction to be commutative; that is, A & B should be the same as B & A. To check, we can also compute `prob(democrat & banker)`:

```
prob(democrat & banker)
```

```
0.004686548995739501
```

As expected, they are the same.

Conditional Probability

Conditional probability is a probability that depends on a condition, but that might not be the most helpful definition. Here are some examples:

- What is the probability that a respondent is a Democrat, given that they are liberal?
- What is the probability that a respondent is female, given that they are a banker?
- What is the probability that a respondent is liberal, given that they are female?

Let's start with the first one, which we can interpret like this: "Of all the respondents who are liberal, what fraction are Democrats?"

We can compute this probability in two steps:

1. Select all respondents who are liberal.
2. Compute the fraction of the selected respondents who are Democrats.

To select liberal respondents, we can use the bracket operator, [], like this:

```
selected = democrat[liberal]
```

`selected` contains the values of `democrat` for liberal respondents, so `prob(selected)` is the fraction of liberals who are Democrats:

```
prob(selected)
```

```
0.5206403320240125
```

A little more than half of liberals are Democrats. If that result is lower than you expected, keep in mind:

1. We used a somewhat strict definition of “Democrat”, excluding independents who “lean” Democratic.
2. The dataset includes respondents as far back as 1974; in the early part of this interval, there was less alignment between political views and party affiliation, compared to the present.

Let's try the second example, “What is the probability that a respondent is female, given that they are a banker?” We can interpret that to mean, “Of all respondents who are bankers, what fraction are female?”

Again, we'll use the bracket operator to select only the bankers and `prob` to compute the fraction that are female:

```
selected = female[banker]
prob(selected)

0.7706043956043956
```

About 77% of the bankers in this dataset are female.

Let's wrap this computation in a function. I'll define `conditional` to take two Boolean Series, `proposition` and `given`, and compute the conditional probability of `proposition` conditioned on `given`:

```
def conditional(proposition, given):
    return prob(proposition[given])
```

We can use `conditional` to compute the probability that a respondent is liberal given that they are female:

```
conditional(liberal, given=female)

0.27581004111500884
```

About 28% of female respondents are liberal.

I included the keyword, `given`, along with the parameter, `female`, to make this expression more readable.

Conditional Probability Is Not Commutative

We have seen that conjunction is commutative; that is, `prob(A & B)` is always equal to `prob(B & A)`.

But conditional probability is *not* commutative; that is, `conditional(A, B)` is not the same as `conditional(B, A)`.

That should be clear if we look at an example. Previously, we computed the probability a respondent is female, given that they are a banker.

```
conditional(female, given=banker)
```

```
0.7706043956043956
```

The result shows that the majority of bankers are female. That is not the same as the probability that a respondent is a banker, given that they are female:

```
conditional(banker, given=female)
```

```
0.02116102749801969
```

Only about 2% of female respondents are bankers.

I hope this example makes it clear that conditional probability is not commutative, and maybe it was already clear to you. Nevertheless, it is a common error to confuse `conditional(A, B)` and `conditional(B, A)`. We'll see some examples later.

Condition and Conjunction

We can combine conditional probability and conjunction. For example, here's the probability a respondent is female, given that they are a liberal Democrat:

```
conditional(female, given=liberal & democrat)
```

```
0.576085409252669
```

About 57% of liberal Democrats are female.

And here's the probability they are a liberal female, given that they are a banker:

```
conditional(liberal & female, given=banker)
```

```
0.17307692307692307
```

About 17% of bankers are liberal women.

Laws of Probability

In the next few sections, we'll derive three relationships between conjunction and conditional probability:

- Theorem 1: Using a conjunction to compute a conditional probability.
- Theorem 2: Using a conditional probability to compute a conjunction.
- Theorem 3: Using `conditional(A, B)` to compute `conditional(B, A)`.

Theorem 3 is also known as Bayes's theorem.

I'll write these theorems using mathematical notation for probability:

- $P(A)$ is the probability of proposition A .

- $P(A \text{ and } B)$ is the probability of the conjunction of A and B , that is, the probability that both are true.
- $P(A | B)$ is the conditional probability of A given that B is true. The vertical line between A and B is pronounced “given”.

With that, we are ready for Theorem 1.

Theorem 1

What fraction of bankers are female? We have already seen one way to compute the answer:

1. Use the bracket operator to select the bankers, then
2. Use `mean` to compute the fraction of bankers who are female.

We can write these steps like this:

```
female[banker].mean()
0.7706043956043956
```

Or we can use the `conditional` function, which does the same thing:

```
conditional(female, given=banker)
0.7706043956043956
```

But there is another way to compute this conditional probability, by computing the ratio of two probabilities:

1. The fraction of respondents who are female bankers, and
2. The fraction of respondents who are bankers.

In other words: of all the bankers, what fraction are female bankers? Here's how we compute this ratio:

```
prob(female & banker) / prob(banker)
0.7706043956043956
```

The result is the same. This example demonstrates a general rule that relates conditional probability and conjunction. Here's what it looks like in math notation:

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

And that's Theorem 1.

Theorem 2

If we start with Theorem 1 and multiply both sides by $P(B)$, we get Theorem 2:

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

This formula suggests a second way to compute a conjunction: instead of using the `&` operator, we can compute the product of two probabilities.

Let's see if it works for `liberal` and `democrat`. Here's the result using `&`:

```
prob(liberal & democrat)  
0.1425238385067965
```

And here's the result using Theorem 2:

```
prob(democrat) * conditional(liberal, democrat)  
0.1425238385067965
```

They are the same.

Theorem 3

We have established that conjunction is commutative. In math notation, that means:

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

If we apply Theorem 2 to both sides, we have:

$$P(B)P(A | B) = P(A)P(B | A)$$

Here's one way to interpret that: if you want to check A and B , you can do it in either order:

1. You can check B first, then A conditioned on B , or
2. You can check A first, then B conditioned on A .

If we divide through by $P(B)$, we get Theorem 3:

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

And that, my friends, is Bayes's theorem.

To see how it works, let's compute the fraction of bankers who are liberal, first using `conditional`:

```
conditional(liberal, given=banker)
```

```
0.2239010989010989
```

Now using Bayes's theorem:

```
prob(liberal) * conditional(banker, liberal) / prob(banker)
```

```
0.2239010989010989
```

They are the same.

The Law of Total Probability

In addition to these three theorems, there's one more thing we'll need to do Bayesian statistics: the law of total probability. Here's one form of the law, expressed in mathematical notation:

$$P(A) = P(B_1 \text{and} A) + P(B_2 \text{and} A)$$

In words, the total probability of A is the sum of two possibilities: either B_1 and A are true or B_2 and A are true. But this law applies only if B_1 and B_2 are:

- Mutually exclusive, which means that only one of them can be true, and
- Collectively exhaustive, which means that one of them must be true.

As an example, let's use this law to compute the probability that a respondent is a banker. We can compute it directly like this:

```
prob(banker)
```

```
0.014769730168391155
```

So let's confirm that we get the same thing if we compute male and female bankers separately.

In this dataset all respondents are designated male or female. Recently, the GSS Board of Overseers announced that they will add more inclusive gender questions to the survey (you can read more about this issue, and their decision, at <https://oreil.ly/onK2P>).

We already have a Boolean Series that is `True` for female respondents. Here's the complementary Series for male respondents:

```
male = (gss['sex'] == 1)
```

Now we can compute the total probability of `banker` like this:

```
prob(male & banker) + prob(female & banker)  
0.014769730168391155
```

Because `male` and `female` are mutually exclusive and collectively exhaustive (MECE), we get the same result we got by computing the probability of `banker` directly.

Applying Theorem 2, we can also write the law of total probability like this:

$$P(A) = P(B_1)P(A|B_1) + P(B_2)P(A|B_2)$$

And we can test it with the same example:

```
(prob(male) * conditional(banker, given=male) +  
prob(female) * conditional(banker, given=female))  
0.014769730168391153
```

When there are more than two conditions, it is more concise to write the law of total probability as a summation:

$$P(A) = \sum_i P(B_i)P(A|B_i)$$

Again, this holds as long as the conditions B_i are mutually exclusive and collectively exhaustive. As an example, let's consider `polviews`, which has seven different values:

```
B = gss['polviews']  
B.value_counts().sort_index()  
  
1.0      1442  
2.0      5808  
3.0      6243  
4.0     18943  
5.0      7940  
6.0      7319  
7.0      1595  
Name: polviews, dtype: int64
```

On this scale, `4.0` represents “Moderate”. So we can compute the probability of a moderate banker like this:

```
i = 4  
prob(B==i) * conditional(banker, B==i)  
0.005822682085615744
```

And we can use `sum` and a **generator expression** to compute the summation:

```
sum(prob(B==i) * conditional(banker, B==i)
    for i in range(1, 8))
0.014769730168391157
```

The result is the same.

In this example, using the law of total probability is a lot more work than computing the probability directly, but it will turn out to be useful, I promise.

Summary

Here's what we have so far:

Theorem 1 gives us a way to compute a conditional probability using a conjunction:

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

Theorem 2 gives us a way to compute a conjunction using a conditional probability:

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

Theorem 3, also known as Bayes's theorem, gives us a way to get from $P(A | B)$ to $P(B | A)$, or the other way around:

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

The Law of Total Probability provides a way to compute probabilities by adding up the pieces:

$$P(A) = \sum_i P(B_i)P(A | B_i)$$

At this point you might ask, “So what?” If we have all of the data, we can compute any probability we want, any conjunction, or any conditional probability, just by counting. We don't have to use these formulas.

And you are right, *if* we have all of the data. But often we don't, and in that case, these formulas can be pretty useful—especially Bayes's theorem. In the next chapter, we'll see how.

Exercises

Exercise 1-1.

Let's use the tools in this chapter to solve a variation of the Linda problem.

Linda is 31 years old, single, outspoken, and very bright. She majored in philosophy. As a student, she was deeply concerned with issues of discrimination and social justice, and also participated in anti-nuclear demonstrations. Which is more probable?

1. Linda is a banker.
2. Linda is a banker and considers herself a liberal Democrat.

To answer this question, compute

- The probability that Linda is a female banker,
- The probability that Linda is a liberal female banker, and
- The probability that Linda is a liberal female banker and a Democrat.

Exercise 1-2.

Use `conditional` to compute the following probabilities:

- What is the probability that a respondent is liberal, given that they are a Democrat?
- What is the probability that a respondent is a Democrat, given that they are liberal?

Think carefully about the order of the arguments you pass to `conditional`.

Exercise 1-3.

There's a **famous quote** about young people, old people, liberals, and conservatives that goes something like:

If you are not a liberal at 25, you have no heart. If you are not a conservative at 35, you have no brain.

Whether you agree with this proposition or not, it suggests some probabilities we can compute as an exercise. Rather than use the specific ages 25 and 35, let's define `young` and `old` as under 30 or over 65:

```
young = (gss['age'] < 30)  
prob(young)
```

```
0.19435991073240008
```

```
old = (gss['age'] >= 65)
prob(old)
```

```
0.17328058429701765
```

For these thresholds, I chose round numbers near the 20th and 80th percentiles. Depending on your age, you may or may not agree with these definitions of “young” and “old”.

I’ll define `conservative` as someone whose political views are “Conservative”, “Slightly Conservative”, or “Extremely Conservative”.

```
conservative = (gss['polviews'] >= 5)
prob(conservative)
```

```
0.3419354838709677
```

Use `prob` and `conditional` to compute the following probabilities:

- What is the probability that a randomly chosen respondent is a young liberal?
- What is the probability that a young person is liberal?
- What fraction of respondents are old conservatives?
- What fraction of conservatives are old?

For each statement, think about whether it is expressing a conjunction, a conditional probability, or both.

For the conditional probabilities, be careful about the order of the arguments. If your answer to the last question is greater than 30%, you have it backwards!

CHAPTER 2

Bayes's Theorem

In the previous chapter, we derived Bayes's theorem:

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

As an example, we used data from the General Social Survey and Bayes's theorem to compute conditional probabilities. But since we had the complete dataset, we didn't really need Bayes's theorem. It was easy enough to compute the left side of the equation directly, and no easier to compute the right side.

But often we don't have a complete dataset, and in that case Bayes's theorem is more useful. In this chapter, we'll use it to solve several more challenging problems related to conditional probability.

The Cookie Problem

We'll start with a thinly disguised version of an **urn problem**:

Suppose there are two bowls of cookies.

- Bowl 1 contains 30 vanilla cookies and 10 chocolate cookies.
- Bowl 2 contains 20 vanilla cookies and 20 chocolate cookies.

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

What we want is the conditional probability that we chose from Bowl 1 given that we got a vanilla cookie, $P(B_1 | V)$.

But what we get from the statement of the problem is:

- The conditional probability of getting a vanilla cookie, given that we chose from Bowl 1, $P(V|B_1)$ and
- The conditional probability of getting a vanilla cookie, given that we chose from Bowl 2, $P(V|B_2)$.

Bayes's theorem tells us how they are related:

$$P(B_1|V) = \frac{P(B_1) P(V|B_1)}{P(V)}$$

The term on the left is what we want. The terms on the right are:

- $P(B_1)$, 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 assume $P(B_1) = 1/2$.
- $P(V|B_1)$, the probability of getting a vanilla cookie from Bowl 1, which is $3/4$.
- $P(V)$, the probability of drawing a vanilla cookie from either bowl.

To compute $P(V)$, we can use the law of total probability:

$$P(V) = P(B_1) P(V|B_1) + P(B_2) P(V|B_2)$$

Plugging in the numbers from the statement of the problem, we have:

$$P(V) = (1/2) (3/4) + (1/2) (1/2) = 5/8$$

We can also compute this result directly, like this:

- Since we had an equal chance of choosing either bowl and the bowls contain the same number of cookies, we had the same chance of choosing any cookie.
- Between the two bowls there are 50 vanilla and 30 chocolate cookies, so $P(V) = 5/8$.

Finally, we can apply Bayes's theorem to compute the posterior probability of Bowl 1:

$$P(B_1|V) = (1/2) (3/4) / (5/8) = 3/5$$

This example demonstrates one use of Bayes's theorem: it provides a way to get from $P(B|A)$ to $P(A|B)$. This strategy is useful in cases like this where it is easier to compute the terms on the right side than the term on the left.

Diachronic Bayes

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

This interpretation is “diachronic”, which means “related to change over time”; in this case, the probability of the hypotheses changes as we see new data.

Rewriting Bayes's theorem with H and D yields:

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

In this interpretation, each term has a name:

- $P(H)$ is the probability of the hypothesis before we see the data, called the **prior** probability, or just **prior**.
- $P(H|D)$ is the probability of the hypothesis after we see the data, called the **posterior**.
- $P(D|H)$ is the probability of the data under the hypothesis, called the **likelihood**.
- $P(D)$ is the **total probability of the data**, under any hypothesis.

Sometimes we can compute the prior based on background information. For example, the Cookie Problem specifies that we choose a bowl at random with 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, we are given the number of cookies in each bowl, so we can compute the probability of the data under each hypothesis.

Computing the total probability of the data can be tricky. It is supposed to be the probability of seeing the data under any hypothesis at all, but it can be hard to nail down what that means.

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

- Mutually exclusive, which means that only one of them can be true, and
- Collectively exhaustive, which means one of them must be true.

When these conditions apply, we can compute $P(D)$ using the law of total probability. For example, with two hypotheses, H_1 and H_2 :

$$P(D) = P(H_1) P(D|H_1) + P(H_2) P(D|H_2)$$

And more generally, with any number of hypotheses:

$$P(D) = \sum_i P(H_i) P(D|H_i)$$

The process in this section, using data and a prior probability to compute a posterior probability, is called a **Bayesian update**.

Bayes Tables

A convenient tool for doing a Bayesian update is a Bayes table. You can write a Bayes table on paper or use a spreadsheet, but in this section I'll use a pandas DataFrame.

First I'll make an empty DataFrame with one row for each hypothesis:

```
import pandas as pd

table = pd.DataFrame(index=['Bowl 1', 'Bowl 2'])
```

Now I'll add a column to represent the priors:

```
table['prior'] = 1/2, 1/2
table
```

	prior
Bowl 1	0.5
Bowl 2	0.5

And a column for the likelihoods:

```
table['likelihood'] = 3/4, 1/2
table
```

	prior	likelihood
Bowl 1	0.5	0.75
Bowl 2	0.5	0.50

Here we see a difference from the previous method: we compute likelihoods for both hypotheses, not just Bowl 1:

- The chance of getting a vanilla cookie from Bowl 1 is 3/4.
- The chance of getting a vanilla cookie from Bowl 2 is 1/2.

You might notice that the likelihoods don't add up to 1. That's OK; each of them is a probability conditioned on a different hypothesis. There's no reason they should add up to 1 and no problem if they don't.

The next step is similar to what we did with Bayes's theorem; we multiply the priors by the likelihoods:

```
table['unnorm'] = table['prior'] * table['likelihood']  
table
```

	prior	likelihood	unnorm
Bowl 1	0.5	0.75	0.375
Bowl 2	0.5	0.50	0.250

I call the result `unnorm` because these values are the “unnormalized posteriors”. Each of them is the product of a prior and a likelihood

$$P(B_i) P(D|B_i)$$

which is the numerator of Bayes's theorem. If we add them up, we have

$$P(B_1) P(D|B_1) + P(B_2) P(D|B_2)$$

which is the denominator of Bayes's theorem, $P(D)$.

So we can compute the total probability of the data like this:

```
prob_data = table['unnorm'].sum()  
prob_data  
0.625
```

Notice that we get 5/8, which is what we got by computing $P(D)$ directly.

And we can compute the posterior probabilities like this:

```
table['posterior'] = table['unnorm'] / prob_data  
table
```

	prior	likelihood	unnorm	posterior
Bowl 1	0.5	0.75	0.375	0.6
Bowl 2	0.5	0.50	0.250	0.4

The posterior probability for Bowl 1 is 0.6, which is what we got using Bayes's theorem explicitly. As a bonus, we also get the posterior probability of Bowl 2, which is 0.4.

When we add up the unnormalized posteriors and divide through, we force the posteriors to add up to 1. This process is called “normalization”, which is why the total probability of the data is also called the “normalizing constant”.

The Dice Problem

A Bayes table can also solve problems with more than two hypotheses. For example:

Suppose I have a box with a 6-sided die, an 8-sided die, and a 12-sided die. I choose one of the dice at random, roll it, and report that the outcome is a 1. What is the probability that I chose the 6-sided die?

In this example, there are three hypotheses with equal prior probabilities. The data is my report that the outcome is a 1.

If I chose the 6-sided die, the probability of the data is 1/6. If I chose the 8-sided die, the probability is 1/8, and if I chose the 12-sided die, it's 1/12.

Here's a Bayes table that uses integers to represent the hypotheses:

```
table2 = pd.DataFrame(index=[6, 8, 12])
```

I'll use fractions to represent the prior probabilities and the likelihoods. That way they don't get rounded off to floating-point numbers.

```
from fractions import Fraction

table2['prior'] = Fraction(1, 3)
table2['likelihood'] = Fraction(1, 6), Fraction(1, 8), Fraction(1, 12)
table2
```

	prior	likelihood
6	1/3	1/6
8	1/3	1/8
12	1/3	1/12

Once you have priors and likelihoods, the remaining steps are always the same, so I'll put them in a function:

```

def update(table):
    """Compute the posterior probabilities."""
    table['unnorm'] = table['prior'] * table['likelihood']
    prob_data = table['unnorm'].sum()
    table['posterior'] = table['unnorm'] / prob_data
    return prob_data

```

And call it like this:

```
prob_data = update(table2)
```

Here is the final Bayes table:

	prior	likelihood	unnorm	posterior
6	1/3	1/6	1/18	4/9
8	1/3	1/8	1/24	1/3
12	1/3	1/12	1/36	2/9

The posterior probability of the 6-sided die is 4/9, which is a little more than the probabilities for the other dice, 3/9 and 2/9. Intuitively, the 6-sided die is the most likely because it had the highest likelihood of producing the outcome we saw.

The Monty Hall Problem

Next we'll use a Bayes table to solve one of the most contentious problems in probability.

The Monty Hall Problem is based on a game show called *Let's Make a Deal*. If you are a contestant on the show, here's how the game works:

- The host, Monty Hall, shows you three closed doors—numbered 1, 2, and 3—and tells you that there is a prize behind each door.
- One prize is valuable (traditionally a car), the other two are less valuable (traditionally goats).
- The object of the game is to guess which door has the car. If you guess right, you get to keep the car.

Suppose you pick Door 1. Before opening the door you chose, Monty opens Door 3 and reveals a goat. Then Monty offers you the option to stick with your original choice or switch to the remaining unopened door.

To maximize your chance of winning the car, should you stick with Door 1 or switch to Door 2?

To answer this question, we have to make some assumptions about the behavior of the host:

1. Monty always opens a door and offers you the option to switch.
2. He never opens the door you picked or the door with the car.
3. If you choose the door with the car, he chooses one of the other doors at random.

Under these assumptions, you are better off switching. If you stick, you win $1/3$ of the time. If you switch, you win $2/3$ of the time.

If you have not encountered this problem before, you might find that answer surprising. You would not be alone; many people have the strong intuition that it doesn't matter if you stick or switch. There are two doors left, they reason, so the chance that the car is behind Door A is 50%. But that is wrong.

To see why, it can help to use a Bayes table. We start with three hypotheses: the car might be behind Door 1, 2, or 3. According to the statement of the problem, the prior probability for each door is $1/3$.

```
table3 = pd.DataFrame(index=['Door 1', 'Door 2', 'Door 3'])
table3['prior'] = Fraction(1, 3)
table3
```

	prior
Door 1	$1/3$
Door 2	$1/3$
Door 3	$1/3$

The data is that Monty opened Door 3 and revealed a goat. So let's consider the probability of the data under each hypothesis:

- If the car is behind Door 1, Monty chooses Door 2 or 3 at random, so the probability he opens Door 3 is $1/2$.
- If the car is behind Door 2, Monty has to open Door 3, so the probability of the data under this hypothesis is 1.
- If the car is behind Door 3, Monty does not open it, so the probability of the data under this hypothesis is 0.

Here are the likelihoods:

```
table3['likelihood'] = Fraction(1, 2), 1, 0
table3
```

	prior	likelihood
Door 1	1/3	1/2
Door 2	1/3	1
Door 3	1/3	0

Now that we have priors and likelihoods, we can use `update` to compute the posterior probabilities:

```
update(table3)
table3
```

	prior	likelihood	unnorm	posterior
Door 1	1/3	1/2	1/6	1/3
Door 2	1/3	1	1/3	2/3
Door 3	1/3	0	0	0

After Monty opens Door 3, the posterior probability of Door 1 is 1/3; the posterior probability of Door 2 is 2/3. So you are better off switching from Door 1 to Door 2.

As this example shows, our intuition for probability is not always reliable. Bayes's theorem can help by providing a divide-and-conquer strategy:

1. First, write down the hypotheses and the data.
2. Next, figure out the prior probabilities.
3. Finally, compute the likelihood of the data under each hypothesis.

The Bayes table does the rest.

Summary

In this chapter we solved the Cookie Problem using Bayes's theorem explicitly and using a Bayes table. There's no real difference between these methods, but the Bayes table can make it easier to compute the total probability of the data, especially for problems with more than two hypotheses.

Then we solved the Dice Problem, which we will see again in the next chapter, and the Monty Hall Problem, which you might hope you never see again.

If the Monty Hall Problem makes your head hurt, you are not alone. But I think it demonstrates the power of Bayes's theorem as a divide-and-conquer strategy for solving tricky problems. And I hope it provides some insight into *why* the answer is what it is.

When Monty opens a door, he provides information we can use to update our belief about the location of the car. Part of the information is obvious. If he opens Door 3, we know the car is not behind Door 3. But part of the information is more subtle. Opening Door 3 is more likely if the car is behind Door 2, and less likely if it is behind Door 1. So the data is evidence in favor of Door 2. We will come back to this notion of evidence in future chapters.

In the next chapter we'll extend the Cookie Problem and the Dice Problem, and take the next step from basic probability to Bayesian statistics.

But first, you might want to work on the exercises.

Exercises

Exercise 2-1.

Suppose you have two coins in a box. One is a normal coin with heads on one side and tails on the other, and one is a trick coin with heads on both sides. You choose a coin at random and see that one of the sides is heads. What is the probability that you chose the trick coin?

Exercise 2-2.

Suppose you meet someone and learn that they have two children. You ask if either child is a girl and they say yes. What is the probability that both children are girls?

Hint: Start with four equally likely hypotheses.

Exercise 2-3.

There are many variations of the **Monty Hall Problem**. For example, suppose Monty always chooses Door 2 if he can, and only chooses Door 3 if he has to (because the car is behind Door 2).

If you choose Door 1 and Monty opens Door 2, what is the probability the car is behind Door 3?

If you choose Door 1 and Monty opens Door 3, what is the probability the car is behind Door 2?

Exercise 2-4.

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.

- In 1994, the color mix in a bag of plain M&M's was 30% Brown, 20% Yellow, 20% Red, 10% Green, 10% Orange, 10% Tan.
- In 1996, it was 24% Blue, 20% Green, 16% Orange, 14% Yellow, 13% Red, 13% Brown.

Suppose 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?

Hint: The trick to this question is to define the hypotheses and the data carefully.

CHAPTER 3

Distributions

In the previous chapter we used Bayes's theorem to solve a Cookie Problem; then we solved it again using a Bayes table. In this chapter, at the risk of testing your patience, we will solve it one more time using a `Pmf` object, which represents a “probability mass function”. I'll explain what that means, and why it is useful for Bayesian statistics.

We'll use `Pmf` objects to solve some more challenging problems and take one more step toward Bayesian statistics. But we'll start with distributions.

Distributions

In statistics a **distribution** is a set of possible outcomes and their corresponding probabilities. For example, if you toss a coin, there are two possible outcomes with approximately equal probability. If you roll a 6-sided die, the set of possible outcomes is the numbers 1 to 6, and the probability associated with each outcome is 1/6.

To represent distributions, we'll use a library called `empiricaldist`. An “empirical” distribution is based on data, as opposed to a theoretical distribution. We'll use this library throughout the book. I'll introduce the basic features in this chapter and we'll see additional features later.

Probability Mass Functions

If the outcomes in a distribution are discrete, we can describe the distribution with a **probability mass function**, or PMF, which is a function that maps from each possible outcome to its probability.

`empiricaldist` provides a class called `Pmf` that represents a probability mass function. To use `Pmf` you can import it like this:

```
from empiricaldist import Pmf
```

The following example makes a `Pmf` that represents the outcome of a coin toss.

```
coin = Pmf()
coin['heads'] = 1/2
coin['tails'] = 1/2
coin
```

	probs
heads	0.5
tails	0.5

`Pmf` creates an empty `Pmf` with no outcomes. Then we can add new outcomes using the bracket operator. In this example, the two outcomes are represented with strings, and they have the same probability, 0.5.

You can also make a `Pmf` from a sequence of possible outcomes.

The following example uses `Pmf.from_seq` to make a `Pmf` that represents a 6-sided die.

```
die = Pmf.from_seq([1,2,3,4,5,6])
die
```

	probs
1	0.166667
2	0.166667
3	0.166667
4	0.166667
5	0.166667
6	0.166667

In this example, all outcomes in the sequence appear once, so they all have the same probability, 1/6.

More generally, outcomes can appear more than once, as in the following example:

```
letters = Pmf.from_seq(list('Mississippi'))
letters
```

probs	
M	0.090909
i	0.363636
p	0.181818
s	0.363636

The letter M appears once out of 11 characters, so its probability is 1/11. The letter i appears 4 times, so its probability is 4/11.

Since the letters in a string are not outcomes of a random process, I'll use the more general term "quantities" for the letters in the Pmf.

The Pmf class inherits from a pandas Series, so anything you can do with a Series, you can also do with a Pmf.

For example, you can use the bracket operator to look up a quantity and get the corresponding probability:

```
letters['s']
0.36363636363636365
```

In the word "Mississippi", about 36% of the letters are "s".

However, if you ask for the probability of a quantity that's not in the distribution, you get a KeyError.

You can also call a Pmf as if it were a function, with a letter in parentheses:

```
letters('s')
0.36363636363636365
```

If the quantity is in the distribution, the results are the same. But if it is not in the distribution, the result is 0, not an error:

```
letters('t')
0
```

With parentheses, you can also provide a sequence of quantities and get a sequence of probabilities:

```
die([1,4,7])
array([0.16666667, 0.16666667, 0.          ])
```

The quantities in a Pmf can be strings, numbers, or any other type that can be stored in the index of a pandas Series. If you are familiar with pandas, that will help you work with Pmf objects. But I will explain what you need to know as we go along.

The Cookie Problem Revisited

In this section I'll use a `Pmf` to solve the Cookie Problem from “[The Cookie Problem](#)” on page 17. Here's the statement of the problem again:

Suppose there are two bowls of cookies.

- Bowl 1 contains 30 vanilla cookies and 10 chocolate cookies.
- Bowl 2 contains 20 vanilla cookies and 20 chocolate cookies.

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

Here's a `Pmf` that represents the two hypotheses and their prior probabilities:

```
prior = Pmf.from_seq(['Bowl 1', 'Bowl 2'])
prior
```

	probs
Bowl 1	0.5
Bowl 2	0.5

This distribution, which contains the prior probability for each hypothesis, is called—wait for it—the **prior distribution**.

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

```
likelihood_vanilla = [0.75, 0.5]
posterior = prior * likelihood_vanilla
posterior
```

	probs
Bowl 1	0.375
Bowl 2	0.250

The result is the unnormalized posteriors; that is, they don't add up to 1. To make them add up to 1, we can use `normalize`, which is a method provided by `Pmf`:

```
posterior.normalize()
0.625
```

The return value from `normalize` is the total probability of the data, which is $5/8$.

`posterior`, which contains the posterior probability for each hypothesis, is called—wait now—the **posterior distribution**.

```
posterior
```

	probs
Bowl 1	0.6
Bowl 2	0.4

From the posterior distribution we can select the posterior probability for Bowl 1:

```
posterior('Bowl 1')
```

```
0.6
```

And the answer is 0.6.

One benefit of using `Pmf` objects is that it is easy to do successive updates with more data. For example, suppose you put the first cookie back (so the contents of the bowls don't change) and draw again from the same bowl. If the second cookie is also vanilla, we can do a second update like this:

```
posterior *= likelihood_vanilla
posterior.normalize()
posterior
```

	probs
Bowl 1	0.692308
Bowl 2	0.307692

Now the posterior probability for Bowl 1 is almost 70%. But suppose we do the same thing again and get a chocolate cookie.

Here are the likelihoods for the new data:

```
likelihood_chocolate = [0.25, 0.5]
```

And here's the update:

```
posterior *= likelihood_chocolate
posterior.normalize()
posterior
```

	probs
Bowl 1	0.529412
Bowl 2	0.470588

Now the posterior probability for Bowl 1 is about 53%. After two vanilla cookies and one chocolate, the posterior probabilities are close to 50/50.

101 Bowls

Next let's solve a Cookie Problem with 101 bowls:

- Bowl 0 contains 0% vanilla cookies,
- Bowl 1 contains 1% vanilla cookies,
- Bowl 2 contains 2% vanilla cookies,

and so on, up to

- Bowl 99 contains 99% vanilla cookies, and
- Bowl 100 contains all vanilla cookies.

As in the previous version, there are only two kinds of cookies, vanilla and chocolate. So Bowl 0 is all chocolate cookies, Bowl 1 is 99% chocolate, and so on.

Suppose we choose a bowl at random, choose a cookie at random, and it turns out to be vanilla. What is the probability that the cookie came from Bowl x , for each value of x ?

To solve this problem, I'll use `np.arange` to make an array that represents 101 hypotheses, numbered from 0 to 100:

```
import numpy as np  
  
hypos = np.arange(101)
```

We can use this array to make the prior distribution:

```
prior = Pmf(.1, hypos)  
prior.normalize()  
  
101
```

As this example shows, we can initialize a `Pmf` with two parameters. The first parameter is the prior probability; the second parameter is a sequence of quantities.

In this example, the probabilities are all the same, so we only have to provide one of them; it gets “broadcast” across the hypotheses. Since all hypotheses have the same prior probability, this distribution is **uniform**.

Here are the first few hypotheses and their probabilities:

```
prior.head()
```

probs	
0	0.009901
1	0.009901
2	0.009901

The likelihood of the data is the fraction of vanilla cookies in each bowl, which we can calculate using `hypos`:

```
likelihood_vanilla = hypos/100
likelihood_vanilla[:5]

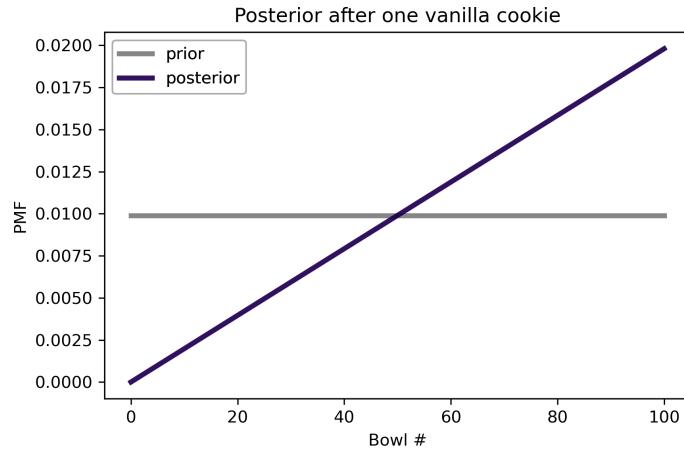
array([0. , 0.01, 0.02, 0.03, 0.04])
```

Now we can compute the posterior distribution in the usual way:

```
posterior1 = prior * likelihood_vanilla
posterior1.normalize()
posterior1.head()
```

probs	
0	0.000000
1	0.000198
2	0.000396

The following figure shows the prior distribution and the posterior distribution after one vanilla cookie:

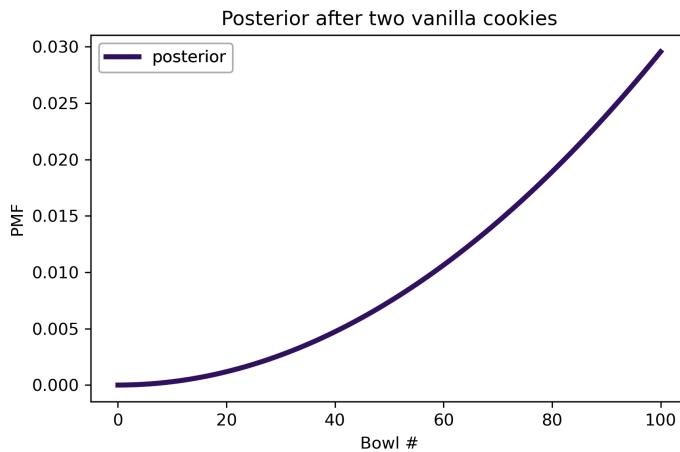


The posterior probability of Bowl 0 is 0 because it contains no vanilla cookies. The posterior probability of Bowl 100 is the highest because it contains the most vanilla cookies. In between, the shape of the posterior distribution is a line because the likelihoods are proportional to the bowl numbers.

Now suppose we put the cookie back, draw again from the same bowl, and get another vanilla cookie. Here's the update after the second cookie:

```
posterior2 = posterior1 * likelihood_vanilla  
posterior2.normalize()
```

And here's what the posterior distribution looks like:

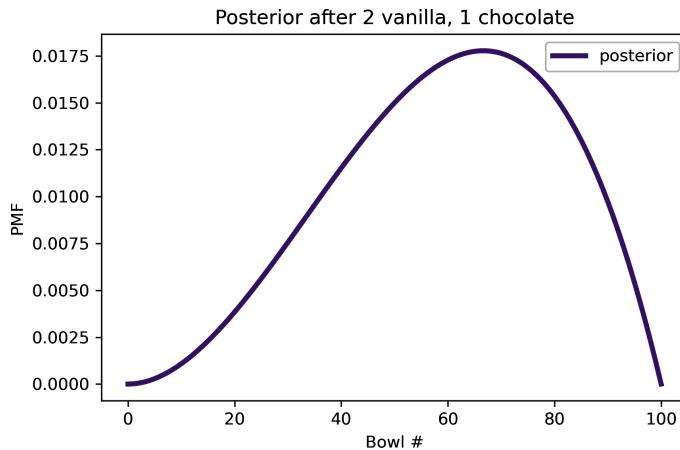


After two vanilla cookies, the high-numbered bowls have the highest posterior probabilities because they contain the most vanilla cookies; the low-numbered bowls have the lowest probabilities.

But suppose we draw again and get a chocolate cookie. Here's the update:

```
likelihood_chocolate = 1 - hypos/100  
  
posterior3 = posterior2 * likelihood_chocolate  
posterior3.normalize()
```

And here's the posterior distribution:



Now Bowl 100 has been eliminated because it contains no chocolate cookies. But the high-numbered bowls are still more likely than the low-numbered bowls, because we have seen more vanilla cookies than chocolate.

In fact, the peak of the posterior distribution is at Bowl 67, which corresponds to the fraction of vanilla cookies in the data we've observed, $2/3$.

The quantity with the highest posterior probability is called the **MAP**, which stands for “maximum a posteori probability”, where “a posteori” is unnecessary Latin for “posterior”.

To compute the MAP, we can use the `Series` method `idxmax`:

```
posterior3.idxmax()
```

```
67
```

Or `pmf` provides a more memorable name for the same thing:

```
posterior3.max_prob()
```

```
67
```

As you might suspect, this example isn't really about bowls; it's about estimating proportions. Imagine that you have one bowl of cookies. You don't know what fraction of cookies are vanilla, but you think it is equally likely to be any fraction from 0 to 1. If you draw three cookies and two are vanilla, what proportion of cookies in the bowl do you think are vanilla? The posterior distribution we just computed is the answer to that question.

We'll come back to estimating proportions in the next chapter. But first let's use a `pmf` to solve the Dice Problem.

The Dice Problem

In the previous chapter we solved the Dice Problem using a Bayes table. Here's the statement of the problem:

Suppose I have a box with a 6-sided die, an 8-sided die, and a 12-sided die.

I choose one of the dice at random, roll it, and report that the outcome is a 1.

What is the probability that I chose the 6-sided die?

Let's solve it using a `Pmf`. I'll use integers to represent the hypotheses:

```
hypos = [6, 8, 12]
```

We can make the prior distribution like this:

```
prior = Pmf(1/3, hypos)
prior
```

probs	
6	0.333333
8	0.333333
12	0.333333

As in the previous example, the prior probability gets broadcast across the hypotheses. The `Pmf` object has two attributes:

- `qs` contains the quantities in the distribution;
- `ps` contains the corresponding probabilities.

```
prior.qs
array([ 6,  8, 12])
prior.ps
array([0.33333333, 0.33333333, 0.33333333])
```

Now we're ready to do the update. Here's the likelihood of the data for each hypothesis:

```
likelihood1 = 1/6, 1/8, 1/12
```

And here's the update:

```
posterior = prior * likelihood1
posterior.normalize()
posterior
```

probs	
6	0.444444
8	0.333333
12	0.222222

The posterior probability for the 6-sided die is 4/9.

Now suppose I roll the same die again and get a 7. Here are the likelihoods:

```
likelihood2 = 0, 1/8, 1/12
```

The likelihood for the 6-sided die is 0 because it is not possible to get a 7 on a 6-sided die. The other two likelihoods are the same as in the previous update.

Here's the update:

```
posterior *= likelihood2
posterior.normalize()
posterior
```

probs	
6	0.000000
8	0.692308
12	0.307692

After rolling a 1 and a 7, the posterior probability of the 8-sided die is about 69%.

Updating Dice

The following function is a more general version of the update in the previous section:

```
def update_dice(pmf, data):
    """Update pmf based on new data."""
    hypos = pmf.qs
    likelihood = 1 / hypos
    impossible = (data > hypos)
    likelihood[impossible] = 0
    pmf *= likelihood
    pmf.normalize()
```

The first parameter is a `Pmf` that represents the possible dice and their probabilities. The second parameter is the outcome of rolling a die.

The first line selects quantities from the `Pmf` that represent the hypotheses. Since the hypotheses are integers, we can use them to compute the likelihoods. In general, if there are n sides on the die, the probability of any possible outcome is $1/n$.

However, we have to check for impossible outcomes! If the outcome exceeds the hypothetical number of sides on the die, the probability of that outcome is 0.

`impossible` is a Boolean `Series` that is `True` for each impossible outcome. I use it as an index into `likelihood` to set the corresponding probabilities to 0.

Finally, I multiply `pmf` by the likelihoods and normalize.

Here's how we can use this function to compute the updates in the previous section. I start with a fresh copy of the prior distribution:

```
pmf = prior.copy()  
pmf
```

probs	
6	0.333333
8	0.333333
12	0.333333

And use `update_dice` to do the updates:

```
update_dice(pmf, 1)  
update_dice(pmf, 7)  
pmf
```

probs	
6	0.000000
8	0.692308
12	0.307692

The result is the same. We will see a version of this function in the next chapter.

Summary

This chapter introduces the `empiricaldist` module, which provides `Pmf`, which we use to represent a set of hypotheses and their probabilities.

`empiricaldist` is based on pandas; the `Pmf` class inherits from the pandas `Series` class and provides additional features specific to probability mass functions. We'll use `Pmf` and other classes from `empiricaldist` throughout the book because they simplify the code and make it more readable. But we could do the same things directly with pandas.

We use a `Pmf` to solve the Cookie Problem and the Dice Problem, which we saw in the previous chapter. With a `Pmf` it is easy to perform sequential updates with multiple pieces of data.

We also solved a more general version of the Cookie Problem, with 101 bowls rather than two. Then we computed the MAP, which is the quantity with the highest posterior probability.

In the next chapter, I'll introduce the Euro Problem, and we will use the binomial distribution. And, at last, we will make the leap from using Bayes's theorem to doing Bayesian statistics.

But first you might want to work on the exercises.

Exercises

Exercise 3-1.

Suppose I have a box with a 6-sided die, an 8-sided die, and a 12-sided die. I choose one of the dice at random, roll it four times, and get 1, 3, 5, and 7. What is the probability that I chose the 8-sided die?

You can use the `update_dice` function or do the update yourself.

Exercise 3-2.

In the previous version of the Dice Problem, the prior probabilities are the same because the box contains one of each die. But suppose the box contains 1 die that is 4-sided, 2 dice that are 6-sided, 3 dice that are 8-sided, 4 dice that are 12-sided, and 5 dice that are 20-sided. I choose a die, roll it, and get a 7. What is the probability that I chose an 8-sided die?

Hint: To make the prior distribution, call `Pmf` with two parameters.

Exercise 3-3.

Suppose I have two sock drawers. One contains equal numbers of black and white socks. The other contains equal numbers of red, green, and blue socks. Suppose I choose a drawer at random, choose two socks at random, and I tell you that I got a matching pair. What is the probability that the socks are white?

For simplicity, let's assume that there are so many socks in both drawers that removing one sock makes a negligible change to the proportions.

Exercise 3-4.

Here's a problem from *Bayesian Data Analysis*:

Elvis Presley had a twin brother (who died at birth). What is the probability that Elvis was an identical twin?

Hint: In 1935, about 2/3 of twins were fraternal and 1/3 were identical.

CHAPTER 4

Estimating Proportions

In the previous chapter we solved the 101 Bowls Problem, and I admitted that it is not really about guessing which bowl the cookies came from; it is about estimating proportions.

In this chapter, we take another step toward Bayesian statistics by solving the Euro Problem. We'll start with the same prior distribution, and we'll see that the update is the same, mathematically. But I will argue that it is a different problem, philosophically, and use it to introduce two defining elements of Bayesian statistics: choosing prior distributions, and using probability to represent the unknown.

The Euro Problem

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

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?

To answer that question, we'll proceed in two steps. First we'll use the binomial distribution to see where that 7% came from; then we'll use Bayes's theorem to estimate the probability that this coin comes up heads.

The Binomial Distribution

Suppose I tell you that a coin is “fair”, that is, the probability of heads is 50%. If you spin it twice, there are four outcomes: HH, HT, TH, and TT. All four outcomes have the same probability, 25%.

If we add up the total number of heads, there are three possible results: 0, 1, or 2. The probabilities of 0 and 2 are 25%, and the probability of 1 is 50%.

More generally, suppose the probability of heads is p and we spin the coin n times. The probability that we get a total of k heads is given by the **binomial distribution**:

$$\binom{n}{k} p^k (1-p)^{n-k}$$

for any value of k from 0 to n , including both. The term $\binom{n}{k}$ is the **binomial coefficient**, usually pronounced “ n choose k ”.

We could evaluate this expression ourselves, but we can also use the SciPy function `binom.pmf`. For example, if we flip a coin $n=2$ times and the probability of heads is $p=0.5$, here’s the probability of getting $k=1$ heads:

```
from scipy.stats import binom

n = 2
p = 0.5
k = 1

binom.pmf(k, n, p)

0.5
```

Instead of providing a single value for k , we can also call `binom.pmf` with an array of values:

```
import numpy as np
ks = np.arange(n+1)

ps = binom.pmf(ks, n, p)
ps

array([0.25, 0.5 , 0.25])
```

The result is a NumPy array with the probability of 0, 1, or 2 heads. If we put these probabilities in a `Pmf`, the result is the distribution of k for the given values of n and p .

Here's what it looks like:

```
from empiricaldist import Pmf
```

```
pmf_k = Pmf(ps, ks)
```

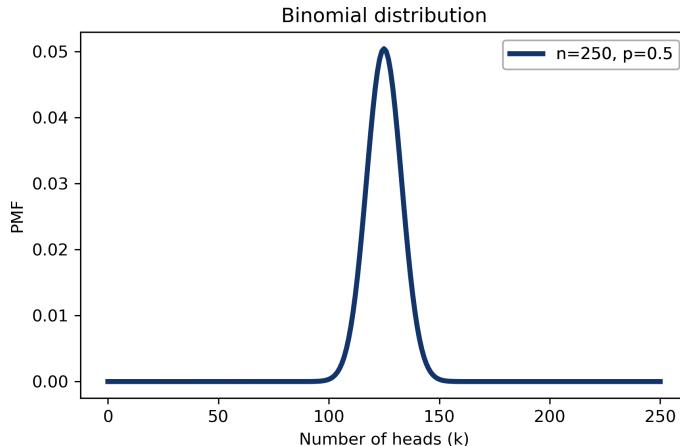
	probs
0	0.25
1	0.50
2	0.25

The following function computes the binomial distribution for given values of n and p and returns a Pmf that represents the result:

```
def make_binomial(n, p):
    """Make a binomial Pmf."""
    ks = np.arange(n+1)
    ps = binom.pmf(ks, n, p)
    return Pmf(ps, ks)
```

Here's what it looks like with $n=250$ and $p=0.5$:

```
pmf_k = make_binomial(n=250, p=0.5)
```



The most likely quantity in this distribution is 125:

```
pmf_k.max_prob()
```

```
125
```

But even though it is the most likely quantity, the probability that we get exactly 125 heads is only about 5%:

```
pmf_k[125]  
0.05041221314731537
```

In MacKay's example, we got 140 heads, which is even less likely than 125:

```
pmf_k[140]  
0.008357181724917673
```

In the article MacKay quotes, the statistician says, "If the coin were unbiased the chance of getting a result as extreme as that would be less than 7%."

We can use the binomial distribution to check his math. The following function takes a PMF and computes the total probability of quantities greater than or equal to `threshold`:

```
def prob_ge(pmf, threshold):  
    """Probability of quantities greater than threshold."""  
    ge = (pmf.qs >= threshold)  
    total = pmf[ge].sum()  
    return total
```

Here's the probability of getting 140 heads or more:

```
prob_ge(pmf_k, 140)  
0.033210575620022706
```

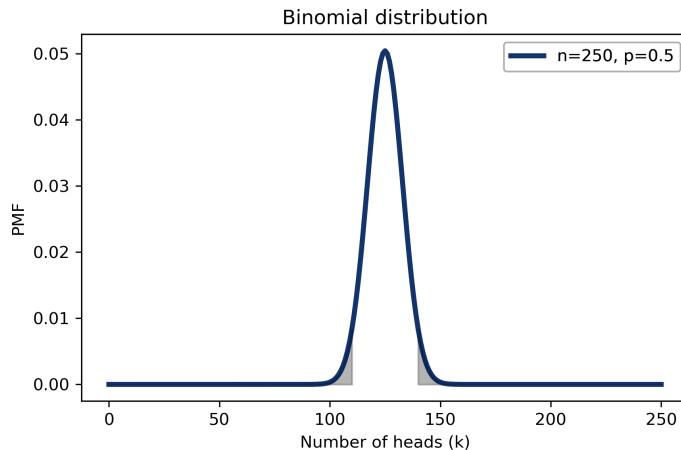
`Pmf` provides a method that does the same computation:

```
pmf_k.prob_ge(140)  
0.033210575620022706
```

The result is about 3.3%, which is less than the quoted 7%. The reason for the difference is that the statistician includes all outcomes "as extreme as" 140, which includes outcomes less than or equal to 110.

To see where that comes from, recall that the expected number of heads is 125. If we get 140, we've exceeded that expectation by 15. And if we get 110, we have come up short by 15.

7% is the sum of both of these “tails”, as shown in the following figure:



Here's how we compute the total probability of the left tail:

```
pmf_k.prob_le(110)  
0.033210575620022706
```

The probability of outcomes less than or equal to 110 is also 3.3%, so the total probability of outcomes “as extreme” as 140 is 6.6%.

The point of this calculation is that these extreme outcomes are unlikely if the coin is fair.

That's interesting, but it doesn't answer MacKay's question. Let's see if we can.

Bayesian Estimation

Any given coin has some probability of landing heads up when spun on edge; I'll call this probability x . It seems reasonable to believe that x depends on physical characteristics of the coin, like 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 .

For simplicity, I'll start with a uniform prior, which assumes that all values of x are equally likely. That might not be a reasonable assumption, so we'll come back and consider other priors later.

We can make a uniform prior like this:

```
hypos = np.linspace(0, 1, 101)
prior = Pmf(1, hypos)
```

hypos is an array of equally spaced values between 0 and 1.

We can use the hypotheses to compute the likelihoods, like this:

```
likelihood_heads = hypos
likelihood_tails = 1 - hypos
```

I'll put the likelihoods for heads and tails in a dictionary to make it easier to do the update:

```
likelihood = {
    'H': likelihood_heads,
    'T': likelihood_tails
}
```

To represent the data, I'll construct a string with H repeated 140 times and T repeated 110 times:

```
dataset = 'H' * 140 + 'T' * 110
```

The following function does the update:

```
def update_euro(pmf, dataset):
    """Update pmf with a given sequence of H and T."""
    for data in dataset:
        pmf *= likelihood[data]

    pmf.normalize()
```

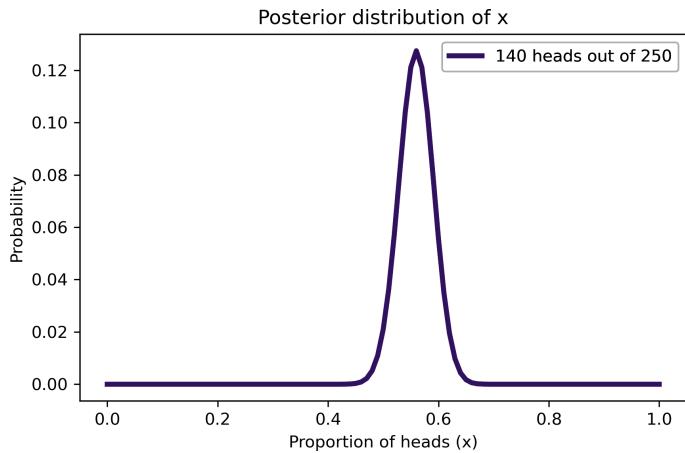
The first argument is a `Pmf` that represents the prior. The second argument is a sequence of strings. Each time through the loop, we multiply `pmf` by the likelihood of one outcome, H for heads or T for tails.

Notice that `normalize` is outside the loop, so the posterior distribution only gets normalized once, at the end. That's more efficient than normalizing it after each spin (although we'll see later that it can also cause problems with floating-point arithmetic).

Here's how we use `update_euro`:

```
posterior = prior.copy()
update_euro(posterior, dataset)
```

And here's what the posterior looks like:



This figure shows the posterior distribution of x , which is the proportion of heads for the coin we observed.

The posterior distribution represents our beliefs about x after seeing the data. It indicates that values less than 0.4 and greater than 0.7 are unlikely; values between 0.5 and 0.6 are the most likely.

In fact, the most likely value for x is 0.56, which is the proportion of heads in the dataset, 140/250.

```
posterior.max_prob()
```

```
0.56
```

Triangle Prior

So far we've been using a uniform prior:

```
uniform = Pmf(1, hypos, name='uniform')
uniform.normalize()
```

But that might not be a reasonable choice based on what we know about coins. I can believe that if a coin is lopsided, x might deviate substantially from 0.5, but it seems unlikely that the Belgian Euro coin is so imbalanced that x is 0.1 or 0.9.

It might be more reasonable to choose a prior that gives higher probability to values of x near 0.5 and lower probability to extreme values.

As an example, let's try a triangle-shaped prior. Here's the code that constructs it:

```
ramp_up = np.arange(50)
ramp_down = np.arange(50, -1, -1)

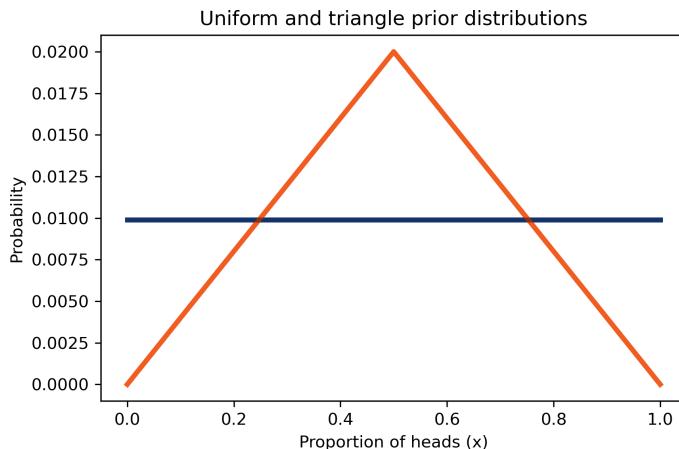
a = np.append(ramp_up, ramp_down)

triangle = Pmf(a, hypos, name='triangle')
triangle.normalize()

2500
```

`arange` returns a NumPy array, so we can use `np.append` to append `ramp_down` to the end of `ramp_up`. Then we use `a` and `hypos` to make a `Pmf`.

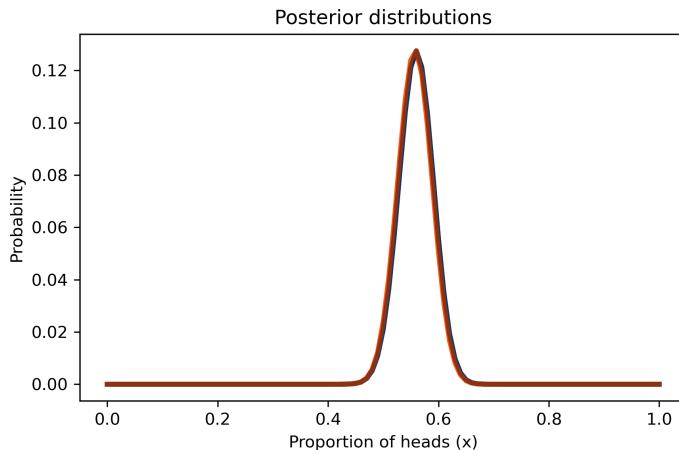
The following figure shows the result, along with the uniform prior:



Now we can update both priors with the same data:

```
update_euro(uniform, dataset)
update_euro(triangle, dataset)
```

Here are the posteriors:



The differences between the posterior distributions are barely visible, and so small they would hardly matter in practice.

And that's good news. To see why, imagine two people who disagree angrily about which prior is better, uniform or triangle. Each of them has reasons for their preference, but neither of them can persuade the other to change their mind.

But suppose they agree to use the data to update their beliefs. When they compare their posterior distributions, they find that there is almost nothing left to argue about.

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 distribution.

The Binomial Likelihood Function

So far we've been computing the updates one spin at a time, so for the Euro Problem we have to do 250 updates.

A more efficient alternative is to compute the likelihood of the entire dataset at once. For each hypothetical value of x , we have to compute the probability of getting 140 heads out of 250 spins.

Well, we know how to do that; this is the question the binomial distribution answers. If the probability of heads is p , the probability of k heads in n spins is:

$$\binom{n}{k} p^k (1-p)^{n-k}$$

And we can use SciPy to compute it. The following function takes a `Pmf` that represents a prior distribution and a tuple of integers that represent the data:

```
from scipy.stats import binom

def update_binomial(pmf, data):
    """Update pmf using the binomial distribution."""
    k, n = data
    xs = pmf.qs
    likelihood = binom.pmf(k, n, xs)
    pmf *= likelihood
    pmf.normalize()
```

The data are represented with a tuple of values for `k` and `n`, rather than a long string of outcomes. Here's the update:

```
uniform2 = Pmf(1, hypos, name='uniform2')
data = 140, 250
update_binomial(uniform2, data)
```

We can use `allclose` to confirm that the result is the same as in the previous section except for a small floating-point round-off.

```
np.allclose(uniform, uniform2)
True
```

But this way of doing the computation is much more efficient.

Bayesian Statistics

You might have noticed similarities between the Euro Problem and the 101 Bowls Problem in “[101 Bowls](#)” on page 34. The prior distributions are the same, the likelihoods are the same, and with the same data, the results would be the same. But there are two differences.

The first is the choice of the prior. With 101 bowls, the uniform prior is implied by the statement of the problem, which says that we choose one of the bowls at random with equal probability.

In the Euro Problem, the choice of the prior is subjective; that is, reasonable people could disagree, maybe because they have different information about coins or because they interpret the same information differently.

Because the priors are subjective, the posteriors are subjective, too. And some people find that problematic.

The other difference is the nature of what we are estimating. In the 101 Bowls Problem, we choose the bowl randomly, so it is uncontroversial to compute the probability of choosing each bowl. In the Euro Problem, the proportion of heads is a physical

property of a given coin. Under some interpretations of probability, that's a problem because physical properties are not considered random.

As an example, consider the age of the universe. Currently, our best estimate is 13.80 billion years, but it might be off by **0.02 billion years in either direction**.

Now suppose we would like to know the probability that the age of the universe is actually greater than 13.81 billion years. Under some interpretations of probability, we would not be able to answer that question. We would be required to say something like, "The age of the universe is not a random quantity, so it has no probability of exceeding a particular value."

Under the Bayesian interpretation of probability, it is meaningful and useful to treat physical quantities as if they were random and compute probabilities about them.

In the Euro Problem, the prior distribution represents what we believe about coins in general and the posterior distribution represents what we believe about a particular coin after seeing the data. So we can use the posterior distribution to compute probabilities about the coin and its proportion of heads.

The subjectivity of the prior and the interpretation of the posterior are key differences between using Bayes's theorem and doing Bayesian statistics.

Bayes's theorem is a mathematical law of probability; no reasonable person objects to it. But Bayesian statistics is surprisingly controversial. Historically, many people have been bothered by its subjectivity and its use of probability for things that are not random.

If you are interested in this history, I recommend Sharon Bertsch McGrayne's book, *The Theory That Would Not Die*.

Summary

In this chapter I posed David MacKay's Euro Problem and we started to solve it. Given the data, we computed the posterior distribution for x , the probability a Euro coin comes up heads.

We tried two different priors, updated them with the same data, and found that the posteriors were nearly the same. This is good news, because it suggests that if two people start with different beliefs and see the same data, their beliefs tend to converge.

This chapter introduces the binomial distribution, which we used to compute the posterior distribution more efficiently. And I discussed the differences between applying Bayes's theorem, as in the 101 Bowls Problem, and doing Bayesian statistics, as in the Euro Problem.

However, we still haven't answered MacKay's question: "Do these data give evidence that the coin is biased rather than fair?" I'm going to leave this question hanging a little longer; we'll come back to it in [Chapter 10](#).

In the next chapter, we'll solve problems related to counting, including trains, tanks, and rabbits.

But first you might want to work on these exercises.

Exercises

Exercise 4-1.

In Major League Baseball (MLB), most players have a batting average between .200 and .330, which means that their probability of getting a hit is between 0.2 and 0.33.

Suppose a player appearing in their first game gets 3 hits out of 3 attempts. What is the posterior distribution for their probability of getting a hit?

Exercise 4-2.

Whenever you survey people about sensitive issues, you have to deal with [social desirability bias](#), which is the tendency of people to adjust their answers to show themselves in the most positive light. One way to improve the accuracy of the results is [randomized response](#).

As an example, suppose you want to know how many people cheat on their taxes. If you ask them directly, it is likely that some of the cheaters will lie. You can get a more accurate estimate if you ask them indirectly, like this: Ask each person to flip a coin and, without revealing the outcome,

- If they get heads, they report YES.
- If they get tails, they honestly answer the question, "Do you cheat on your taxes?"

If someone says YES, we don't know whether they actually cheat on their taxes; they might have flipped heads. Knowing this, people might be more willing to answer honestly.

Suppose you survey 100 people this way and get 80 YESes and 20 NOs. Based on this data, what is the posterior distribution for the fraction of people who cheat on their taxes? What is the most likely quantity in the posterior distribution?

Exercise 4-3.

Suppose you want to test whether a coin is fair, but you don't want to spin it hundreds of times. So you make a machine that spins the coin automatically and uses computer vision to determine the outcome.

However, you discover that the machine is not always accurate. Specifically, suppose the probability is $y=0.2$ that an actual heads is reported as tails, or actual tails reported as heads.

If we spin a coin 250 times and the machine reports 140 heads, what is the posterior distribution of x ? What happens as you vary the value of y ?

Exercise 4-4.

In preparation for an alien invasion, the Earth Defense League (EDL) has been working on new missiles to shoot down space invaders. Of course, some missile designs are better than others; let's assume that each design has some probability of hitting an alien ship, x .

Based on previous tests, the distribution of x in the population of designs is approximately uniform between 0.1 and 0.4.

Now suppose the new ultra-secret Alien Blaster 9000 is being tested. In a press conference, an EDL general reports that the new design has been tested twice, taking two shots during each test. The results of the test are confidential, so the general won't say how many targets were hit, but they report: "The same number of targets were hit in the two tests, so we have reason to think this new design is consistent."

Is this data good or bad? That is, does it increase or decrease your estimate of x for the Alien Blaster 9000?

CHAPTER 5

Estimating Counts

In the previous chapter we solved problems that involve estimating proportions. In the Euro Problem, we estimated the probability that a coin lands heads up, and in the exercises, you estimated a batting average, the fraction of people who cheat on their taxes, and the chance of shooting down an invading alien.

Clearly, some of these problems are more realistic than others, and some are more useful than others.

In this chapter, we'll work on problems related to counting, or estimating the size of a population. Again, some of the examples will seem silly, but some of them, like the German Tank Problem, have real applications, sometimes in life and death situations.

The Train Problem

I found the Train Problem in Frederick Mosteller's *Fifty Challenging Problems in Probability with Solutions*:

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, so we'll start with something simple and then consider alternatives. Let's assume that N is equally likely to be any value from 1 to 1000.

Here's the prior distribution:

```
import numpy as np
from empiricaldist import Pmf

hypos = np.arange(1, 1001)
prior = Pmf(1, hypos)
```

Now let's figure out the likelihood of the data. In a hypothetical fleet of N locomotives, what is the probability that we would see number 60? If we assume that we are equally likely to see any locomotive, the chance of seeing any particular one is $1/N$.

Here's the function that does the update:

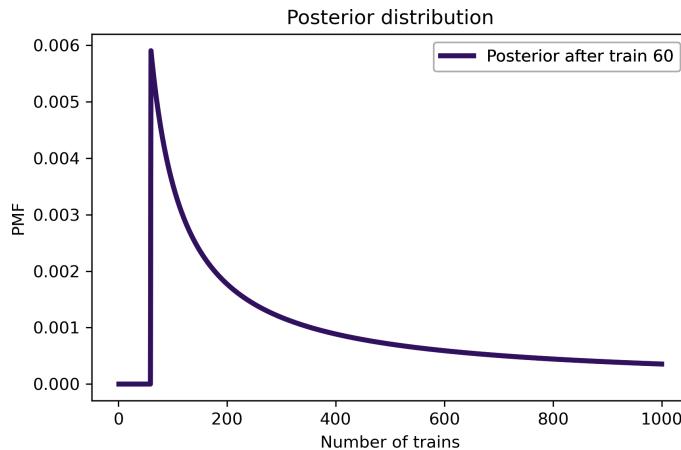
```
def update_train(pmf, data):
    """Update pmf based on new data."""
    hypos = pmf.qs
    likelihood = 1 / hypos
    impossible = (data > hypos)
    likelihood[impossible] = 0
    pmf *= likelihood
    pmf.normalize()
```

This function might look familiar; it is the same as the update function for the Dice Problem in the previous chapter. In terms of likelihood, the Train Problem is the same as the Dice Problem.

Here's the update:

```
data = 60
posterior = prior.copy()
update_train(posterior, data)
```

Here's what the posterior looks like:



Not surprisingly, all values of N below 60 have been eliminated.

The most likely value, if you had to guess, is 60.

```
posterior.max_prob()
```

```
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 alternative is to compute the mean of the posterior distribution. Given a set of possible quantities, q_i , and their probabilities, p_i , the mean of the distribution is:

$$\text{mean} = \sum_i p_i q_i$$

Which we can compute like this:

```
np.sum(posterior.ps * posterior.qs)
```

```
333.41989326370776
```

Or we can use the method provided by `pmf`:

```
posterior.mean()
```

```
333.41989326370776
```

The mean of the posterior is 333, so that might be a good guess if you want to minimize error. If you played this guessing game over and over, using the mean of the posterior as your estimate would minimize the **mean squared error** over the long run.

Sensitivity to the Prior

The prior I used in the previous section is uniform from 1 to 1000, but I offered no justification for choosing a uniform distribution or that particular upper bound. We might wonder whether the posterior distribution is sensitive to the prior. With so little data—only one observation—it is.

This table shows what happens as we vary the upper bound:

Posterior mean	
Upper bound	
500	207.079228
1000	333.419893
2000	552.179017

As we vary the upper bound, the posterior mean changes substantially. So that's bad.

When the posterior is sensitive to the prior, there are two ways to proceed:

- Get more data.
- Get more background information and choose a better prior.

With more data, 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.

Here's how the posterior means depend on the upper bound of the prior, when we observe three trains:

Posterior mean	
Upper bound	
500	151.849588
1000	164.305586
2000	171.338181

The differences are smaller, but apparently three trains are not enough for the posteriors to converge.

Power Law 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 train-operating company with 1,000 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*.

This law suggests that if there are 1,000 companies with fewer than 10 locomotives, there might be 100 companies with 100 locomotives, 10 companies with 1,000, and possibly one company with 10,000 locomotives.

Mathematically, a power law means that the number of companies with a given size, N , is proportional to $(1/N)^\alpha$, where α is a parameter that is often near 1.

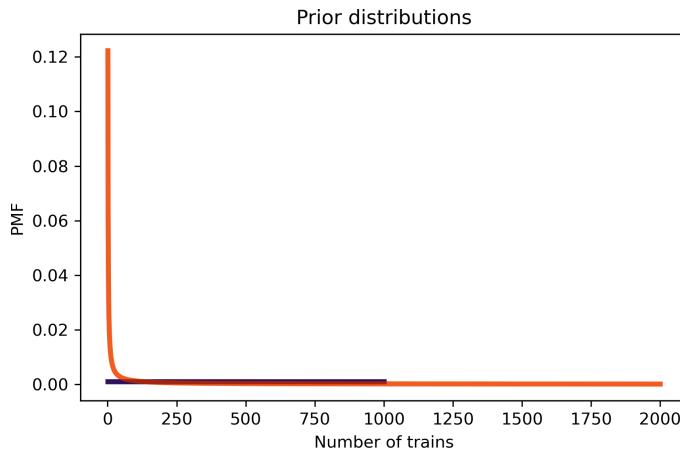
We can construct a power law prior like this:

```
alpha = 1.0
ps = hypos**(-alpha)
power = Pmf(ps, hypos, name='power law')
power.normalize()
```

For comparison, here's the uniform prior again:

```
hypos = np.arange(1, 1001)
uniform = Pmf(1, hypos, name='uniform')
uniform.normalize()
1000
```

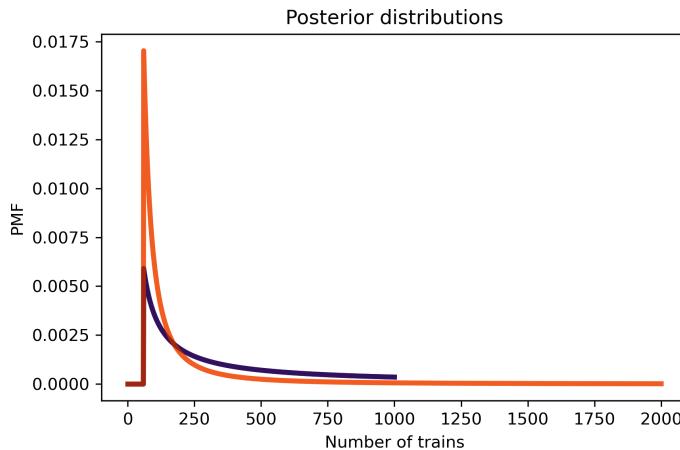
Here's what a power law prior looks like, compared to the uniform prior:



Here's the update for both priors:

```
dataset = [60]
update_train(uniform, dataset)
update_train(power, dataset)
```

And here are the posterior distributions:



The power law gives less prior probability to high values, which yields lower posterior means, and less sensitivity to the upper bound.

Here's how the posterior means depend on the upper bound when we use a power law prior and observe three trains:

Posterior mean	
Upper bound	
500	130.708470
1000	133.275231
2000	133.997463

Now the differences are 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.

Credible Intervals

So far we have seen two ways to summarize a posterior distribution: the value with the highest posterior probability (the MAP) and the posterior mean. These are both **point estimates**, that is, single values that estimate the quantity we are interested in.

Another way to summarize a posterior distribution is with percentiles. If you have taken a standardized test, you might be familiar with percentiles. For example, if your score is the 90th percentile, that means you did as well as or better than 90% of the people who took the test.

If we are given a value, x , we can compute its **percentile rank** by finding all values less than or equal to x and adding up their probabilities.

Pmf provides a method that does this computation. So, for example, we can compute the probability that the company has less than or equal to 100 trains:

```
power.prob_le(100)
0.2937469222495771
```

With a power law prior and a dataset of three trains, the result is about 29%. So 100 trains is the 29th percentile.

Going the other way, suppose we want to compute a particular percentile; for example, the median of a distribution is the 50th percentile. We can compute it by adding up probabilities until the total exceeds 0.5. Here's a function that does it:

```
def quantile(pmf, prob):
    """Compute a quantile with the given prob."""
    total = 0
    for q, p in pmf.items():
        total += p
        if total >= prob:
            return q
    return np.nan
```

The loop uses `items`, which iterates the quantities and probabilities in the distribution. Inside the loop we add up the probabilities of the quantities in order. When the total equals or exceeds `prob`, we return the corresponding quantity.

This function is called `quantile` because it computes a quantile rather than a percentile. The difference is the way we specify `prob`. If `prob` is a percentage between 0 and 100, we call the corresponding quantity a percentile. If `prob` is a probability between 0 and 1, we call the corresponding quantity a **quantile**.

Here's how we can use this function to compute the 50th percentile of the posterior distribution:

```
quantile(power, 0.5)
```

113

The result, 113 trains, is the median of the posterior distribution.

`Pmf` provides a method called `quantile` that does the same thing. We can call it like this to compute the 5th and 95th percentiles:

```
power.quantile([0.05, 0.95])  
array([ 91., 243.])
```

The result is the interval from 91 to 243 trains, which implies:

- The probability is 5% that the number of trains is less than or equal to 91.
- The probability is 5% that the number of trains is greater than 243.

Therefore the probability is 90% that the number of trains falls between 91 and 243 (excluding 91 and including 243). For this reason, this interval is called a 90% **credible interval**.

`Pmf` also provides `credible_interval`, which computes an interval that contains the given probability.

```
power.credible_interval(0.9)  
array([ 91., 243.])
```

The German Tank Problem

During World War II, the Economic Warfare Division of the American Embassy in London used statistical analysis to estimate German production of tanks and other equipment.

The Western Allies had captured log books, inventories, and repair records that included chassis and engine serial numbers for individual tanks.

Analysis of these records indicated that serial numbers were allocated by manufacturer and tank type in blocks of 100 numbers, that numbers in each block were used sequentially, and that not all numbers in each block were used. So the problem of estimating German tank production could be reduced, within each block of 100 numbers, to a form of the Train Problem.

Based on this insight, American and British analysts produced estimates substantially lower than estimates from other forms of intelligence. And after the war, records indicated that they were substantially more accurate.

They performed similar analyses for tires, trucks, rockets, and other equipment, yielding accurate and actionable economic intelligence.

The German Tank Problem is historically interesting; it is also a nice example of real-world application of statistical estimation.

For more on this problem, see [this Wikipedia page](#) and Ruggles and Brodie, “An Empirical Approach to Economic Intelligence in World War II”, *Journal of the American Statistical Association*, March 1947, [available in the CIA’s online reading room](#).

Informative Priors

Among Bayesians, there are two approaches to choosing prior distributions. Some recommend choosing the prior that best represents background information about the problem; in that case the prior is said to be **informative**. The problem with using an informative prior is that people might have different information or interpret it differently. So informative priors might seem arbitrary.

The alternative is a so-called **uninformative prior**, which is intended to be as unrestricted as possible, to let the data speak for itself. In some cases you can identify a unique prior that has some desirable property, like representing minimal prior information about the estimated quantity.

Uninformative priors are appealing because they seem more objective. But I am generally in favor of using informative priors. Why? First, Bayesian analysis is always based on modeling decisions. Choosing the prior is one of those decisions, but it is not the only one, and it might not even be the most subjective. So even if an uninformative prior is more objective, the entire analysis is still subjective.

Also, for most practical problems, you are likely to be in one of two situations: either you have a lot of data or not very much. If you have a lot of data, the choice of the prior doesn’t matter; informative and uninformative priors yield almost the same results. If you don’t have much data, using relevant background information (like the power law distribution) makes a big difference.

And if, as in the German Tank Problem, you have to make life and death decisions based on your results, you should probably use all of the information at your disposal, rather than maintaining the illusion of objectivity by pretending to know less than you do.

Summary

This chapter introduced the Train Problem, which turns out to have the same likelihood function as the Dice Problem, and which can be applied to the German Tank Problem. In all of these examples, the goal is to estimate a count, or the size of a population.

In the next chapter, I'll introduce “odds” as an alternative to probabilities, and Bayes's rule as an alternative form of Bayes's theorem. We'll compute distributions of sums and products, and use them to estimate the number of members of Congress who are corrupt, among other problems.

But first, you might want to work on these exercises.

Exercises

Exercise 5-1.

Suppose you are giving a talk in a large lecture hall and the fire marshal interrupts because they think the audience exceeds 1,200 people, which is the safe capacity of the room.

You think there are fewer than 1,200 people, and you offer to prove it. It would take too long to count, so you try an experiment:

- You ask how many people were born on May 11 and two people raise their hands.
- You ask how many were born on May 23 and 1 person raises their hand.
- Finally, you ask how many were born on August 1, and no one raises their hand.

How many people are in the audience? What is the probability that there are more than 1,200 people? Hint: Remember the binomial distribution.

Exercise 5-2.

I often see **rabbits** in the garden behind my house, but it's not easy to tell them apart, so I don't really know how many there are.

Suppose I deploy a motion-sensing **camera trap** that takes a picture of the first rabbit it sees each day. After three days, I compare the pictures and conclude that two of them are the same rabbit and the other is different.

How many rabbits visit my garden?

To answer this question, we have to think about the prior distribution and the likelihood of the data:

- I have sometimes seen four rabbits at the same time, so I know there are at least that many. I would be surprised if there were more than 10. So, at least as a starting place, I think a uniform prior from 4 to 10 is reasonable.
- To keep things simple, let's assume that all rabbits who visit my garden are equally likely to be caught by the camera trap in a given day. Let's also assume it is guaranteed that the camera trap gets a picture every day.

Exercise 5-3.

Suppose that in the criminal justice system, all prison sentences are either 1, 2, or 3 years, with an equal number of each. One day, you visit a prison and choose a prisoner at random. What is the probability that they are serving a 3-year sentence? What is the average remaining sentence of the prisoners you observe?

Exercise 5-4.

If I chose a random adult in the US, what is the probability that they have a sibling? To be precise, what is the probability that their mother has had at least one other child?

This article from the Pew Research Center provides some relevant data.

Exercise 5-5.

The **Doomsday argument** is “a probabilistic argument that claims to predict the number of future members of the human species given an estimate of the total number of humans born so far.”

Suppose there are only two kinds of intelligent civilizations that can happen in the universe. The “short-lived” kind go extinct after only 200 billion individuals are born. The “long-lived” kind survive until 2,000 billion individuals are born. And suppose that the two kinds of civilization are equally likely. Which kind of civilization do you think we live in?

The Doomsday argument says we can use the total number of humans born so far as data. According to the [Population Reference Bureau](#), the total number of people who have ever lived is about 108 billion.

Since you were born quite recently, let's assume that you are, in fact, human being number 108 billion. If N is the total number who will ever live and we consider you to be a randomly-chosen person, it is equally likely that you could have been person 1, or N , or any number in between. So what is the probability that you would be number 108 billion?

Given this data and dubious prior, what is the probability that our civilization will be short-lived?

CHAPTER 6

Odds and Addends

This chapter presents a new way to represent a degree of certainty, **odds**, and a new form of Bayes's theorem, called **Bayes's rule**. Bayes's rule is convenient if you want to do a Bayesian update on paper or in your head. It also sheds light on the important idea of **evidence** and how we can quantify the strength of evidence.

The second part of the chapter is about “addends”, that is, quantities being added, and how we can compute their distributions. We'll define functions that compute the distribution of sums, differences, products, and other operations. Then we'll use those distributions as part of a Bayesian update.

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”, but you might not know what that means. The **odds in favor** of an event are the ratio of the probability it will occur to the probability that it will not.

The following function does this calculation:

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

For example, if my team has a 75% chance of winning, the odds in their favor are three to one, because the chance of winning is three times the chance of losing:

```
odds(0.75)
```

```
3.0
```

You can write odds in decimal form, but it is also common to write them as a ratio of integers. So “three to one” is sometimes 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 my horse has a 10% chance of winning, the odds in favor are 1:9:

```
odds(0.1)
```

```
0.1111111111111112
```

But in that case it would be more common to say that the odds against are 9:1:

```
odds(0.9)
```

```
9.000000000000002
```

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

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

For example, if the odds are 3/2, the corresponding probability is 3/5:

```
prob(3/2)
```

```
0.6
```

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

```
def prob2(yes, no):
    return yes / (yes + no)

prob2(3, 2)
0.6
```

Probabilities and odds are different representations of the same information; given either one, you can compute the other. But some computations are easier when we work with odds, as we'll see in the next section, and some computations are even easier with log odds, which we'll see later.

Bayes's Rule

So far we have worked with Bayes's theorem in the “probability form”:

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

Writing $\text{odds}(A)$ for odds in favor of A , we can express Bayes's theorem in “odds form”:

$$\text{odds}(A|D) = \text{odds}(A) \frac{P(D|A)}{P(D|B)}$$

This is Bayes's rule, which says that the posterior odds are the prior odds times the likelihood ratio. Bayes's rule is 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. 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.

```
prior_odds = 1
likelihood_ratio = (3/4) / (1/2)
post_odds = prior_odds * likelihood_ratio
post_odds

1.5

post_prob = prob(post_odds)
post_prob

0.6
```

If we draw another cookie and it's chocolate, we can do another update:

```
likelihood_ratio = (1/4) / (1/2)
post_odds *= likelihood_ratio
post_odds

0.75
```

And convert back to probability:

```
post_prob = prob(post_odds)
post_prob

0.42857142857142855
```

Oliver's Blood

I'll use Bayes's rule to solve 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, which corresponds to probability 50%. The posterior odds are $3/2$, or probability 60%. So the vanilla cookie is evidence in favor of Bowl 1.

Bayes's rule provides a way to make this intuition more precise. Again:

$$\text{odds}(A|D) = \text{odds}(A) \frac{P(D|A)}{P(D|B)}$$

Dividing through by $\text{odds}(A)$, we get:

$$\frac{\text{odds}(A|D)}{\text{odds}(A)} = \frac{P(D|A)}{P(D|B)}$$

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 is greater than 1, that means that the data were more likely under A than under B . And 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 A than under B , so the odds in favor of A go down.

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

Let's apply that to the problem at hand. If Oliver is one of the people who left blood at the crime scene, he accounts for the 'O' sample; in that case, the probability of the data is 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, 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 might have 'O' and the second 'AB',
- Or the first person might have 'AB' and the second 'O'.

The probability of either combination is $(0.6)(0.01)$, which is 0.6%, so the total probability is twice that, or 1.2%. So the data are a little more likely if Oliver is *not* one of the people who left blood at the scene.

We can use these probabilities to compute the likelihood ratio:

```
like1 = 0.01
like2 = 2 * 0.6 * 0.01

likelihood_ratio = like1 / like2
likelihood_ratio

0.8333333333333334
```

Since the likelihood ratio is less than 1, the blood tests are evidence *against* the hypothesis that Oliver left blood at the scene.

But it is weak evidence. For example, if the prior odds were 1 (that is, 50% probability), the posterior odds would be 0.83, which corresponds to a probability of 45%:

```
post_odds = 1 * like1 / like2
prob(post_odds)

0.45454545454545453
```

So this evidence doesn't "move the needle" very much.

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

If this result still 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 unexplained. If Oliver doesn't account for the 'O' blood, we have two chances to find someone in the population with 'AB' blood. And that factor of two makes the difference.

Exercise 6-1.

Suppose that based on other evidence, your prior belief in Oliver's guilt is 90%. How much would the blood evidence in this section change your beliefs? What if you initially thought there was only a 10% chance of his guilt?

Addends

The second half of this chapter is about distributions of sums and results of other operations. We'll start with a Forward Problem, where we are given the inputs and compute the distribution of the output. Then we'll work on Inverse Problems, where we are given the outputs and we compute the distribution of the inputs.

As a first example, suppose you roll two dice and add them up. What is the distribution of the sum? I'll use the following function to create a `Pmf` that represents the possible outcomes of a die:

```

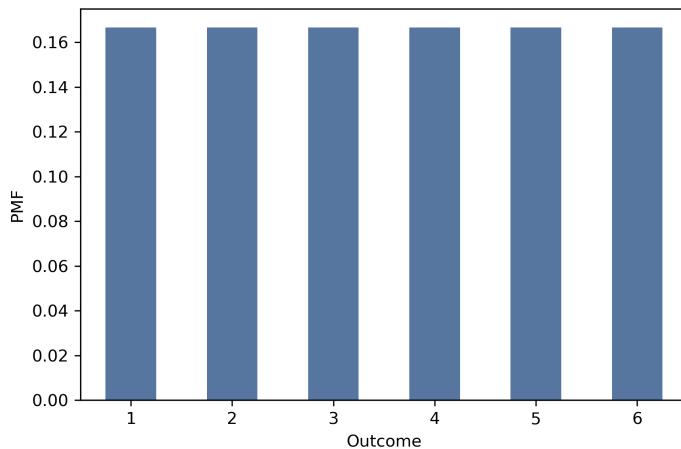
import numpy as np
from empiricaldist import Pmf

def make_die(sides):
    outcomes = np.arange(1, sides+1)
    die = Pmf(1/sides, outcomes)
    return die

```

On a 6-sided die, the outcomes are 1 through 6, all equally likely.

```
die = make_die(6)
```



If we roll two dice and add them up, there are 11 possible outcomes, 2 through 12, but they are not equally likely. To compute the distribution of the sum, we have to enumerate the possible outcomes.

And that's how this function works:

```

def add_dist(pmf1, pmf2):
    """Compute the distribution of a sum."""
    res = Pmf()
    for q1, p1 in pmf1.items():
        for q2, p2 in pmf2.items():
            q = q1 + q2
            p = p1 * p2
            res[q] = res[q] + p
    return res

```

The parameters are `Pmf` objects representing distributions.

The loops iterate through the quantities and probabilities in the `Pmf` objects. Each time through the loop `q` gets the sum of a pair of quantities, and `p` gets the probability of the pair. Because the same sum might appear more than once, we have to add up the total probability for each sum.

Notice a subtle element of this line:

```
res[q] = res(q) + p
```

I use parentheses on the right side of the assignment, which returns 0 if `q` does not appear yet in `res`. I use brackets on the left side of the assignment to create or update an element in `res`; using parentheses on the left side would not work.

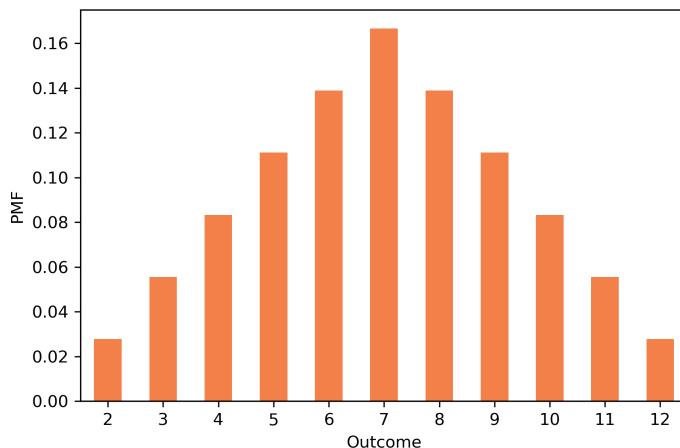
`Pmf` provides `add_dist`, which does the same thing. You can call it as a method, like this:

```
twice = die.add_dist(die)
```

Or as a function, like this:

```
twice = Pmf.add_dist(die, die)
```

And here's what the result looks like:



If we have a sequence of `Pmf` objects that represent dice, we can compute the distribution of the sum like this:

```
def add_dist_seq(seq):
    """Compute Pmf of the sum of values from seq."""
    total = seq[0]
    for other in seq[1:]:
        total = total.add_dist(other)
    return total
```

As an example, we can make a list of three dice like this:

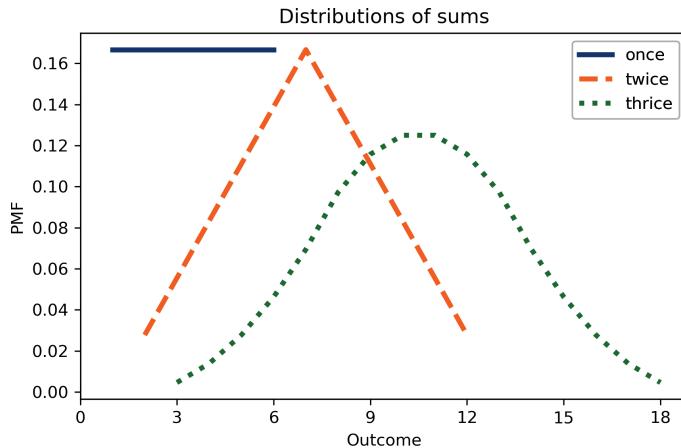
```
dice = [die] * 3
```

And we can compute the distribution of their sum like this:

```
thrice = add_dist_seq(dice)
```

The following figure shows what these three distributions look like:

- The distribution of a single die is uniform from 1 to 6.
- The sum of two dice has a triangle distribution between 2 and 12.
- The sum of three dice has a bell-shaped distribution between 3 and 18.



As an aside, this example demonstrates the Central Limit Theorem, which says that the distribution of a sum converges on a bell-shaped normal distribution, at least under some conditions.

Gluten Sensitivity

In 2015 I read a paper that tested whether people diagnosed with gluten sensitivity (but not celiac disease) were able to distinguish gluten flour from non-gluten flour in a blind challenge ([you can read the paper here](#)).

Out of 35 subjects, 12 correctly identified the gluten flour based on resumption of symptoms while they were eating it. Another 17 wrongly identified the gluten-free flour based on their symptoms, and 6 were unable to distinguish.

The authors conclude, “Double-blind gluten challenge induces symptom recurrence in just one-third of patients.”

This conclusion seems odd to me, because if none of the patients were sensitive to gluten, we would expect some of them to identify the gluten flour by chance. So here’s the question: based on this data, how many of the subjects are sensitive to gluten and how many are guessing?

We can use Bayes's theorem to answer this question, but first we have to make some modeling decisions. I'll assume:

- People who are sensitive to gluten have a 95% chance of correctly identifying gluten flour under the challenge conditions, and
- People who are not sensitive have a 40% chance of identifying the gluten flour by chance (and a 60% chance of either choosing the other flour or failing to distinguish).

These particular values are arbitrary, but the results are not sensitive to these choices.

I will solve this problem in two steps. First, assuming that we know how many subjects are sensitive, I will compute the distribution of the data. Then, using the likelihood of the data, I will compute the posterior distribution of the number of sensitive patients.

The first is the **Forward Problem**; the second is the **Inverse Problem**.

The Forward Problem

Suppose we know that 10 of the 35 subjects are sensitive to gluten. That means that 25 are not:

```
n = 35
num_sensitive = 10
num_insensitive = n - num_sensitive
```

Each sensitive subject has a 95% chance of identifying the gluten flour, so the number of correct identifications follows a binomial distribution.

I'll use `make_binomial`, which we defined in “[The Binomial Distribution](#)” on page 44, to make a `Pmf` that represents the binomial distribution:

```
from utils import make_binomial

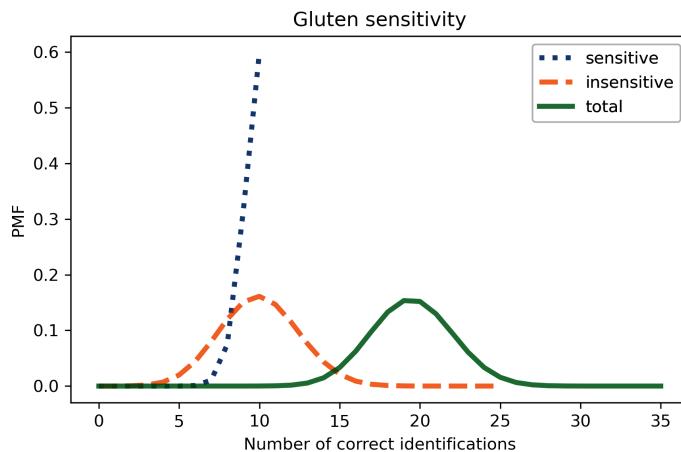
dist_sensitive = make_binomial(num_sensitive, 0.95)
dist_insensitive = make_binomial(num_insensitive, 0.40)
```

The results are the distributions for the number of correct identifications in each group.

Now we can use `add_dist` to compute the distribution of the total number of correct identifications:

```
dist_total = Pmf.add_dist(dist_sensitive, dist_insensitive)
```

Here are the results:



We expect most of the sensitive subjects to identify the gluten flour correctly. Of the 25 insensitive subjects, we expect about 10 to identify the gluten flour by chance. So we expect about 20 correct identifications in total.

This is the answer to the Forward Problem: given the number of sensitive subjects, we can compute the distribution of the data.

The Inverse Problem

Now let's solve the Inverse Problem: given the data, we'll compute the posterior distribution of the number of sensitive subjects.

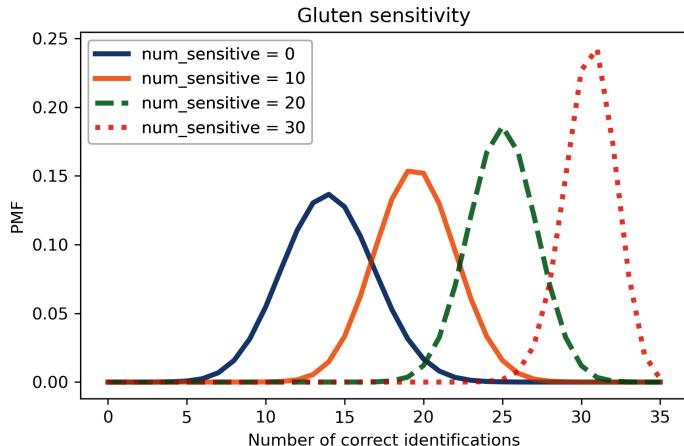
Here's how. I'll loop through the possible values of `num_sensitive` and compute the distribution of the data for each:

```
import pandas as pd

table = pd.DataFrame()
for num_sensitive in range(0, n+1):
    num_insensitive = n - num_sensitive
    dist_sensitive = make_binomial(num_sensitive, 0.95)
    dist_insensitive = make_binomial(num_insensitive, 0.4)
    dist_total = Pmf.add_dist(dist_sensitive, dist_insensitive)
    table[num_sensitive] = dist_total
```

The loop enumerates the possible values of `num_sensitive`. For each value, it computes the distribution of the total number of correct identifications, and stores the result as a column in a pandas DataFrame.

The following figure shows selected columns from the `DataFrame`, corresponding to different hypothetical values of `num_sensitive`:



Now we can use this table to compute the likelihood of the data:

```
likelihood1 = table.loc[12]
```

`loc` selects a row from the `DataFrame`. The row with index 12 contains the probability of 12 correct identifications for each hypothetical value of `num_sensitive`. And that's exactly the likelihood we need to do a Bayesian update.

I'll use a uniform prior, which implies that I would be equally surprised by any value of `num_sensitive`:

```
hypos = np.arange(n+1)
prior = Pmf(1, hypos)
```

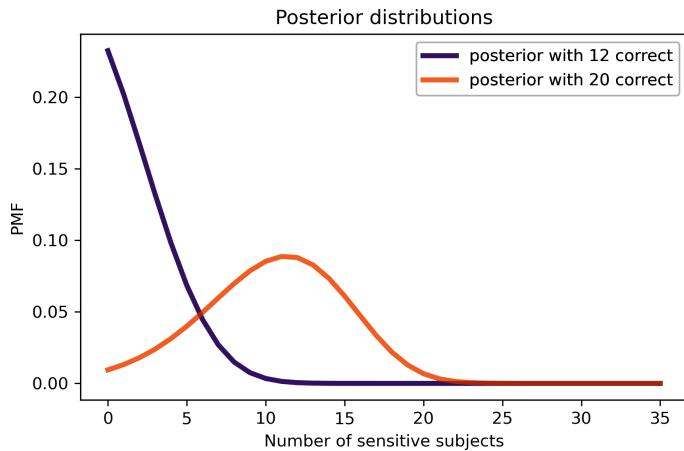
And here's the update:

```
posterior1 = prior * likelihood1
posterior1.normalize()
```

For comparison, I also compute the posterior for another possible outcome, 20 correct identifications:

```
likelihood2 = table.loc[20]
posterior2 = prior * likelihood2
posterior2.normalize()
```

The following figure shows posterior distributions of `num_sensitive` based on the actual data, 12 correct identifications, and the other possible outcome, 20 correct identifications.



With 12 correct identifications, the most likely conclusion is that none of the subjects are sensitive to gluten. If there had been 20 correct identifications, the most likely conclusion would be that 11-12 of the subjects were sensitive.

```
posterior1.max_prob()
0
posterior2.max_prob()
11
```

Summary

This chapter presents two topics that are almost unrelated except that they make the title of the chapter catchy.

The first part of the chapter is about Bayes's rule, evidence, and how we can quantify the strength of evidence using a likelihood ratio or Bayes factor.

The second part is about `add_dist`, which computes the distribution of a sum. We can use this function to solve Forward and Inverse Problems; that is, given the parameters of a system, we can compute the distribution of the data or, given the data, we can compute the distribution of the parameters.

In the next chapter, we'll compute distributions for minimums and maximums, and use them to solve more Bayesian problems. But first you might want to work on these exercises.

More Exercises

Exercise 6-2.

Let's use Bayes's rule to solve the Elvis problem from [Chapter 3](#):

Elvis Presley had a twin brother who died at birth. What is the probability that Elvis was an identical twin?

In 1935, about 2/3 of twins were fraternal and 1/3 were identical. The question contains two pieces of information we can use to update this prior.

- First, Elvis's twin was also male, which is more likely if they were identical twins, with a likelihood ratio of 2.
- Also, Elvis's twin died at birth, which is more likely if they were identical twins, with a likelihood ratio of 1.25.

If you are curious about where those numbers come from, I wrote [a blog post about it](#).

Exercise 6-3.

The following is an [interview question that appeared on glassdoor.com](#), attributed to Facebook:

You're about to get on a plane to Seattle. You want to know if you should bring an umbrella. You call 3 random friends of yours who live there and ask each independently if it's raining. Each of your friends has a 2/3 chance of telling you the truth and a 1/3 chance of messing with you by lying. All 3 friends tell you that "Yes" it is raining. What is the probability that it's actually raining in Seattle?

Use Bayes's rule to solve this problem. As a prior you can assume that it rains in Seattle about 10% of the time.

This question causes some confusion about the differences between Bayesian and frequentist interpretations of probability; if you are curious about this point, [I wrote a blog article about it](#).

Exercise 6-4.

[According to the CDC](#), people who smoke are about 25 times more likely to develop lung cancer than nonsmokers.

[Also according to the CDC](#), about 14% of adults in the US are smokers. If you learn that someone has lung cancer, what is the probability they are a smoker?

Exercise 6-5.

In *Dungeons & Dragons*, the amount of damage a goblin can withstand is the sum of two 6-sided dice. The amount of damage you inflict with a short sword is determined by rolling one 6-sided die. A goblin is defeated if the total damage you inflict is greater than or equal to the amount it can withstand.

Suppose you are fighting a goblin and you have already inflicted 3 points of damage. What is your probability of defeating the goblin with your next successful attack?

Hint: You can use `Pmf.add_dist` to add a constant amount, like 3, to a `Pmf` and `Pmf.sub_dist` to compute the distribution of remaining points.

Exercise 6-6.

Suppose I have a box with a 6-sided die, an 8-sided die, and a 12-sided die. I choose one of the dice at random, roll it twice, multiply the outcomes, and report that the product is 12. What is the probability that I chose the 8-sided die?

Hint: `Pmf` provides a function called `mul_dist` that takes two `Pmf` objects and returns a `Pmf` that represents the distribution of the product.

Exercise 6-7.

Betrayal at House on the Hill is a strategy game in which characters with different attributes explore a haunted house. Depending on their attributes, the characters roll different numbers of dice. For example, if attempting a task that depends on knowledge, Professor Longfellow rolls 5 dice, Madame Zistra rolls 4, and Ox Bellows rolls 3. Each die yields 0, 1, or 2 with equal probability.

If a randomly chosen character attempts a task three times and rolls a total of 3 on the first attempt, 4 on the second, and 5 on the third, which character do you think it was?

Exercise 6-8.

There are 538 members of the United States Congress.

Suppose we audit their investment portfolios and find that 312 of them outperform the market. Let's assume that an honest member of Congress has only a 50% chance of outperforming the market, but a dishonest member who trades on inside information has a 90% chance. How many members of Congress are honest?

Minimum, Maximum, and Mixture

In the previous chapter we computed distributions of sums. In this chapter, we'll compute distributions of minimums and maximums, and use them to solve both Forward and Inverse Problems.

Then we'll look at distributions that are mixtures of other distributions, which will turn out to be particularly useful for making predictions.

But we'll start with a powerful tool for working with distributions, the cumulative distribution function.

Cumulative Distribution Functions

So far we have been using probability mass functions to represent distributions. A useful alternative is the **cumulative distribution function**, or CDF.

As an example, I'll use the posterior distribution from the Euro Problem, which we computed in “[Bayesian Estimation](#)” on page 47.

Here's the uniform prior we started with:

```
import numpy as np
from empiricaldist import Pmf

hypos = np.linspace(0, 1, 101)
pmf = Pmf(1, hypos)
data = 140, 250
```

And here's the update:

```

from scipy.stats import binom

def update_binomial(pmf, data):
    """Update pmf using the binomial distribution."""
    k, n = data
    xs = pmf.qs
    likelihood = binom.pmf(k, n, xs)
    pmf *= likelihood
    pmf.normalize()

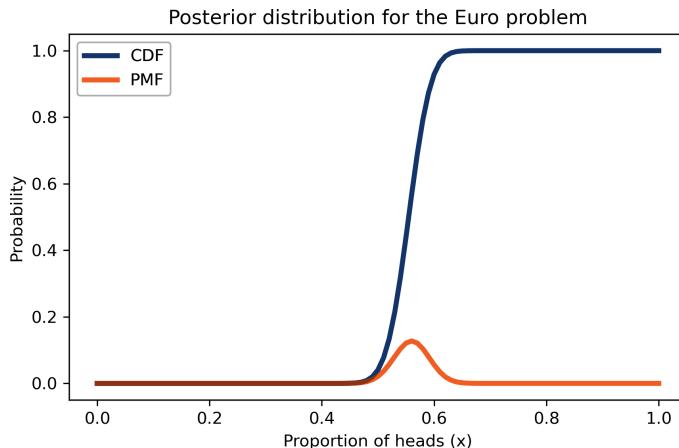
update_binomial(pmf, data)

```

The CDF is the cumulative sum of the PMF, so we can compute it like this:

```
cumulative = pmf.cumsum()
```

Here's what it looks like, along with the PMF:



The range of the CDF is always from 0 to 1, in contrast with the PMF, where the maximum can be any probability.

The result from `cumsum` is a pandas `Series`, so we can use the bracket operator to select an element:

```

cumulative[0.61]
0.9638303193984253

```

The result is about 0.96, which means that the total probability of all quantities less than or equal to 0.61 is 96%.

To go the other way—to look up a probability and get the corresponding quantile—we can use interpolation:

```
from scipy.interpolate import interp1d

ps = cumulative.values
qs = cumulative.index

interp = interp1d(ps, qs)
interp(0.96)

array(0.60890171)
```

The result is about 0.61, so that confirms that the 96th percentile of this distribution is 0.61.

`empiricaldist` provides a class called `Cdf` that represents a cumulative distribution function. Given a `Pmf`, you can compute a `Cdf` like this:

```
cdf = pmf.make_cdf()

make_cdf uses np.cumsum to compute the cumulative sum of the probabilities.

You can use brackets to select an element from a Cdf:
```

```
cdf[0.61]
0.9638303193984253
```

But if you look up a quantity that's not in the distribution, you get a `KeyError`.

To avoid this problem, you can call a `Cdf` as a function, using parentheses. If the argument does not appear in the `Cdf`, it interpolates between quantities.

```
cdf(0.615)
array(0.96383032)
```

Going the other way, you can use `quantile` to look up a cumulative probability and get the corresponding quantity:

```
cdf.quantile(0.9638303)
array(0.61)
```

`Cdf` also provides `credible_interval`, which computes a credible interval that contains the given probability:

```
cdf.credible_interval(0.9)
array([0.51, 0.61])
```

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. Given a Cdf, you can get the equivalent Pmf like this:

```
pmf = cdf.make_pmf()
```

`make_pmf` uses `np.diff` to compute differences between consecutive cumulative probabilities.

One reason Cdf objects are useful is that they compute quantiles efficiently. Another is that they make it easy to compute the distribution of a maximum or minimum, as we'll see in the next section.

Best Three of Four

In *Dungeons & Dragons*, each character has six attributes: strength, intelligence, wisdom, dexterity, constitution, and charisma.

To generate a new character, players roll four 6-sided dice for each attribute and add up the best three. For example, if I roll for strength and get 1, 2, 3, 4 on the dice, my character's strength would be the sum of 2, 3, and 4, which is 9.

As an exercise, let's figure out the distribution of these attributes. Then, for each character, we'll figure out the distribution of their best attribute.

I'll import two functions from the previous chapter: `make_die`, which makes a Pmf that represents the outcome of rolling a die, and `add_dist_seq`, which takes a sequence of Pmf objects and computes the distribution of their sum.

Here's a Pmf that represents a 6-sided die and a sequence with three references to it:

```
from utils import make_die

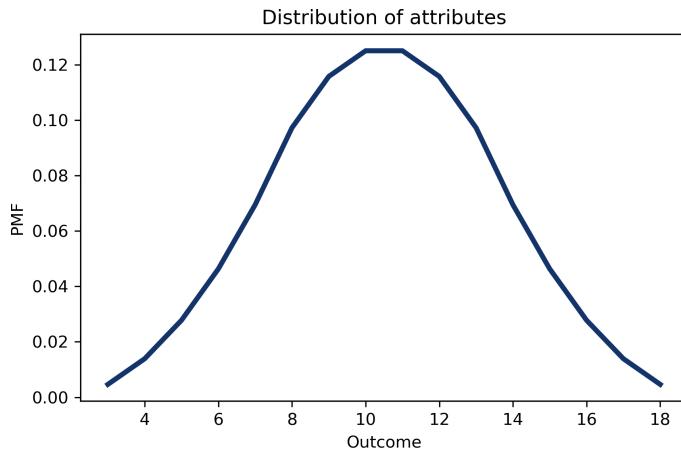
die = make_die(6)
dice = [die] * 3
```

And here's the distribution of the sum of three dice:

```
from utils import add_dist_seq

pmf_3d6 = add_dist_seq(dice)
```

Here's what it looks like:



If we roll four dice and add up the best three, computing the distribution of the sum is a bit more complicated. I'll estimate the distribution by simulating 10,000 rolls.

First I'll create an array of random values from 1 to 6, with 10,000 rows and 4 columns:

```
n = 10000
a = np.random.randint(1, 7, size=(n, 4))
```

To find the best three outcomes in each row, I'll use `sort` with `axis=1`, which sorts the rows in ascending order:

```
a.sort(axis=1)
```

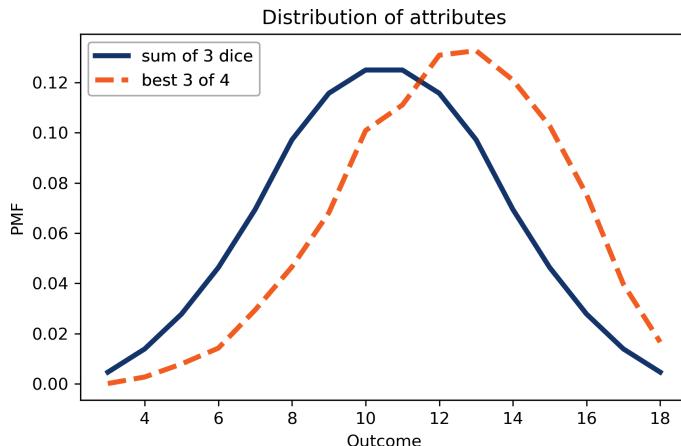
Finally, I'll select the last three columns and add them up:

```
t = a[:, 1:].sum(axis=1)
```

Now `t` is an array with a single column and 10,000 rows. We can compute the PMF of the values in `t` like this:

```
pmf_best3 = Pmf.from_seq(t)
```

The following figure shows the distribution of the sum of three dice, `pmf_3d6`, and the distribution of the best three out of four, `pmf_best3`:



As you might expect, choosing the best three out of four tends to yield higher values.

Next we'll find the distribution for the maximum of six attributes, each the sum of the best three of four dice.

Maximum

To compute the distribution of a maximum or minimum, we can make good use of the cumulative distribution function. First, I'll compute the `Cdf` of the best three of four distribution:

```
cdf_best3 = pmf_best3.make_cdf()
```

Recall that `Cdf(x)` is the sum of probabilities for quantities less than or equal to x . Equivalently, it is the probability that a random value chosen from the distribution is less than or equal to x .

Now suppose I draw 6 values from this distribution. The probability that all 6 of them are less than or equal to x is `Cdf(x)` raised to the 6th power, which we can compute like this:

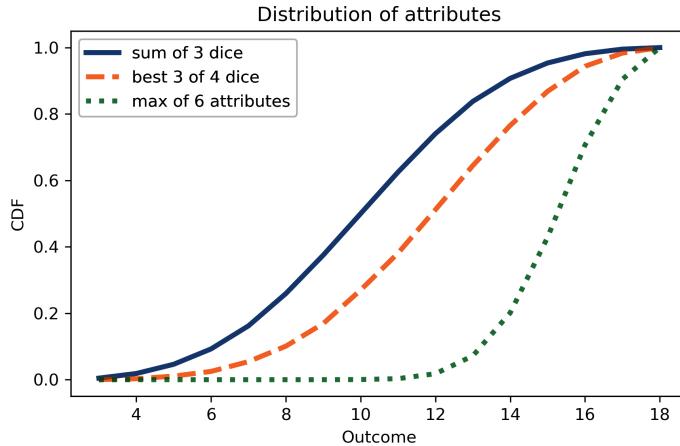
```
cdf_best3**6
```

If all 6 values are less than or equal to x , that means that their maximum is less than or equal to x . So the result is the CDF of their maximum. We can convert it to a `Cdf` object, like this:

```
from empiricaldist import Cdf

cdf_max6 = Cdf(cdf_best3**6)
```

The following figure shows the CDFs for the three distributions we have computed.



`Cdf` provides `max_dist`, which does the same computation, so we can also compute the `Cdf` of the maximum like this:

```
cdf_max_dist6 = cdf_best3.max_dist(6)
```

In the next section we'll find the distribution of the minimum. The process is similar, but a little more complicated. See if you can figure it out before you go on.

Minimum

In the previous section we computed the distribution of a character's best attribute. Now let's compute the distribution of the worst.

To compute the distribution of the minimum, we'll use the **complementary CDF**, which we can compute like this:

```
prob_gt = 1 - cdf_best3
```

As the variable name suggests, the complementary CDF is the probability that a value from the distribution is greater than x . If we draw 6 values from the distribution, the probability that all 6 exceed x is:

```
prob_gt6 = prob_gt**6
```

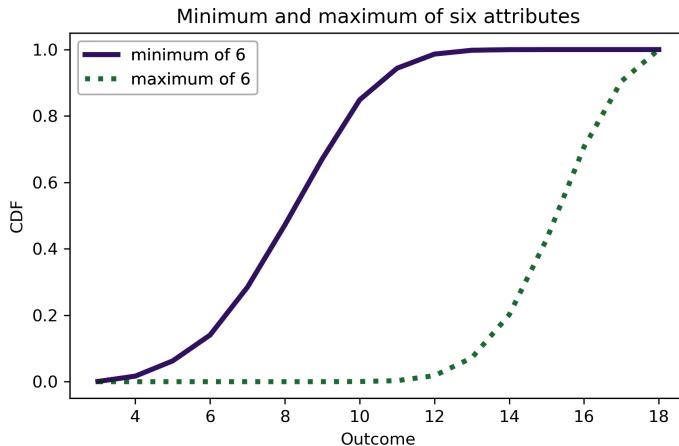
If all 6 exceed x , that means their minimum exceeds x , so `prob_gt6` is the complementary CDF of the minimum. And that means we can compute the CDF of the minimum like this:

```
prob_le6 = 1 - prob_gt6
```

The result is a pandas `Series` that represents the CDF of the minimum of six attributes. We can put those values in a `Cdf` object like this:

```
cdf_min6 = Cdf(prob_le6)
```

Here's what it looks like, along with the distribution of the maximum:



`Cdf` provides `min_dist`, which does the same computation, so we can also compute the `Cdf` of the minimum like this:

```
cdf_min_dist6 = cdf_best3.min_dist(6)
```

And we can confirm that the differences are small:

```
np.allclose(cdf_min_dist6, cdf_min6)
```

```
True
```

In the exercises at the end of this chapter, you'll use distributions of the minimum and maximum to do Bayesian inference. But first we'll see what happens when we mix distributions.

Mixture

In this section I'll show how we can compute a distribution that is a mixture of other distributions. I'll explain what that means with some simple examples; then, more usefully, we'll see how these mixtures are used to make predictions.

Here's another example inspired by *Dungeons & Dragons*:

- Suppose your character is armed with a dagger in one hand and a short sword in the other.
- During each round, you attack a monster with one of your two weapons, chosen at random.
- The dagger causes one 4-sided die of damage; the short sword causes one 6-sided die of damage.

What is the distribution of damage you inflict in each round?

To answer this question, I'll make a Pmf to represent the 4-sided and 6-sided dice:

```
d4 = make_die(4)
d6 = make_die(6)
```

Now, let's compute the probability you inflict 1 point of damage.

- If you attacked with the dagger, it's 1/4.
- If you attacked with the short sword, it's 1/6.

Because the probability of choosing either weapon is 1/2, the total probability is the average:

```
prob_1 = (d4(1) + d6(1)) / 2
prob_1
0.2083333333333331
```

For the outcomes 2, 3, and 4, the probability is the same, but for 5 and 6, it's different, because those outcomes are impossible with the 4-sided die.

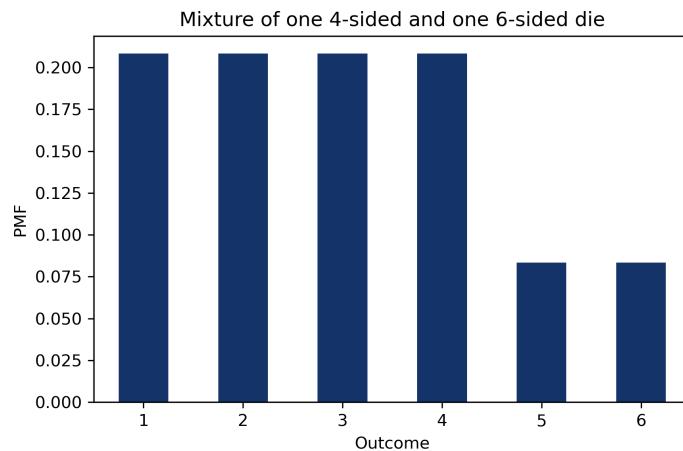
```
prob_6 = (d4(6) + d6(6)) / 2
prob_6
0.0833333333333333
```

To compute the distribution of the mixture, we could loop through the possible outcomes and compute their probabilities.

But we can do the same computation using the + operator:

```
mix1 = (d4 + d6) / 2
```

Here's what the mixture of these distributions looks like:



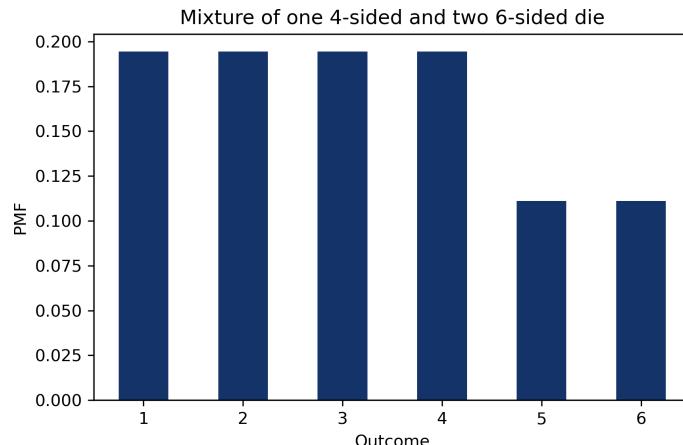
Now suppose you are fighting three monsters:

- One has a club, which causes one 4-sided die of damage.
- One has a mace, which causes one 6-sided die.
- And one has a quarterstaff, which also causes one 6-sided die.

Because the melee is disorganized, you are attacked by one of these monsters each round, chosen at random. To find the distribution of the damage they inflict, we can compute a weighted average of the distributions, like this:

```
mix2 = (d4 + 2*d6) / 3
```

This distribution is a mixture of one 4-sided die and two 6-sided dice. Here's what it looks like:

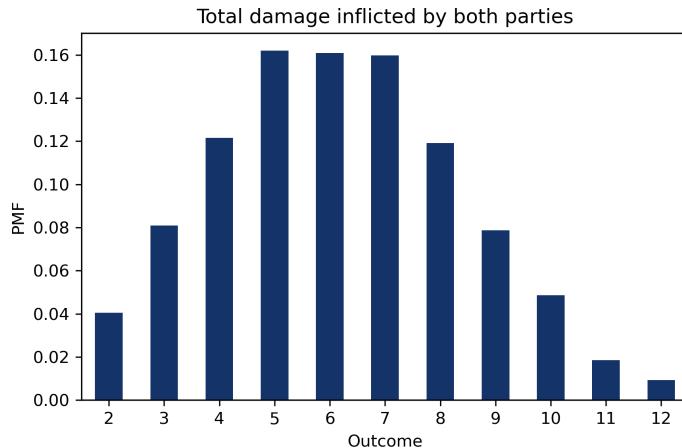


In this section we used the `+` operator, which adds the probabilities in the distributions, not to be confused with `Pmf.add_dist`, which computes the distribution of the sum of the distributions.

To demonstrate the difference, I'll use `Pmf.add_dist` to compute the distribution of the total damage done per round, which is the sum of the two mixtures:

```
total_damage = Pmf.add_dist(mix1, mix2)
```

And here's what it looks like:



General Mixtures

In the previous section we computed mixtures in an *ad hoc* way. Now we'll see a more general solution. In future chapters, we'll use this solution to generate predictions for real-world problems, not just role-playing games. But if you'll bear with me, we'll continue the previous example for one more section.

Suppose three more monsters join the combat, each of them with a battle axe that causes one 8-sided die of damage. Still, only one monster attacks per round, chosen at random, so the damage they inflict is a mixture of:

- One 4-sided die,
- Two 6-sided dice, and
- Three 8-sided dice.

I'll use a `Pmf` to represent a randomly chosen monster:

```
hypos = [4,6,8]
counts = [1,2,3]
pmf_dice = Pmf(counts, hypos)
pmf_dice.normalize()
pmf_dice
```

`probs`

4	0.166667
6	0.333333
8	0.500000

This distribution represents the number of sides on the die we'll roll and the probability of rolling each one. For example, one of the six monsters has a dagger, so the probability is 1/6 that we roll a 4-sided die.

Next I'll make a sequence of `Pmf` objects to represent the dice:

```
dice = [make_die(sides) for sides in hypos]
```

To compute the distribution of the mixture, I'll compute the weighted average of the dice, using the probabilities in `pmf_dice` as the weights.

To express this computation concisely, it is convenient to put the distributions into a pandas `DataFrame`:

```
import pandas as pd

pd.DataFrame(dice)
```

1	2	3	4	5	6	7	8
0.250000	0.250000	0.250000	0.250000	NaN	NaN	NaN	NaN
0.166667	0.166667	0.166667	0.166667	0.166667	0.166667	NaN	NaN
0.125000	0.125000	0.125000	0.125000	0.125000	0.125000	0.125	0.125

The result is a `DataFrame` with one row for each distribution and one column for each possible outcome. Not all rows are the same length, so pandas fills the extra spaces with the special value `NaN`, which stands for “not a number”. We can use `fillna` to replace the `NaN` values with 0:

```
pd.DataFrame(dice).fillna(0)
```

1	2	3	4	5	6	7	8
0.250000	0.250000	0.250000	0.250000	0.000000	0.000000	0.000	0.000
0.166667	0.166667	0.166667	0.166667	0.166667	0.166667	0.000	0.000
0.125000	0.125000	0.125000	0.125000	0.125000	0.125000	0.125	0.125

The next step is to multiply each row by the probabilities in `pmf_dice`, which turns out to be easier if we transpose the matrix so the distributions run down the columns rather than across the rows:

```
df = pd.DataFrame(dice).fillna(0).transpose()
```

Now we can multiply by the probabilities in `pmf_dice`:

```
df *= pmf_dice.ps  
df
```

1	0.041667	0.055556	0.0625
2	0.041667	0.055556	0.0625
3	0.041667	0.055556	0.0625
4	0.041667	0.055556	0.0625
5	0.000000	0.055556	0.0625
6	0.000000	0.055556	0.0625
7	0.000000	0.000000	0.0625
8	0.000000	0.000000	0.0625

And add up the weighted distributions:

```
df.sum(axis=1)
```

The argument `axis=1` means we want to sum across the rows. The result is a pandas `Series`.

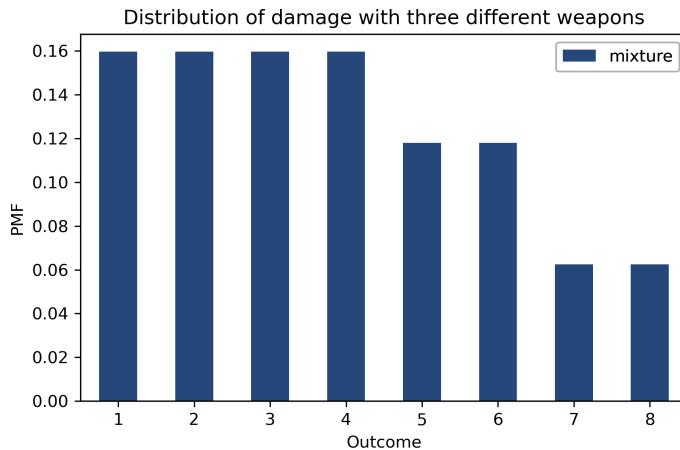
Putting it all together, here's a function that makes a weighted mixture of distributions:

```
def make_mixture(pmf, pmf_seq):  
    """Make a mixture of distributions."""  
    df = pd.DataFrame(pmf_seq).fillna(0).transpose()  
    df *= np.array(pmf)  
    total = df.sum(axis=1)  
    return Pmf(total)
```

The first parameter is a `Pmf` that maps from each hypothesis to a probability. The second parameter is a sequence of `Pmf` objects, one for each hypothesis. We can call it like this:

```
mix = make_mixture(pmf_dice, dice)
```

And here's what it looks like:



In this section I used pandas so that `make_mixture` is concise, efficient, and hopefully not too hard to understand. In the exercises at the end of the chapter, you'll have a chance to practice with mixtures, and we will use `make_mixture` again in the next chapter.

Summary

This chapter introduces the `Cdf` object, which represents the cumulative distribution function (CDF).

A `PMF` and the corresponding `Cdf` are equivalent in the sense that they contain the same information, so you can convert from one to the other. The primary difference between them is performance: some operations are faster and easier with a `PMF`; others are faster with a `Cdf`.

In this chapter we used `Cdf` objects to compute distributions of maximums and minimums; these distributions are useful for inference if we are given a maximum or minimum as data. You will see some examples in the exercises, and in future chapters. We also computed mixtures of distributions, which we will use in the next chapter to make predictions.

But first you might want to work on these exercises.

Exercises

Exercise 7-1.

When you generate a Dungeons & Dragons character, instead of rolling dice, you can use the “standard array” of attributes, which is 15, 14, 13, 12, 10, and 8. Do you think you are better off using the standard array or (literally) rolling the dice?

Compare the distribution of the values in the standard array to the distribution we computed for the best three out of four:

- Which distribution has higher mean? Use the `mean` method.
- Which distribution has higher standard deviation? Use the `std` method.
- The lowest value in the standard array is 8. For each attribute, what is the probability of getting a value less than 8? If you roll the dice six times, what's the probability that at least one of your attributes is less than 8?
- The highest value in the standard array is 15. For each attribute, what is the probability of getting a value greater than 15? If you roll the dice six times, what's the probability that at least one of your attributes is greater than 15?

Exercise 7-2.

Suppose you are fighting three monsters:

- One is armed with a short sword that causes one 6-sided die of damage,
- One is armed with a battle axe that causes one 8-sided die of damage, and
- One is armed with a bastard sword that causes one 10-sided die of damage.

One of the monsters, chosen at random, attacks you and does 1 point of damage.

Which monster do you think it was? Compute the posterior probability that each monster was the attacker.

If the same monster attacks you again, what is the probability that you suffer 6 points of damage?

Hint: Compute a posterior distribution as we have done before and pass it as one of the arguments to `make_mixture`.

Exercise 7-3.

Henri Poincaré was a French mathematician who taught at the Sorbonne around 1900. The following anecdote about him is probably fiction, but it makes an interesting probability problem.

Supposedly Poincaré suspected that his local bakery was selling loaves of bread that were lighter than the advertised weight of 1 kg, so every day for a year he bought a loaf of bread, brought it home and weighed it. At the end of the year, he plotted the distribution of his measurements and showed that it fit a normal distribution with mean 950 g and standard deviation 50 g. He brought this evidence to the bread police, who gave the baker a warning.

For the next year, Poincaré continued to weigh his bread every day. At the end of the year, he found that the average weight was 1000 g, just as it should be, but again he complained to the bread police, and this time they fined the baker.

Why? Because the shape of the new distribution was asymmetric. Unlike the normal distribution, it was skewed to the right, which is consistent with the hypothesis that the baker was still making 950 g loaves, but deliberately giving Poincaré the heavier ones.

To see whether this anecdote is plausible, let's suppose that when the baker sees Poincaré coming, he hefts n loaves of bread and gives Poincaré the heaviest one. How many loaves would the baker have to heft to make the average of the maximum 1000 g?

CHAPTER 8

Poisson Processes

This chapter introduces the [Poisson process](#), which is a model used to describe events that occur at random intervals. As an example of a Poisson process, we'll model goal-scoring in soccer, which is American English for the game everyone else calls "football". We'll use goals scored in a game to estimate the parameter of a Poisson process; then we'll use the posterior distribution to make predictions.

And we'll solve the World Cup Problem.

The World Cup Problem

In the 2018 FIFA World Cup final, France defeated Croatia 4 goals to 2. Based on this outcome:

1. How confident should we be that France is the better team?
2. If the same teams played again, what is the chance France would win again?

To answer these questions, we have to make some modeling decisions.

- First, I'll assume that for any team against another team there is some unknown goal-scoring rate, measured in goals per game, which I'll denote with the Python variable `lam` or the Greek letter λ , pronounced "lambda".
- Second, I'll assume that a goal is equally likely during any minute of a game. So, in a 90-minute game, the probability of scoring during any minute is $\lambda/90$.
- Third, I'll assume that a team never scores twice during the same minute.

Of course, none of these assumptions is completely true in the real world, but I think they are reasonable simplifications. As George Box said, "All models are wrong; some are useful" (<https://oreil.ly/oeTQU>).

In this case, the model is useful because if these assumptions are true, at least roughly, the number of goals scored in a game follows a Poisson distribution, at least roughly.

The Poisson Distribution

If the number of goals scored in a game follows a **Poisson distribution** with a goal-scoring rate, λ , the probability of scoring k goals is

$$\lambda^k \exp(-\lambda) / k!$$

for any non-negative value of k .

SciPy provides a `poisson` object that represents a Poisson distribution. We can create one with $\lambda = 1.4$ like this:

```
from scipy.stats import poisson

lam = 1.4
dist = poisson(lam)
type(dist)

scipy.stats._distn_infrastructure.rv_frozen
```

The result is an object that represents a “frozen” random variable and provides `pmf`, which evaluates the probability mass function of the Poisson distribution.

```
k = 4
dist.pmf(k)

0.039471954028253146
```

This result implies that if the average goal-scoring rate is 1.4 goals per game, the probability of scoring 4 goals in a game is about 4%.

We'll use the following function to make a `PMF` that represents a Poisson distribution:

```
from empiricaldist import PMF

def make_poisson_pmf(lam, qs):
    """Make a PMF of a Poisson distribution."""
    ps = poisson(lam).pmf(qs)
    pmf = PMF(ps, qs)
    pmf.normalize()
    return pmf
```

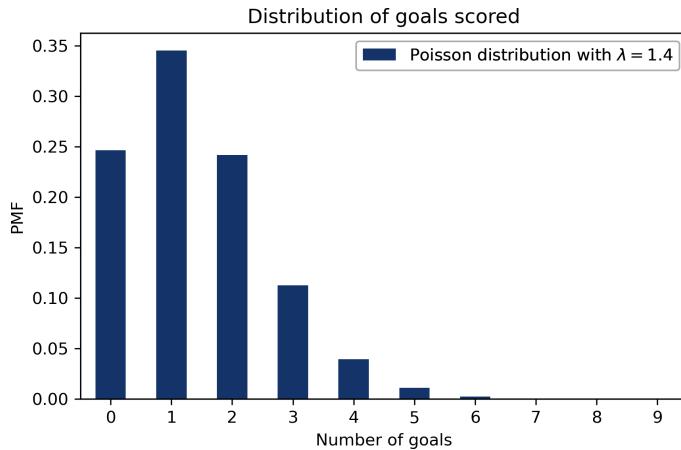
`make_poisson_pmf` takes as parameters the goal-scoring rate, `lam`, and an array of quantities, `qs`, where it should evaluate the Poisson PMF. It returns a `PMF` object.

For example, here's the distribution of goals scored for `lam=1.4`, computed for values of `k` from 0 to 9:

```
import numpy as np

lam = 1.4
goals = np.arange(10)
pmf_goals = make_poisson_pmf(lam, goals)
```

And here's what it looks like:



The most likely outcomes are 0, 1, and 2; higher values are possible but increasingly unlikely. Values above 7 are negligible. This distribution shows that if we know the goal-scoring rate, we can predict the number of goals.

Now let's turn it around: given a number of goals, what can we say about the goal-scoring rate?

To answer that, we need to think about the prior distribution of `lam`, which represents the range of possible values and their probabilities before we see the score.

The Gamma Distribution

If you have ever seen a soccer game, you have some information about `lam`. In most games, teams score a few goals each. In rare cases, a team might score more than 5 goals, but they almost never score more than 10.

Using [data from previous World Cups](#), I estimate that each team scores about 1.4 goals per game, on average. So I'll set the mean of `lam` to be 1.4.

For a good team against a bad one, we expect λ_m to be higher; for a bad team against a good one, we expect it to be lower.

To model the distribution of goal-scoring rates, I'll use a **gamma distribution**, which I chose because:

1. The goal scoring rate is continuous and non-negative, and the gamma distribution is appropriate for this kind of quantity.
2. The gamma distribution has only one parameter, `alpha`, which is the mean. So it's easy to construct a gamma distribution with the mean we want.
3. As we'll see, the shape of the gamma distribution is a reasonable choice, given what we know about soccer.

And there's one more reason, which I will reveal in [Chapter 18](#).

SciPy provides `gamma`, which creates an object that represents a gamma distribution. And the `gamma` object provides `pdf`, which evaluates the **probability density function** (PDF) of the gamma distribution.

Here's how we use it:

```
from scipy.stats import gamma

alpha = 1.4
qs = np.linspace(0, 10, 101)
ps = gamma(alpha).pdf(qs)
```

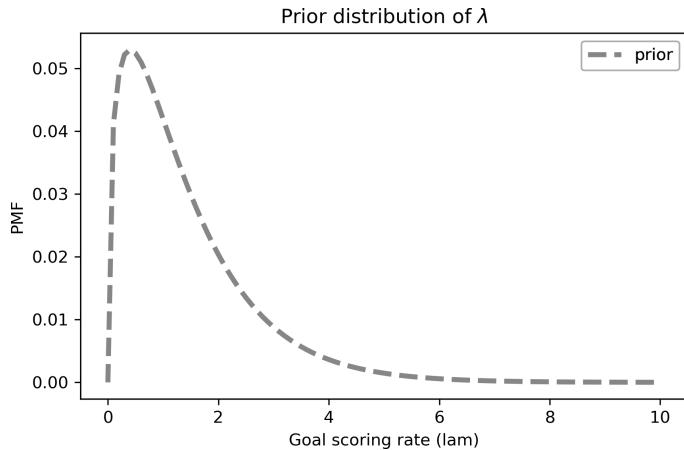
The parameter, `alpha`, is the mean of the distribution. The `qs` are possible values of λ_m between 0 and 10. The `ps` are **probability densities**, which we can think of as unnormalized probabilities.

To normalize them, we can put them in a `Pmf` and call `normalize`:

```
from empiricaldist import Pmf

prior = Pmf(ps, qs)
prior.normalize()
```

The result is a discrete approximation of a gamma distribution. Here's what it looks like:



This distribution represents our prior knowledge about goal scoring: λ is usually less than 2, occasionally as high as 6, and seldom higher than that.

As usual, reasonable people could disagree about the details of the prior, but this is good enough to get started. Let's do an update.

The Update

Suppose you are given the goal-scoring rate, λ , and asked to compute the probability of scoring a number of goals, k . That is precisely the question we answered by computing the Poisson PMF.

For example, if λ is 1.4, the probability of scoring 4 goals in a game is:

```
lam = 1.4
k = 4
poisson(lam).pmf(4)
0.039471954028253146
```

Now suppose we have an array of possible values for λ ; we can compute the likelihood of the data for each hypothetical value of λ , like this:

```
lams = prior.qs
k = 4
likelihood = poisson(lams).pmf(k)
```

And that's all we need to do the update. To get the posterior distribution, we multiply the prior by the likelihoods we just computed and normalize the result.

The following function encapsulates these steps:

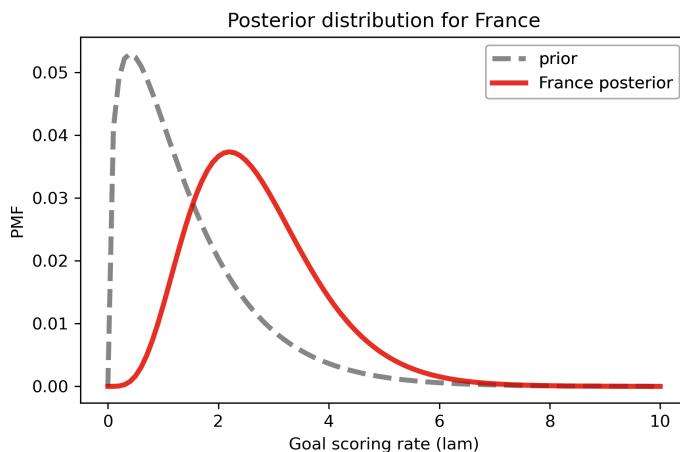
```
def update_poisson(pmf, data):
    """Update Pmf with a Poisson likelihood."""
    k = data
    lams = pmf.qs
    likelihood = poisson(lams).pmf(k)
    pmf *= likelihood
    pmf.normalize()
```

The first parameter is the prior; the second is the number of goals.

In the example, France scored 4 goals, so I'll make a copy of the prior and update it with the data:

```
france = prior.copy()
update_poisson(france, 4)
```

Here's what the posterior distribution looks like, along with the prior:

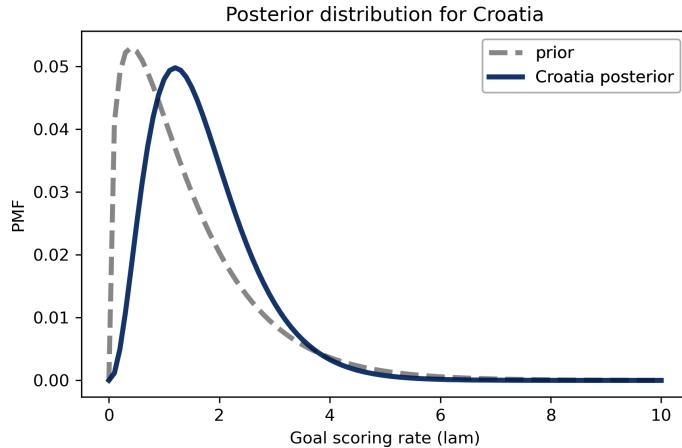


The data, $k=4$, makes us think higher values of λ are more likely and lower values are less likely. So the posterior distribution is shifted to the right.

Let's do the same for Croatia:

```
croatia = prior.copy()
update_poisson(croatia, 2)
```

And here are the results:



Here are the posterior means for these distributions:

```
print(croatia.mean(), france.mean())
1.6999765866755225 2.699772393342308
```

The mean of the prior distribution is about 1.4. After Croatia scores 2 goals, their posterior mean is 1.7, which is near the midpoint of the prior and the data. Likewise after France scores 4 goals, their posterior mean is 2.7.

These results are typical of a Bayesian update: the location of the posterior distribution is a compromise between the prior and the data.

Probability of Superiority

Now that we have a posterior distribution for each team, we can answer the first question: How confident should we be that France is the better team?

In the model, “better” means having a higher goal-scoring rate against the opponent. We can use the posterior distributions to compute the probability that a random value drawn from France’s distribution exceeds a value drawn from Croatia’s.

One way to do that is to enumerate all pairs of values from the two distributions, adding up the total probability that one value exceeds the other:

```
def prob_gt(pmf1, pmf2):
    """Compute the probability of superiority."""
    total = 0
    for q1, p1 in pmf1.items():
        for q2, p2 in pmf2.items():
            if q1 > q2:
                total += p1 * p2
    return total
```

This is similar to the method we use in “[Addends](#)” on page 73 to compute the distribution of a sum. Here’s how we use it:

```
prob_gt(france, croatia)
```

```
0.7499366290930155
```

Pmf provides a function that does the same thing:

```
Pmf.prob_gt(france, croatia)
```

```
0.7499366290930174
```

The results are slightly different because Pmf.prob_gt uses array operators rather than for loops.

Either way, the result is close to 75%. So, on the basis of one game, we have moderate confidence that France is actually the better team.

Of course, we should remember that this result is based on the assumption that the goal-scoring rate is constant. In reality, if a team is down by one goal, they might play more aggressively toward the end of the game, making them more likely to score, but also more likely to give up an additional goal.

As always, the results are only as good as the model.

Predicting the Rematch

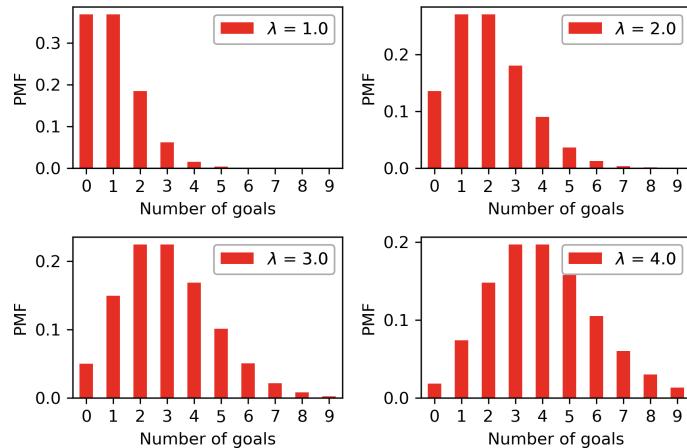
Now we can take on the second question: If the same teams played again, what is the chance Croatia would win? To answer this question, we’ll generate the “posterior predictive distribution”, which is the number of goals we expect a team to score.

If we knew the goal-scoring rate, `lam`, the distribution of goals would be a Poisson distribution with parameter `lam`. Since we don’t know `lam`, the distribution of goals is a mixture of a Poisson distributions with different values of `lam`.

First I’ll generate a sequence of Pmf objects, one for each value of `lam`:

```
pmf_seq = [make_poisson_pmf(lam, goals)
           for lam in prior.qs]
```

The following figure shows what these distributions look like for a few values of `lam`.

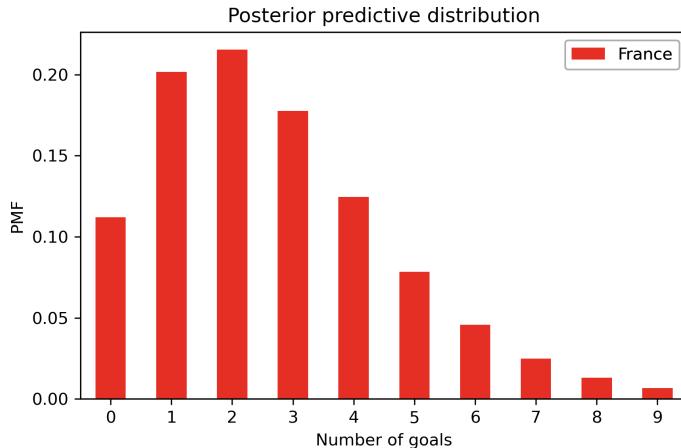


The predictive distribution is a mixture of these `pmf` objects, weighted with the posterior probabilities. We can use `make_mixture` from “General Mixtures” on page 93 to compute this mixture:

```
from utils import make_mixture

pred_france = make_mixture(france, pmf_seq)
```

Here’s the predictive distribution for the number of goals France would score in a rematch:



This distribution represents two sources of uncertainty: we don’t know the actual value of `lam`, and even if we did, we would not know the number of goals in the next game.

Here's the predictive distribution for Croatia:

```
pred_croatia = make_mixture(croatia, pmf_seq)
```

We can use these distributions to compute the probability that France wins, loses, or ties the rematch:

```
win = Pmf.prob_gt(pred_france, pred_croatia)
win
0.5703522415934519

lose = Pmf.prob_lt(pred_france, pred_croatia)
lose
0.26443376257235873

tie = Pmf.prob_eq(pred_france, pred_croatia)
tie
0.16521399583418947
```

Assuming that France wins half of the ties, their chance of winning the rematch is about 65%:

```
win + tie/2
0.6529592395105466
```

This is a bit lower than their probability of superiority, which is 75%. And that makes sense, because we are less certain about the outcome of a single game than we are about the goal-scoring rates. Even if France is the better team, they might lose the game.

The Exponential Distribution

As an exercise at the end of this notebook, you'll have a chance to work on the following variation on the World Cup Problem:

In the 2014 FIFA World Cup, Germany played Brazil in a semifinal match. Germany scored after 11 minutes and again at the 23 minute mark. At that point in the match, how many goals would you expect Germany to score after 90 minutes? What was the probability that they would score 5 more goals (as, in fact, they did)?

In this version, notice that the data is not the number of goals in a fixed period of time, but the time between goals.

To compute the likelihood of data like this, we can take advantage of the theory of Poisson processes again. If each team has a constant goal-scoring rate, we expect the time between goals to follow an [exponential distribution](#).

If the goal-scoring rate is λ , the probability of seeing an interval between goals of t is proportional to the PDF of the exponential distribution:

$$\lambda \exp(-\lambda t)$$

Because t is a continuous quantity, the value of this expression is not a probability; it is a probability density. However, it is proportional to the probability of the data, so we can use it as a likelihood in a Bayesian update.

SciPy provides `expon`, which creates an object that represents an exponential distribution. However, it does not take `lam` as a parameter in the way you might expect, which makes it awkward to work with. Since the PDF of the exponential distribution is so easy to evaluate, I'll use my own function:

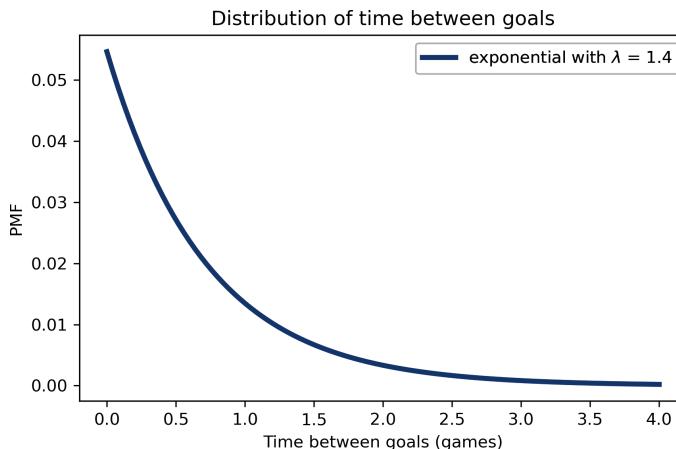
```
def expo_pdf(t, lam):
    """Compute the PDF of the exponential distribution."""
    return lam * np.exp(-lam * t)
```

To see what the exponential distribution looks like, let's assume again that `lam` is 1.4; we can compute the distribution of t like this:

```
lam = 1.4
qs = np.linspace(0, 4, 101)
ps = expo_pdf(qs, lam)
pmf_time = Pmf(ps, qs)
pmf_time.normalize()

25.616650745459093
```

And here's what it looks like:



It is counterintuitive, but true, that the most likely time to score a goal is immediately. After that, the probability of each successive interval is a little lower.

With a goal-scoring rate of 1.4, it is possible that a team will take more than one game to score a goal, but it is unlikely that they will take more than two games.

Summary

This chapter introduces three new distributions, so it can be hard to keep them straight. Let's review:

- If a system satisfies the assumptions of a Poisson model, the number of events in a period of time follows a Poisson distribution, which is a discrete distribution with integer quantities from 0 to infinity. In practice, we can usually ignore low-probability quantities above a finite limit.
- Also under the Poisson model, the interval between events follows an exponential distribution, which is a continuous distribution with quantities from 0 to infinity. Because it is continuous, it is described by a probability density function (PDF) rather than a probability mass function (PMF). But when we use an exponential distribution to compute the likelihood of the data, we can treat densities as unnormalized probabilities.
- The Poisson and exponential distributions are parameterized by an event rate, denoted λ or `lam`.
- For the prior distribution of λ , I used a gamma distribution, which is a continuous distribution with quantities from 0 to infinity, but I approximated it with a discrete, bounded PMF. The gamma distribution has one parameter, denoted α or alpha, which is also its mean.

I chose the gamma distribution because the shape is consistent with our background knowledge about goal-scoring rates. There are other distributions we could have used; however, we will see in [Chapter 18](#) that the gamma distribution can be a particularly good choice.

But we have a few things to do before we get there, starting with these exercises.

Exercises

Exercise 8-1.

Let's finish the exercise we started:

In the 2014 FIFA World Cup, Germany played Brazil in a semifinal match. Germany scored after 11 minutes and again at the 23 minute mark. At that point in the match, how many goals would you expect Germany to score after 90 minutes? What was the probability that they would score 5 more goals (as, in fact, they did)?

Here are the steps I recommend:

1. Starting with the same gamma prior we used in the previous problem, compute the likelihood of scoring a goal after 11 minutes for each possible value of λ . Don't forget to convert all times into games rather than minutes.
2. Compute the posterior distribution of λ for Germany after the first goal.
3. Compute the likelihood of scoring another goal after 12 more minutes and do another update. Plot the prior, posterior after one goal, and posterior after two goals.
4. Compute the posterior predictive distribution of goals Germany might score during the remaining time in the game, 90-23 minutes. Note: You will have to think about how to generate predicted goals for a fraction of a game.
5. Compute the probability of scoring 5 or more goals during the remaining time.

Exercise 8-2.

Returning to the first version of the World Cup Problem, suppose France and Croatia play a rematch. What is the probability that France scores first?

Exercise 8-3.

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?

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 well modeled by a gamma distribution with mean 2.8.

In what ways do you think the outcome of these games might violate the assumptions of the Poisson model? How would these violations affect your predictions?

CHAPTER 9

Decision Analysis

This chapter presents a problem inspired by the game show *The Price is Right*. It is a silly example, but it demonstrates a useful process called [Bayesian decision analysis](#).

As in previous examples, we'll use data and prior distribution to compute a posterior distribution; then we'll use the posterior distribution to choose an optimal strategy in a game that involves bidding.

As part of the solution, we will use kernel density estimation (KDE) to estimate the prior distribution, and a normal distribution to compute the likelihood of the data.

And at the end of the chapter, I pose a related problem you can solve as an exercise.

The Price Is Right Problem

On November 1, 2007, contestants named Letia and Nathaniel appeared on *The Price is Right*, an American television game show. They competed in a game called “The Showcase”, where the objective is to guess the price of a collection of prizes. The contestant who comes closest to the actual price, without going over, wins the prizes.

Nathaniel went first. His showcase included a dishwasher, a wine cabinet, a laptop computer, and a car. He bid \$26,000.

Letia's showcase included a pinball machine, a video arcade game, a pool table, and a cruise of the Bahamas. She bid \$21,500. The actual price of Nathaniel's showcase was \$25,347. His bid was too high, so he lost. The actual price of Letia's showcase was \$21,578.

She was only off by \$78, so she won her showcase and, because her bid was off by less than 250, she also won Nathaniel's showcase.

For a Bayesian thinker, this scenario suggests several questions:

1. Before seeing the prizes, what prior beliefs should the contestants have about the price of the showcase?
2. After seeing the prizes, how should the contestants update those beliefs?
3. Based on the posterior distribution, what should the contestants bid?

The third question demonstrates a common use of Bayesian methods: decision analysis.

This problem is inspired by [an example](#) in Cameron Davidson-Pilon's book, *Probabilistic Programming and Bayesian Methods for Hackers*.

The Prior

To choose a prior distribution of prices, we can take advantage of data from previous episodes. Fortunately, [fans of the show keep detailed records](#).

For this example, I downloaded files containing the price of each showcase from the 2011 and 2012 seasons and the bids offered by the contestants.

The following function reads the data and cleans it up a little:

```
import pandas as pd

def read_data(filename):
    """Read the showcase price data."""
    df = pd.read_csv(filename, index_col=0, skiprows=[1])
    return df.dropna().transpose()
```

I'll read both files and concatenate them:

```
df2011 = read_data('showcases.2011.csv')
df2012 = read_data('showcases.2012.csv')

df = pd.concat([df2011, df2012], ignore_index=True)
```

Here's what the dataset looks like:

```
df.head(3)
```

	Showcase 1	Showcase 2	Bid 1	Bid 2	Difference 1	Difference 2
0	50969.0	45429.0	42000.0	34000.0	8969.0	11429.0
1	21901.0	34061.0	14000.0	59900.0	7901.0	-25839.0
2	32815.0	53186.0	32000.0	45000.0	815.0	8186.0

The first two columns, Showcase 1 and Showcase 2, are the values of the showcases in dollars. The next two columns are the bids the contestants made. The last two columns are the differences between the actual values and the bids.

Kernel Density Estimation

This dataset contains the prices for 313 previous showcases, which we can think of as a sample from the population of possible prices.

We can use this sample to estimate the prior distribution of showcase prices. One way to do that is kernel density estimation (KDE), which uses the sample to estimate a smooth distribution. If you are not familiar with KDE, you can [read about it online](#).

SciPy provides `gaussian_kde`, which takes a sample and returns an object that represents the estimated distribution.

The following function takes `sample`, makes a KDE, evaluates it at a given sequence of quantities, `qs`, and returns the result as a normalized PMF.

```
from scipy.stats import gaussian_kde
from empiricaldist import Pmf

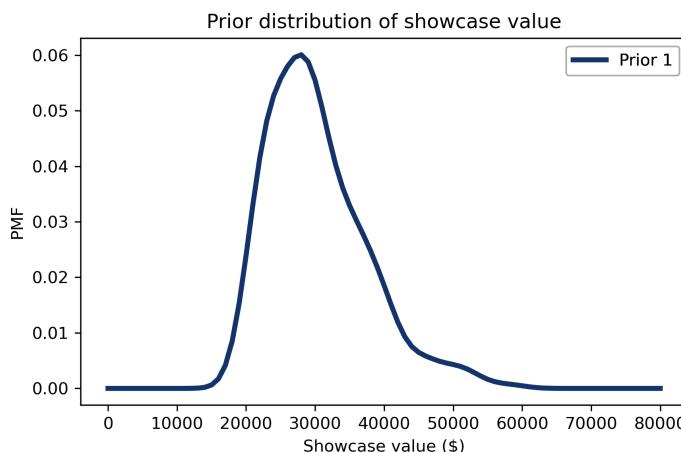
def kde_from_sample(sample, qs):
    """Make a kernel density estimate from a sample."""
    kde = gaussian_kde(sample)
    ps = kde(qs)
    pmf = Pmf(ps, qs)
    pmf.normalize()
    return pmf
```

We can use it to estimate the distribution of values for Showcase 1:

```
import numpy as np

qs = np.linspace(0, 80000, 81)
prior1 = kde_from_sample(df['Showcase 1'], qs)
```

Here's what it looks like:



Exercise 9-1.

Use this function to make a Pmf that represents the prior distribution for Showcase 2, and plot it.

Distribution of Error

To update these priors, we have to answer these questions:

- What data should we consider and how should we quantify it?
- Can we compute a likelihood function? That is, for each hypothetical price, can we compute the conditional likelihood of the data?

To answer these questions, I will model each contestant as a price-guessing instrument with known error characteristics. In this model, when the contestant sees the prizes, they guess the price of each prize and add up the prices. Let's call this total `guess`.

Now the question we have to answer is, “If the actual price is `price`, what is the likelihood that the contestant’s guess would be `guess`?”

Equivalently, if we define `error = guess - price`, we can ask, “What is the likelihood that the contestant’s guess is off by `error`?”

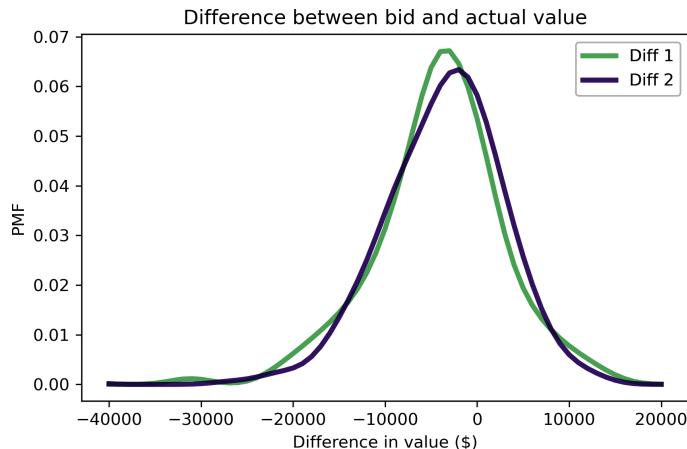
To answer this question, I’ll use the historical data again. For each showcase in the dataset, let’s look at the difference between the contestant’s bid and the actual price:

```
sample_diff1 = df['Bid 1'] - df['Showcase 1']
sample_diff2 = df['Bid 2'] - df['Showcase 2']
```

To visualize the distribution of these differences, we can use KDE again:

```
qs = np.linspace(-40000, 20000, 61)
kde_diff1 = kde_from_sample(sample_diff1, qs)
kde_diff2 = kde_from_sample(sample_diff2, qs)
```

Here’s what these distributions look like:



It looks like the bids are too low more often than too high, which makes sense. Remember that under the rules of the game, you lose if you overbid, so contestants probably underbid to some degree deliberately.

For example, if they guess that the value of the showcase is \$40,000, they might bid \$36,000 to avoid going over.

It looks like these distributions are well modeled by a normal distribution, so we can summarize them with their mean and standard deviation.

For example, here is the mean and standard deviation of `Diff` for Player 1:

```
mean_diff1 = sample_diff1.mean()
std_diff1 = sample_diff1.std()

print(mean_diff1, std_diff1)
-4116.3961661341855 6899.909806377117
```

Now we can use these differences to model the contestant's distribution of errors. This step is a little tricky because we don't actually know the contestant's guesses; we only know what they bid.

So we have to make some assumptions:

- I'll assume that contestants underbid because they are being strategic, and that on average their guesses are accurate. In other words, the mean of their errors is 0.
- But I'll assume that the spread of the differences reflects the actual spread of their errors. So, I'll use the standard deviation of the differences as the standard deviation of their errors.

Based on these assumptions, I'll make a normal distribution with parameters 0 and `std_diff1`. SciPy provides an object called `norm` that represents a normal distribution with the given mean and standard deviation:

```
from scipy.stats import norm

error_dist1 = norm(0, std_diff1)
```

The result is an object that provides `pdf`, which evaluates the probability density function of the normal distribution.

For example, here is the probability density of `error=-100`, based on the distribution of errors for Player 1:

```
error = -100
error_dist1.pdf(error)

5.781240564008691e-05
```

By itself, this number doesn't mean very much, because probability densities are not probabilities. But they are proportional to probabilities, so we can use them as likelihoods in a Bayesian update, as we'll see in the next section.

Update

Suppose you are Player 1. You see the prizes in your showcase and your guess for the total price is \$23,000.

From your guess I will subtract away each hypothetical price in the prior distribution; the result is your error under each hypothesis.

```
guess1 = 23000
error1 = guess1 - prior1.qs
```

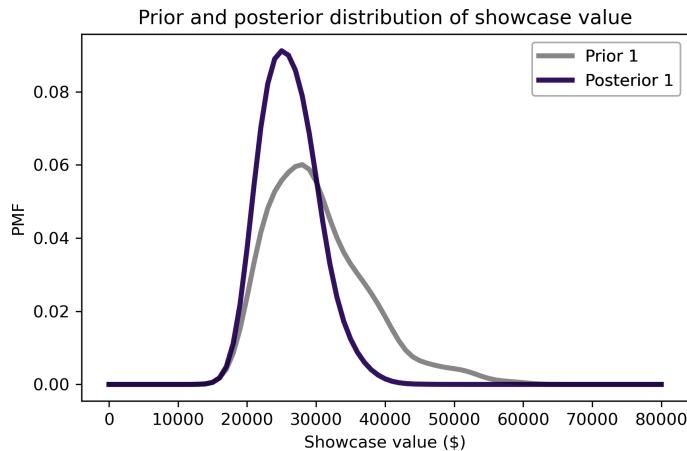
Now suppose we know, based on past performance, that your estimation error is well modeled by `error_dist1`. Under that assumption we can compute the likelihood of your error under each hypothesis:

```
likelihood1 = error_dist1.pdf(error1)
```

The result is an array of likelihoods, which we can use to update the prior:

```
posterior1 = prior1 * likelihood1
posterior1.normalize()
```

Here's what the posterior distribution looks like:



Because your initial guess is in the lower end of the range, the posterior distribution has shifted to the left. We can compute the posterior mean to see by how much:

```
prior1.mean(), posterior1.mean()
(30299.488817891375, 26192.024002392536)
```

Before you saw the prizes, you expected to see a showcase with a value close to \$30,000. After making a guess of \$23,000, you updated the prior distribution. Based on the combination of the prior and your guess, you now expect the actual price to be about \$26,000.

Exercise 9-2.

Now suppose you are Player 2. When you see your showcase, you guess that the total price is \$38,000.

Use `diff2` to construct a normal distribution that represents the distribution of your estimation errors.

Compute the likelihood of your guess for each actual price and use it to update `prior2`.

Plot the posterior distribution and compute the posterior mean. Based on the prior and your guess, what do you expect the actual price of the showcase to be?

Probability of Winning

Now that we have a posterior distribution for each player, let's think about strategy.

First, from the point of view of Player 1, let's compute the probability that Player 2 overbids. To keep it simple, I'll use only the performance of past players, ignoring the value of the showcase.

The following function takes a sequence of past bids and returns the fraction that overbid.

```
def prob_overbid(sample_diff):
    """Compute the probability of an overbid."""
    return np.mean(sample_diff > 0)
```

Here's an estimate for the probability that Player 2 overbids:

```
prob_overbid(sample_diff2)
0.29073482428115016
```

Now suppose Player 1 underbids by \$5,000. What is the probability that Player 2 underbids by more?

The following function uses past performance to estimate the probability that a player underbids by more than a given amount, `diff`:

```
def prob_worse_than(diff, sample_diff):
    """Probability opponent diff is worse than given diff."""
    return np.mean(sample_diff < diff)
```

Here's the probability that Player 2 underbids by more than \$5,000:

```
prob_worse_than(-5000, sample_diff2)
0.38338658146964855
```

And here's the probability they underbid by more than \$10,000:

```
prob_worse_than(-10000, sample_diff2)
0.14376996805111822
```

We can combine these functions to compute the probability that Player 1 wins, given the difference between their bid and the actual price:

```
def compute_prob_win(diff, sample_diff):
    """Probability of winning for a given diff."""
    # if you overbid you lose
    if diff > 0:
        return 0

    # if the opponent overbids, you win
    p1 = prob_overbid(sample_diff)
```

```

# or if their bid is worse than yours, you win
p2 = prob_worse_than(diff, sample_diff)

# p1 and p2 are mutually exclusive, so we can add them
return p1 + p2

```

Here's the probability that you win, given that you underbid by \$5,000:

```

compute_prob_win(-5000, sample_diff2)
0.6741214057507987

```

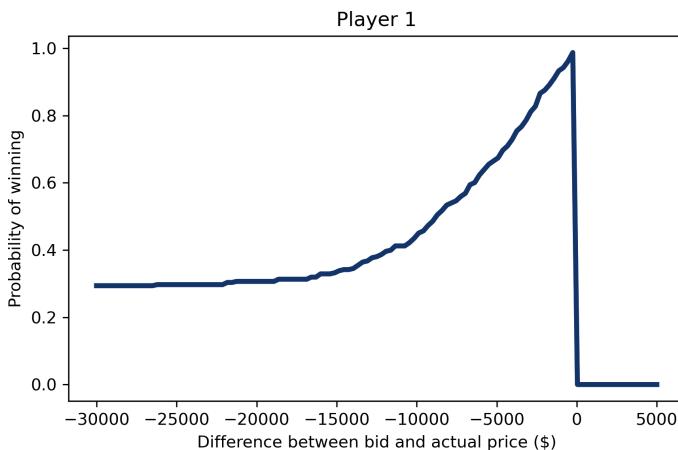
Now let's look at the probability of winning for a range of possible differences:

```

xs = np.linspace(-30000, 5000, 121)
ys = [compute_prob_win(x, sample_diff2)
      for x in xs]

```

Here's what it looks like:



If you underbid by \$30,000, the chance of winning is about 30%, which is mostly the chance your opponent overbids.

As your bids gets closer to the actual price, your chance of winning approaches 1.

And, of course, if you overbid, you lose (even if your opponent also overbids).

Exercise 9-3.

Run the same analysis from the point of view of Player 2. Using the sample of differences from Player 1, compute:

1. The probability that Player 1 overbids.
2. The probability that Player 1 underbids by more than \$5,000.

3. The probability that Player 2 wins, given that they underbid by \$5,000.

Then plot the probability that Player 2 wins for a range of possible differences between their bid and the actual price.

Decision Analysis

In the previous section we computed the probability of winning, given that we have underbid by a particular amount.

In reality the contestants don't know how much they have underbid by, because they don't know the actual price.

But they do have a posterior distribution that represents their beliefs about the actual price, and they can use that to estimate their probability of winning with a given bid.

The following function takes a possible bid, a posterior distribution of actual prices, and a sample of differences for the opponent.

It loops through the hypothetical prices in the posterior distribution and, for each price:

1. Computes the difference between the bid and the hypothetical price,
2. Computes the probability that the player wins, given that difference, and
3. Adds up the weighted sum of the probabilities, where the weights are the probabilities in the posterior distribution.

```
def total_prob_win(bid, posterior, sample_diff):
    """Computes the total probability of winning with a given bid.

    bid: your bid
    posterior: Pmf of showcase value
    sample_diff: sequence of differences for the opponent

    returns: probability of winning
    """
    total = 0
    for price, prob in posterior.items():
        diff = bid - price
        total += prob * compute_prob_win(diff, sample_diff)
    return total
```

This loop implements the law of total probability:

$$P(\text{win}) = \sum_{\text{price}} P(\text{price}) P(\text{win} \mid \text{price})$$

Here's the probability that Player 1 wins, based on a bid of \$25,000 and the posterior distribution `posterior1`:

```
total_prob_win(25000, posterior1, sample_diff2)
```

```
0.4842210945439812
```

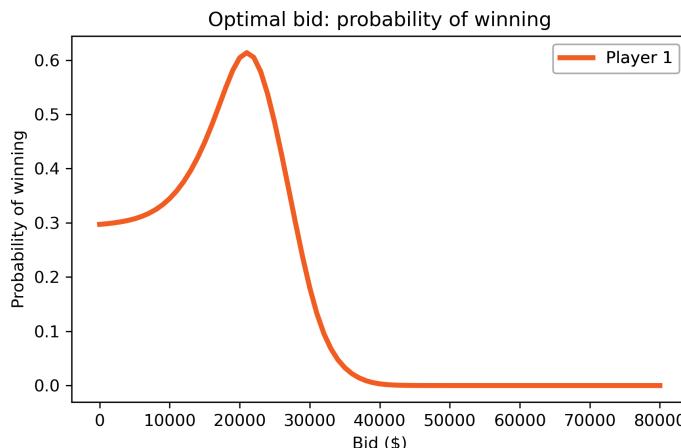
Now we can loop through a series of possible bids and compute the probability of winning for each one:

```
bids = posterior1.qs

probs = [total_prob_win(bid, posterior1, sample_diff2)
         for bid in bids]

prob_win_series = pd.Series(probs, index=bids)
```

Here are the results:



And here's the bid that maximizes Player 1's chance of winning:

```
prob_win_series.idxmax()
```

```
21000.0
```

```
prob_win_series.max()
```

```
0.6136807192359474
```

Recall that your guess was \$23,000. Using your guess to compute the posterior distribution, the posterior mean is about \$26,000. But the bid that maximizes your chance of winning is \$21,000.

Exercise 9-4.

Do the same analysis for Player 2.

Maximizing Expected Gain

In the previous section we computed the bid that maximizes your chance of winning. And if that's your goal, the bid we computed is optimal.

But winning isn't everything. Remember that if your bid is off by \$250 or less, you win both showcases. So it might be a good idea to increase your bid a little: it increases the chance you overbid and lose, but it also increases the chance of winning both showcases.

Let's see how that works out. The following function computes how much you will win, on average, given your bid, the actual price, and a sample of errors for your opponent.

```
def compute_gain(bid, price, sample_diff):
    """Compute expected gain given a bid and actual price."""
    diff = bid - price
    prob = compute_prob_win(diff, sample_diff)

    # if you are within 250 dollars, you win both showcases
    if -250 <= diff <= 0:
        return 2 * price * prob
    else:
        return price * prob
```

For example, if the actual price is \$35,000 and you bid \$30,000, you will win about \$23,600 worth of prizes on average, taking into account your probability of losing, winning one showcase, or winning both.

```
compute_gain(30000, 35000, sample_diff2)
23594.249201277955
```

In reality we don't know the actual price, but we have a posterior distribution that represents what we know about it. By averaging over the prices and probabilities in the posterior distribution, we can compute the expected gain for a particular bid.

In this context, "expected" means the average over the possible showcase values, weighted by their probabilities.

```
def expected_gain(bid, posterior, sample_diff):
    """Compute the expected gain of a given bid."""
    total = 0
    for price, prob in posterior.items():
        total += prob * compute_gain(bid, price, sample_diff)
    return total
```

For the posterior we computed earlier, based on a guess of \$23,000, the expected gain for a bid of \$21,000 is about \$16,900:

```
expected_gain(21000, posterior1, sample_diff2)
```

```
16923.59933856512
```

But can we do any better?

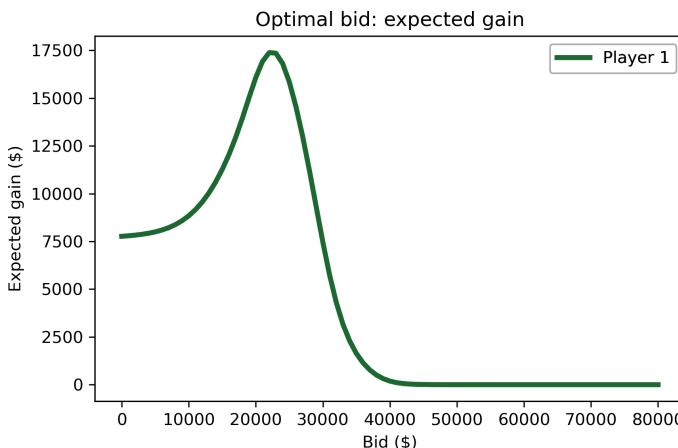
To find out, we can loop through a range of bids and find the one that maximizes expected gain:

```
bids = posterior1.qs
```

```
gains = [expected_gain(bid, posterior1, sample_diff2) for bid in bids]
```

```
expected_gain_series = pd.Series(gains, index=bids)
```

Here are the results:



Here is the optimal bid:

```
expected_gain_series.idxmax()
```

```
22000.0
```

With that bid, the expected gain is about \$17,400:

```
expected_gain_series.max()
```

```
17384.899584430797
```

Recall that your initial guess was \$23,000. The bid that maximizes the chance of winning is \$21,000. And the bid that maximizes your expected gain is \$22,000.

Exercise 9-5.

Do the same analysis for Player 2.

Summary

There's a lot going on this this chapter, so let's review the steps:

1. First we used KDE and data from past shows to estimate prior distributions for the values of the showcases.
2. Then we used bids from past shows to model the distribution of errors as a normal distribution.
3. We did a Bayesian update using the distribution of errors to compute the likelihood of the data.
4. We used the posterior distribution for the value of the showcase to compute the probability of winning for each possible bid, and identified the bid that maximizes the chance of winning.
5. Finally, we used probability of winning to compute the expected gain for each possible bid, and identified the bid that maximizes expected gain.

Incidentally, this example demonstrates the hazard of using the word “optimal” without specifying what you are optimizing. The bid that maximizes the chance of winning is not generally the same as the bid that maximizes expected gain.

Discussion

When people discuss the pros and cons of Bayesian estimation, as contrasted with classical methods sometimes called “frequentist”, they often claim that in many cases Bayesian methods and frequentist methods produce the same results.

In my opinion, this claim is mistaken because Bayesian and frequentist method produce different *kinds* of results:

- The result of frequentist methods is usually a single value that is considered to be the best estimate (by one of several criteria) or an interval that quantifies the precision of the estimate.
- The result of Bayesian methods is a posterior distribution that represents all possible outcomes and their probabilities.

Granted, you can use the posterior distribution to choose a “best” estimate or compute an interval. And in that case the result might be the same as the frequentist estimate.

But doing so discards useful information and, in my opinion, eliminates the primary benefit of Bayesian methods: the posterior distribution is more useful than a single estimate, or even an interval.

The example in this chapter demonstrates the point. Using the entire posterior distribution, we can compute the bid that maximizes the probability of winning, or the bid that maximizes expected gain, even if the rules for computing the gain are complicated (and nonlinear).

With a single estimate or an interval, we can't do that, even if they are "optimal" in some sense. In general, frequentist estimation provides little guidance for decision-making.

If you hear someone say that Bayesian and frequentist methods produce the same results, you can be confident that they don't understand Bayesian methods.

More Exercises

Exercise 9-6.

When I worked in Cambridge, Massachusetts, I usually took the subway to South Station and then a commuter train home to Needham. Because the subway was unpredictable, I left the office early enough that I could wait up to 15 minutes and still catch the commuter train.

When I got to the subway stop, there were usually about 10 people waiting on the platform. If there were fewer than that, I figured I just missed a train, so I expected to wait a little longer than usual. And if there were more than that, I expected another train soon.

But if there were *a lot* more than 10 passengers waiting, I inferred that something was wrong, and I expected a long wait. In that case, I might leave and take a taxi.

We can use Bayesian decision analysis to quantify the analysis I did intuitively. Given the number of passengers on the platform, how long should we expect to wait? And when should we give up and take a taxi?

My analysis of this problem is in `redline.ipynb`, which is in the repository for this book. [Click here to run this notebook on Colab.](#)

Exercise 9-7.

This exercise is inspired by a true story. In 2001, I created [Green Tea Press](#) to publish my books, starting with *Think Python*. I ordered 100 copies from a short-run printer and made the book available for sale through a distributor.

After the first week, the distributor reported that 12 copies were sold. Based on that report, I thought I would run out of copies in about 8 weeks, so I got ready to order more. My printer offered me a discount if I ordered more than 1,000 copies, so I went a little crazy and ordered 2,000.

A few days later, my mother called to tell me that her *copies* of the book had arrived. Surprised, I asked how many. She said 10.

It turned out I had sold only two books to non-relatives. And it took a lot longer than I expected to sell 2,000 copies.

The details of this story are unique, but the general problem is something almost every retailer has to figure out. Based on past sales, how do you predict future sales? And based on those predictions, how do you decide how much to order and when?

Often the cost of a bad decision is complicated. If you place a lot of small orders rather than one big one, your costs are likely to be higher. If you run out of inventory, you might lose customers. And if you order too much, you have to pay the various costs of holding inventory.

So, let's solve a version of the problem I faced. It will take some work to set up the problem; the details are in the notebook for this chapter.

CHAPTER 10

Testing

In “The Euro Problem” on page 43 I presented a problem from David MacKay’s book, *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 started to answer this question in Chapter 4; to review, our answer was based on these modeling decisions:

- If you spin a coin on edge, there is some probability, x , that it will land heads up.
- The value of x varies from one coin to the next, depending on how the coin is balanced and possibly other factors.

Starting with a uniform prior distribution for x , we updated it with the given data, 140 heads and 110 tails. Then we used the posterior distribution to compute the most likely value of x , the posterior mean, and a credible interval.

But we never really answered MacKay’s question: “Do these data give evidence that the coin is biased rather than fair?”

In this chapter, finally, we will.

Estimation

Let’s review the solution to the Euro Problem from “The Binomial Likelihood Function” on page 51. We started with a uniform prior:

```
import numpy as np
from empiricaldist import Pmf

xs = np.linspace(0, 1, 101)
uniform = Pmf(1, xs)
```

And we used the binomial distribution to compute the probability of the data for each possible value of x :

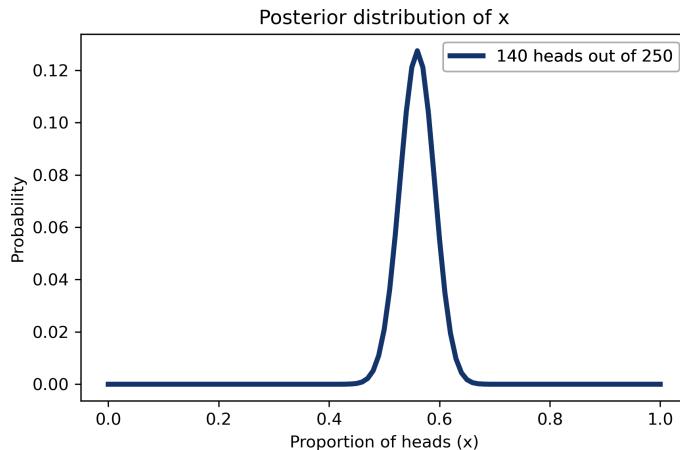
```
from scipy.stats import binom

k, n = 140, 250
likelihood = binom.pmf(k, n, xs)
```

We computed the posterior distribution in the usual way:

```
posterior = uniform * likelihood
posterior.normalize()
```

And here's what it looks like:



Again, the posterior mean is about 0.56, with a 90% credible interval from 0.51 to 0.61:

```
print(posterior.mean(),
      posterior.credible_interval(0.9))

0.5595238095238095 [0.51 0.61]
```

The prior mean was 0.5, and the posterior mean is 0.56, so it seems like the data is evidence that the coin is biased.

But, it turns out not to be that simple.

Evidence

In “Oliver’s Blood” on page 71, I said that data are considered evidence in favor of a hypothesis, A , if the data are more likely under A than under the alternative, B ; that is if

$$P(D|A) > P(D|B)$$

Furthermore, we can quantify the strength of the evidence by computing the ratio of these likelihoods, which is known as the **Bayes factor** and often denoted K :

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

So, for the Euro Problem, let’s consider two hypotheses, `fair` and `biased`, and compute the likelihood of the data under each hypothesis.

If the coin is fair, the probability of heads is 50%, and we can compute the probability of the data (140 heads out of 250 spins) using the binomial distribution:

```
k = 140
n = 250

like_fair = binom.pmf(k, n, p=0.5)
like_fair

0.008357181724917673
```

That’s the probability of the data, given that the coin is fair.

But if the coin is biased, what’s the probability of the data? That depends on what “biased” means. If we know ahead of time that “biased” means the probability of heads is 56%, we can use the binomial distribution again:

```
like_biased = binom.pmf(k, n, p=0.56)
like_biased

0.05077815959517949
```

Now we can compute the likelihood ratio:

```
K = like_biased / like_fair
K

6.075990838368387
```

The data are about 6 times more likely if the coin is biased, by this definition, than if it is fair.

But we used the data to define the hypothesis, which seems like cheating. To be fair, we should define “biased” before we see the data.

Uniformly Distributed Bias

Suppose “biased” means that the probability of heads is anything except 50%, and all other values are equally likely.

We can represent that definition by making a uniform distribution and removing 50%:

```
biased_uniform = uniform.copy()
biased_uniform[0.5] = 0
biased_uniform.normalize()
```

To compute the total probability of the data under this hypothesis, we compute the conditional probability of the data for each value of x :

```
xs = biased_uniform.qs
likelihood = binom.pmf(k, n, xs)
```

Then multiply by the prior probabilities and add up the products:

```
like_uniform = np.sum(biased_uniform * likelihood)
like_uniform

0.0039004919277704267
```

So that’s the probability of the data under the “biased uniform” hypothesis.

Now we can compute the likelihood ratio of the data under the `fair` and `biased uniform` hypotheses:

```
K = like_fair / like_uniform
K

2.1425968518013954
```

The data are about two times more likely if the coin is fair than if it is biased, by this definition of “biased”.

To get a sense of how strong that evidence is, we can apply Bayes’s rule. For example, if the prior probability is 50% that the coin is biased, the prior odds are 1, so the posterior odds are about 2.1 to 1 and the posterior probability is about 68%.

```
prior_odds = 1
posterior_odds = prior_odds * K
posterior_odds

2.1425968518013954

def prob(o):
    return o / (o+1)

posterior_probability = prob(posterior_odds)
posterior_probability

0.6817918278551125
```

Evidence that “moves the needle” from 50% to 68% is not very strong.

Now suppose “biased” doesn’t mean every value of x is equally likely. Maybe values near 50% are more likely and values near the extremes are less likely. We could use a triangle-shaped distribution to represent this alternative definition of “biased”:

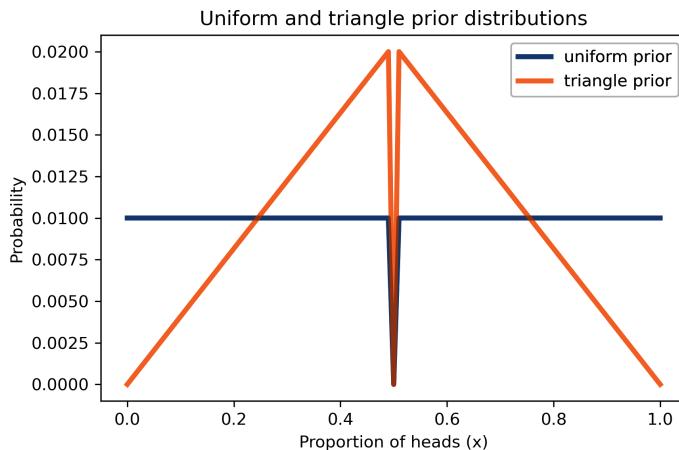
```
ramp_up = np.arange(50)
ramp_down = np.arange(50, -1, -1)
a = np.append(ramp_up, ramp_down)

triangle = Pmf(a, xs, name='triangle')
triangle.normalize()
```

As we did with the uniform distribution, we can remove 50% as a possible value of x (but it doesn’t make much difference if we skip this detail):

```
biased_triangle = triangle.copy()
biased_triangle[0.5] = 0
biased_triangle.normalize()
```

Here’s what the triangle prior looks like, compared to the uniform prior:



Exercise 10-1.

Now compute the total probability of the data under this definition of “biased” and compute the Bayes factor, compared with the fair hypothesis. Is the data evidence that the coin is biased?

Bayesian Hypothesis Testing

What we've done so far in this chapter is sometimes called "Bayesian hypothesis testing" in contrast with [statistical hypothesis testing](#).

In statistical hypothesis testing, we compute a p-value, which is hard to define concisely, and use it to determine whether the results are "statistically significant", which is also hard to define concisely.

The Bayesian alternative is to report the Bayes factor, K , which summarizes the strength of the evidence in favor of one hypothesis or the other.

Some people think it is better to report K than a posterior probability because K does not depend on a prior probability. But as we saw in this example, K often depends on a precise definition of the hypotheses, which can be just as controversial as a prior probability.

In my opinion, Bayesian hypothesis testing is better because it measures the strength of the evidence on a continuum, rather than trying to make a binary determination. But it doesn't solve what I think is the fundamental problem, which is that hypothesis testing is not asking the question we really care about.

To see why, suppose you test the coin and decide that it is biased after all. What can you do with this answer? In my opinion, not much. In contrast, there are two questions I think are more useful (and therefore more meaningful):

- Prediction: Based on what we know about the coin, what should we expect to happen in the future?
- Decision-making: Can we use those predictions to make better decisions?

At this point, we've seen a few examples of prediction. For example, in [Chapter 8](#) we used the posterior distribution of goal-scoring rates to predict the outcome of soccer games.

And we've seen one previous example of decision analysis: In [Chapter 9](#) we used the distribution of prices to choose an optimal bid on *The Price is Right*.

So let's finish this chapter with another example of Bayesian decision analysis, the Bayesian Bandit strategy.

Bayesian Bandits

If you have ever been to a casino, you have probably seen a slot machine, which is sometimes called a "one-armed bandit" because it has a handle like an arm and the ability to take money like a bandit.

The Bayesian Bandit strategy is named after one-armed bandits because it solves a problem based on a simplified version of a slot machine.

Suppose that each time you play a slot machine, there is a fixed probability that you win. And suppose that different machines give you different probabilities of winning, but you don't know what the probabilities are.

Initially, you have the same prior belief about each of the machines, so you have no reason to prefer one over the others. But if you play each machine a few times, you can use the results to estimate the probabilities. And you can use the estimated probabilities to decide which machine to play next.

At a high level, that's the Bayesian Bandit strategy. Now let's see the details.

Prior Beliefs

If we know nothing about the probability of winning, we can start with a uniform prior:

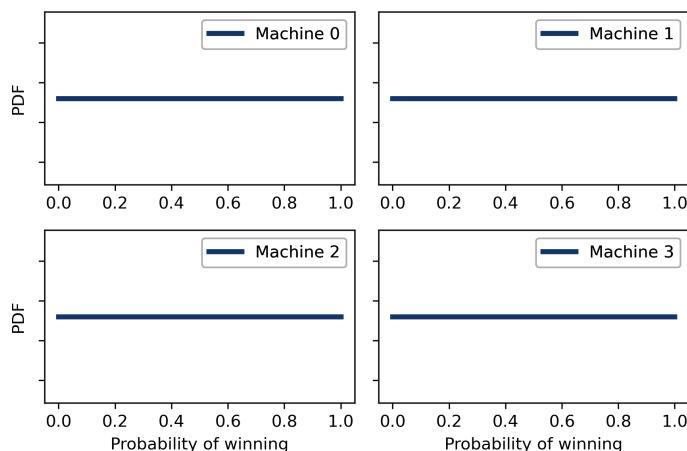
```
xs = np.linspace(0, 1, 101)
prior = Pmf(1, xs)
prior.normalize()
```

Supposing we are choosing from four slot machines, I'll make four copies of the prior, one for each machine:

```
beliefs = [prior.copy() for i in range(4)]
```

Here's what the prior distributions look like for the four machines:

```
plot(beliefs)
```



The Update

Each time we play a machine, we can use the outcome to update our beliefs. The following function does the update.

```
likelihood = {  
    'W': xs,  
    'L': 1 - xs  
}  
  
def update(pmf, data):  
    """Update the probability of winning.  
    pmf *= likelihood[data]  
    pmf.normalize()  
}
```

This function updates the prior distribution in place. `pmf` is a `Pmf` that represents the prior distribution of `x`, which is the probability of winning.

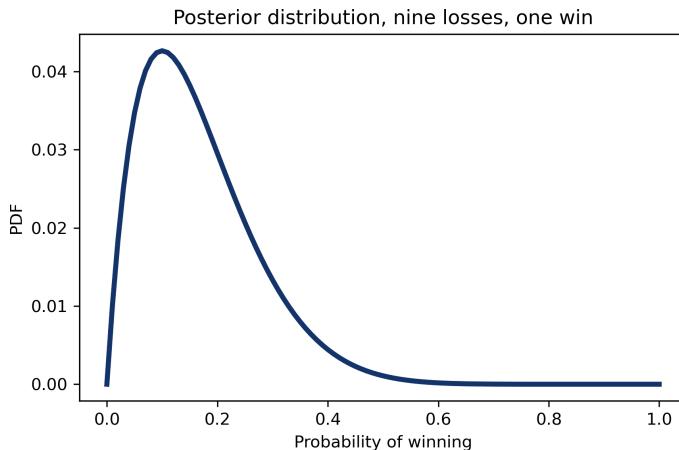
`data` is a string, either `W` if the outcome is a win or `L` if the outcome is a loss.

The likelihood of the data is either `xs` or `1-xs`, depending on the outcome.

Suppose we choose a machine, play 10 times, and win once. We can compute the posterior distribution of `x`, based on this outcome, like this:

```
bandit = prior.copy()  
  
for outcome in 'WLLLLLLLLL':  
    update(bandit, outcome)
```

Here's what the posterior looks like:



Multiple Bandits

Now suppose we have four machines with these probabilities:

```
actual_probs = [0.10, 0.20, 0.30, 0.40]
```

Remember that as a player, we don't know these probabilities.

The following function takes the index of a machine, simulates playing the machine once, and returns the outcome, W or L.

```
from collections import Counter

# count how many times we've played each machine
counter = Counter()

def play(i):
    """Play machine i.

    i: index of the machine to play

    returns: string 'W' or 'L'
    """
    counter[i] += 1
    p = actual_probs[i]
    if np.random.random() < p:
        return 'W'
    else:
        return 'L'
```

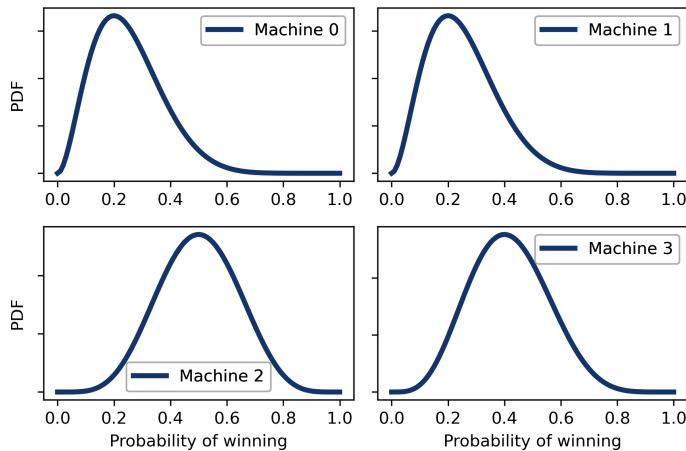
`counter` is a `Counter`, which is a kind of dictionary we'll use to keep track of how many times each machine is played.

Here's a test that plays each machine 10 times:

```
for i in range(4):
    for _ in range(10):
        outcome = play(i)
        update(beliefs[i], outcome)
```

Each time through the inner loop, we play one machine and update our beliefs.

Here's what our posterior beliefs look like:



Here are the actual probabilities, posterior means, and 90% credible intervals:

	Actual P(win)	Posterior mean	Credible interval
0	0.1	0.250	[0.08, 0.47]
1	0.2	0.250	[0.08, 0.47]
2	0.3	0.500	[0.27, 0.73]
3	0.4	0.417	[0.2, 0.65]

We expect the credible intervals to contain the actual probabilities most of the time.

Explore and Exploit

Based on these posterior distributions, which machine do you think we should play next? One option would be to choose the machine with the highest posterior mean.

That would not be a bad idea, but it has a drawback: since we have only played each machine a few times, the posterior distributions are wide and overlapping, which means we are not sure which machine is the best; if we focus on one machine too soon, we might choose the wrong machine and play it more than we should.

To avoid that problem, we could go to the other extreme and play all machines equally until we are confident we have identified the best machine, and then play it exclusively.

That's not a bad idea either, but it has a drawback: while we are gathering data, we are not making good use of it; until we're sure which machine is the best, we are playing the others more than we should.

The Bayesian Bandits strategy avoids both drawbacks by gathering and using data at the same time. In other words, it balances exploration and exploitation.

The kernel of the idea is called **Thompson sampling**: when we choose a machine, we choose at random so that the probability of choosing each machine is proportional to the probability that it is the best.

Given the posterior distributions, we can compute the “probability of superiority” for each machine.

Here’s one way to do it. We can draw a sample of 1,000 values from each posterior distribution, like this:

```
samples = np.array([b.choice(1000)
                    for b in beliefs])
samples.shape
(4, 1000)
```

The result has 4 rows and 1,000 columns. We can use `argmax` to find the index of the largest value in each column:

```
indices = np.argmax(samples, axis=0)
indices.shape
(1000,)
```

The `Pmf` of these indices is the fraction of times each machine yielded the highest values:

```
pmf = Pmf.from_seq(indices)
pmf
```

	probs
0	0.048
1	0.043
2	0.625
3	0.284

These fractions approximate the probability of superiority for each machine. So we could choose the next machine by choosing a value from this `Pmf`.

```
pmf.choice()
```

```
1
```

But that’s a lot of work to choose a single value, and it’s not really necessary, because there’s a shortcut.

If we draw a single random value from each posterior distribution and select the machine that yields the highest value, it turns out that we'll select each machine in proportion to its probability of superiority.

That's what the following function does.

```
def choose(beliefs):
    """Use Thompson sampling to choose a machine.

    Draws a single sample from each distribution.

    returns: index of the machine that yielded the highest value
    """
    ps = [b.choice() for b in beliefs]
    return np.argmax(ps)
```

This function chooses one value from the posterior distribution of each machine and then uses `argmax` to find the index of the machine that yielded the highest value.

Here's an example:

```
choose(beliefs)
3
```

The Strategy

Putting it all together, the following function chooses a machine, plays once, and updates `beliefs`:

```
def choose_play_update(beliefs):
    """Choose a machine, play it, and update beliefs."""

    # choose a machine
    machine = choose(beliefs)

    # play it
    outcome = play(machine)

    # update beliefs
    update(beliefs[machine], outcome)
```

To test it out, let's start again with a fresh set of beliefs and an empty Counter:

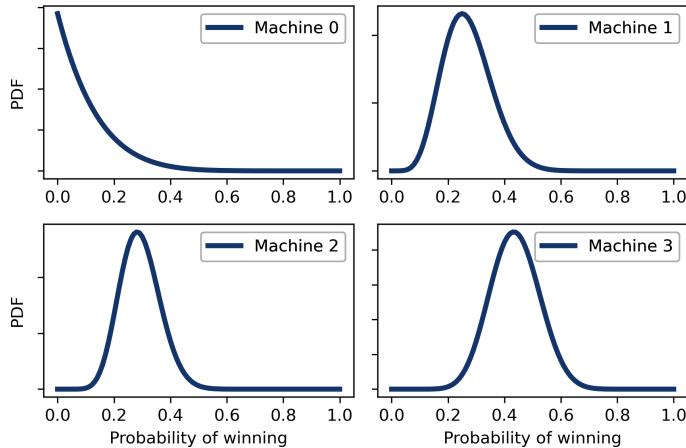
```
beliefs = [prior.copy() for i in range(4)]
counter = Counter()
```

If we run the bandit algorithm 100 times, we can see how beliefs gets updated:

```
num_plays = 100

for i in range(num_plays):
    choose_play_update(beliefs)

plot(beliefs)
```



The following table summarizes the results:

	Actual P(win)	Posterior mean	Credible interval
0	0.1	0.107	[0.0, 0.31]
1	0.2	0.269	[0.14, 0.42]
2	0.3	0.293	[0.18, 0.41]
3	0.4	0.438	[0.3, 0.58]

The credible intervals usually contain the actual probabilities of winning. The estimates are still rough, especially for the lower-probability machines. But that's a feature, not a bug: the goal is to play the high-probability machines most often. Making the estimates more precise is a means to that end, but not an end itself.

More importantly, let's see how many times each machine got played:

	Actual P(win)	Times played
0	0.1	7
1	0.2	24
2	0.3	39
3	0.4	30

If things go according to plan, the machines with higher probabilities should get played more often.

Summary

In this chapter we finally solved the Euro Problem, determining whether the data support the hypothesis that the coin is fair or biased. We found that the answer depends on how we define “biased”. And we summarized the results using a Bayes factor, which quantifies the strength of the evidence.

But the answer wasn’t satisfying because, in my opinion, the question wasn’t interesting. Knowing whether the coin is biased is not useful unless it helps us make better predictions and better decisions.

As an example of a more interesting question, we looked at the One-Armed Bandit problem and a strategy for solving it, the Bayesian Bandit algorithm, which tries to balance exploration and exploitation, that is, gathering more information and making the best use of the information we have.

As an exercise, you’ll have a chance to explore adaptive strategies for standardized testing.

Bayesian bandits and adaptive testing are examples of [Bayesian decision theory](#), which is the idea of using a posterior distribution as part of a decision-making process, often by choosing an action that minimizes the costs we expect on average (or maximizes a benefit).

The strategy we used in “[Maximizing Expected Gain](#)” on page 124 to bid on *The Price is Right* is another example.

These strategies demonstrate what I think is the biggest advantage of Bayesian methods over classical statistics. When we represent knowledge in the form of probability distributions, Bayes’s theorem tells us how to change our beliefs as we get more data, and Bayesian decision theory tells us how to make that knowledge actionable.

More Exercises

Exercise 10-2.

Standardized tests like the [SAT](#) are often used as part of the admission process at colleges and universities. The goal of the SAT is to measure the academic preparation of the test-takers; if it is accurate, their scores should reflect their actual ability in the domain of the test.

Until recently, tests like the SAT were taken with paper and pencil, but now students have the option of taking the test online. In the online format, it is possible for the

test to be “adaptive”, which means that it can **choose each question based on responses to previous questions**.

If a student gets the first few questions right, the test can challenge them with harder questions. If they are struggling, it can give them easier questions. Adaptive testing has the potential to be more “efficient”, meaning that with the same number of questions an adaptive test could measure the ability of a tester more precisely.

To see whether this is true, we will develop a model of an adaptive test and quantify the precision of its measurements.

Details of this exercise are in the notebook.

CHAPTER 11

Comparison

This chapter introduces joint distributions, which are an essential tool for working with distributions of more than one variable.

We'll use them to solve a silly problem on our way to solving a real problem. The silly problem is figuring out how tall two people are, given only that one is taller than the other. The real problem is rating chess players (or participants in other kinds of competition) based on the outcome of a game.

To construct joint distributions and compute likelihoods for these problems, we will use outer products and similar operations. And that's where we'll start.

Outer Operations

Many useful operations can be expressed as the “outer product” of two sequences, or another kind of “outer” operation. Suppose you have sequences like `x` and `y`:

```
x = [1, 3, 5]
y = [2, 4]
```

The outer product of these sequences is an array that contains the product of every pair of values, one from each sequence. There are several ways to compute outer products, but the one I think is the most versatile is a “mesh grid”.

NumPy provides a function called `meshgrid` that computes a mesh grid. If we give it two sequences, it returns two arrays:

```
import numpy as np
X, Y = np.meshgrid(x, y)
```

The first array contains copies of `x` arranged in rows, where the number of rows is the length of `y`:

```
X  
array([[1, 3, 5],  
       [1, 3, 5]])
```

The second array contains copies of y arranged in columns, where the number of columns is the length of x:

```
Y  
array([[2, 2, 2],  
       [4, 4, 4]])
```

Because the two arrays are the same size, we can use them as operands for arithmetic functions like multiplication:

```
X * Y  
array([[ 2,  6, 10],  
       [ 4, 12, 20]])
```

This result is the outer product of x and y. We can see that more clearly if we put it in a DataFrame:

```
import pandas as pd  
  
df = pd.DataFrame(X * Y, columns=x, index=y)  
df
```

1	3	5
2	6	10
4	12	20

The values from x appear as column names; the values from y appear as row labels. Each element is the product of a value from x and a value from y.

We can use mesh grids to compute other operations, like the outer sum, which is an array that contains the *sum* of elements from x and elements from y:

```
X + Y  
array([[3, 5, 7],  
       [5, 7, 9]])
```

We can also use comparison operators to compare elements from x with elements from y:

```
X > Y  
array([[False,  True,  True],  
       [False, False,  True]])
```

The result is an array of Boolean values.

It might not be obvious yet why these operations are useful, but we'll see examples soon. With that, we are ready to take on a new Bayesian problem.

How Tall Is A?

Suppose I choose two people from the population of adult males in the US; I'll call them A and B. If we see that A is taller than B, how tall is A?

To answer this question:

1. I'll use background information about the height of men in the US to form a prior distribution of height,
2. I'll construct a joint prior distribution of height for A and B (and I'll explain what that is),
3. Then I'll update the prior with the information that A is taller, and
4. From the joint posterior distribution I'll extract the posterior distribution of height for A.

In the US the average height of male adults is 178 cm and the standard deviation is 7.7 cm. The distribution is not exactly normal, because nothing in the real world is, but the normal distribution is a pretty good model of the actual distribution, so we can use it as a prior distribution for A and B.

Here's an array of equally-spaced values from 3 standard deviations below the mean to 3 standard deviations above (rounded up a little):

```
mean = 178
qs = np.arange(mean-24, mean+24, 0.5)
```

SciPy provides a function called `norm` that represents a normal distribution with a given mean and standard deviation, and provides `pdf`, which evaluates the probability density function (PDF) of the normal distribution:

```
from scipy.stats import norm

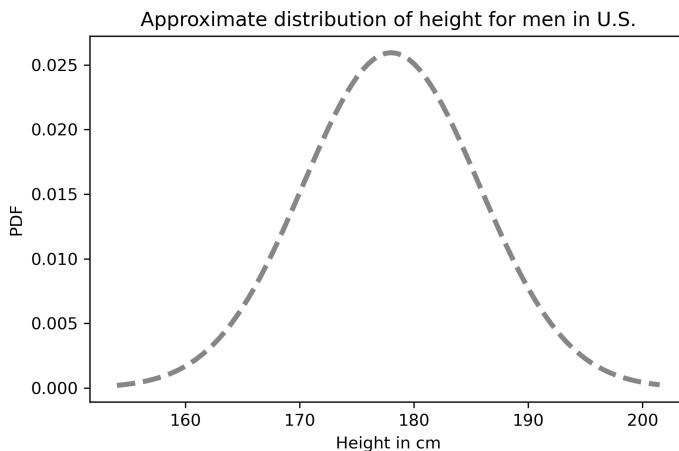
std = 7.7
ps = norm(mean, std).pdf(qs)
```

Probability densities are not probabilities, but if we put them in a `Pmf` and normalize it, the result is a discrete approximation of the normal distribution.

```
from empiricaldist import Pmf

prior = Pmf(ps, qs)
prior.normalize()
```

Here's what it looks like:



This distribution represents what we believe about the heights of A and B before we take into account the data that A is taller.

Joint Distribution

The next step is to construct a distribution that represents the probability of every pair of heights, which is called a joint distribution. The elements of the joint distribution are

$$P(A_x \text{ and } B_y)$$

which is the probability that A is x cm tall and B is y cm tall, for all values of x and y .

At this point all we know about A and B is that they are male residents of the US, so their heights are independent; that is, knowing the height of A provides no additional information about the height of B.

In that case, we can compute the joint probabilities like this:

$$P(A_x \text{ and } B_y) = P(A_x) P(B_y)$$

Each joint probability is the product of one element from the distribution of x and one element from the distribution of y.

So if we have `Pmf` objects that represent the distribution of height for A and B, we can compute the joint distribution by computing the outer product of the probabilities in each `Pmf`.

The following function takes two `Pmf` objects and returns a `DataFrame` that represents the joint distribution.

```
def make_joint(pmf1, pmf2):
    """Compute the outer product of two Pmfs."""
    X, Y = np.meshgrid(pmf1, pmf2)
    return pd.DataFrame(X * Y, columns=pmf1.qs, index=pmf2.qs)
```

The column names in the result are the quantities from `pmf1`; the row labels are the quantities from `pmf2`.

In this example, the prior distributions for A and B are the same, so we can compute the joint prior distribution like this:

```
joint = make_joint(prior, prior)
joint.shape
(96, 96)
```

The result is a `DataFrame` with possible heights of A along the columns, heights of B along the rows, and the joint probabilities as elements.

If the prior is normalized, the joint prior is also normalized.

```
joint.to_numpy().sum()
1.0
```

To add up all of the elements, we convert the `DataFrame` to a NumPy array before calling `sum`. Otherwise, `DataFrame.sum` would compute the sums of the columns and return a `Series`.

Visualizing the Joint Distribution

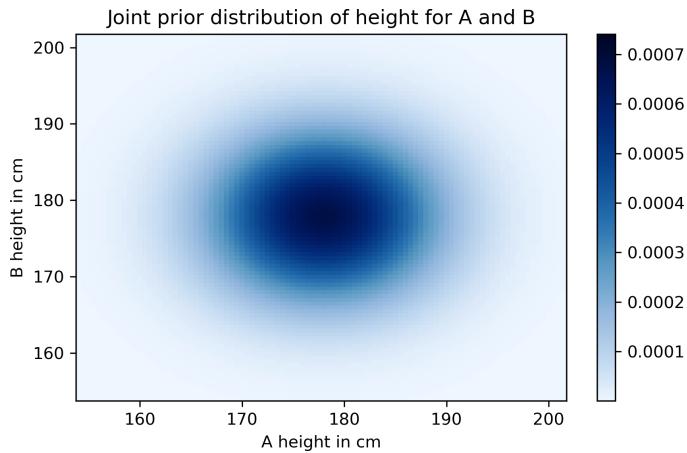
The following function uses `pcolormesh` to plot the joint distribution.

```
import matplotlib.pyplot as plt

def plot_joint(joint, cmap='Blues'):
    """Plot a joint distribution with a color mesh."""
    vmax = joint.to_numpy().max() * 1.1
    plt.pcolormesh(joint.columns, joint.index, joint,
                   cmap=cmap,
                   vmax=vmax,
                   shading='nearest')
    plt.colorbar()

    decorate(xlabel='A height in cm',
              ylabel='B height in cm')
```

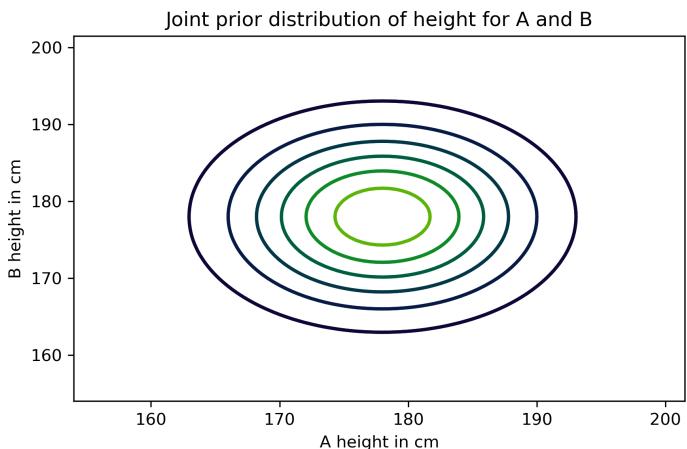
Here's what the joint prior distribution looks like:



As you might expect, the probability is highest (darkest) near the mean and drops off farther from the mean.

Another way to visualize the joint distribution is a contour plot:

```
def plot_contour(joint):
    """Plot a joint distribution with a contour."""
    plt.contour(joint.columns, joint.index, joint,
                linewidths=2)
    decorate(xlabel='A height in cm',
             ylabel='B height in cm')
```



Each line represents a level of equal probability.

Likelihood

Now that we have a joint prior distribution, we can update it with the data, which is that A is taller than B.

Each element in the joint distribution represents a hypothesis about the heights of A and B. To compute the likelihood of every pair of quantities, we can extract the column names and row labels from the prior, like this:

```
x = joint.columns  
y = joint.index
```

And use them to compute a mesh grid:

```
X, Y = np.meshgrid(x, y)
```

X contains copies of the quantities in x, which are possible heights for A. Y contains copies of the quantities in y, which are possible heights for B. If we compare X and Y, the result is a Boolean array:

```
A_taller = (X > Y)  
A_taller.dtype  
dtype('bool')
```

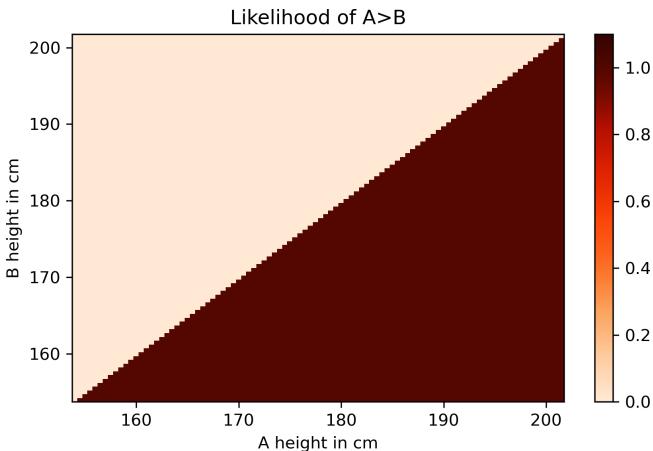
To compute likelihoods, I'll use np.where to make an array with 1 where A_taller is True and 0 elsewhere:

```
a = np.where(A_taller, 1, 0)
```

To visualize this array of likelihoods, I'll put it in a DataFrame with the values of x as column names and the values of y as row labels:

```
likelihood = pd.DataFrame(a, index=y, columns=x)
```

Here's what it looks like:



The likelihood of the data is 1 where $X > Y$ and 0 elsewhere.

The Update

We have a prior, we have a likelihood, and we are ready for the update. As usual, the unnormalized posterior is the product of the prior and the likelihood.

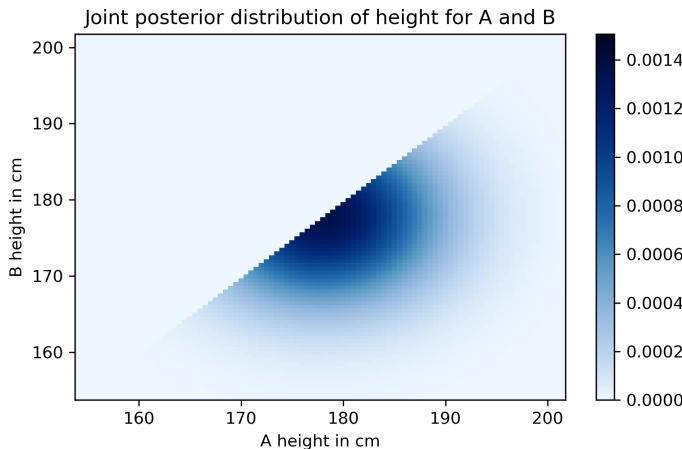
```
posterior = joint * likelihood
```

I'll use the following function to normalize the posterior:

```
def normalize(joint):
    """Normalize a joint distribution."""
    prob_data = joint.to_numpy().sum()
    joint /= prob_data
    return prob_data

normalize(posterior)
```

And here's what it looks like:



All pairs where B is taller than A have been eliminated. The rest of the posterior looks the same as the prior, except that it has been renormalized.

Marginal Distributions

The joint posterior distribution represents what we believe about the heights of A and B given the prior distributions and the information that A is taller.

From this joint distribution, we can compute the posterior distributions for A and B. To see how, let's start with a simpler problem.

Suppose we want to know the probability that A is 180 cm tall. We can select the column from the joint distribution where $x=180$:

```
column = posterior[180]
column.head()

154.0    0.000010
154.5    0.000013
155.0    0.000015
155.5    0.000019
156.0    0.000022
Name: 180.0, dtype: float64
```

This column contains posterior probabilities for all cases where $x=180$; if we add them up, we get the total probability that A is 180 cm tall.

```
column.sum()

0.03017221271570807
```

It's about 3%.

Now, to get the posterior distribution of height for A, we can add up all of the columns, like this:

```
column_sums = posterior.sum(axis=0)
column_sums.head()

154.0    0.000000e+00
154.5    1.012260e-07
155.0    2.736152e-07
155.5    5.532519e-07
156.0    9.915650e-07
dtype: float64
```

The argument `axis=0` means we want to add up the columns.

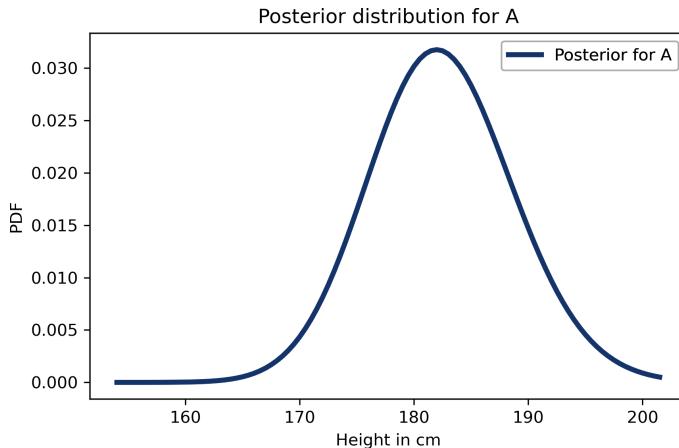
The result is a `Series` that contains every possible height for A and its probability. In other words, it is the distribution of heights for A.

We can put it in a `Pmf` like this:

```
marginal_A = Pmf(column_sums)
```

When we extract the distribution of a single variable from a joint distribution, the result is called a **marginal distribution**. The name comes from a common visualization that shows the joint distribution in the middle and the marginal distributions in the margins.

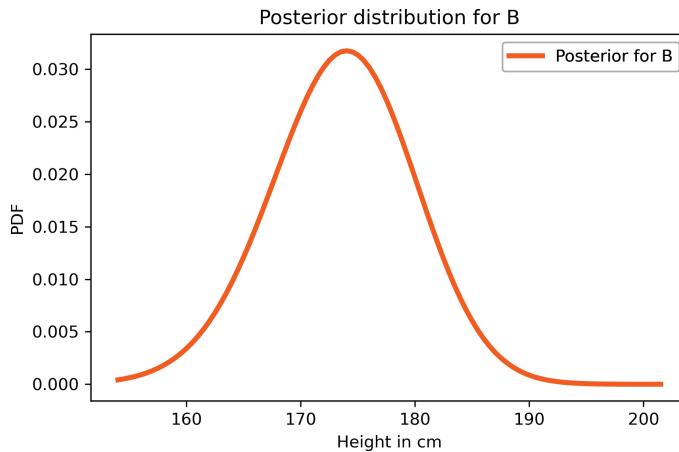
Here's what the marginal distribution for A looks like:



Similarly, we can get the posterior distribution of height for B by adding up the rows and putting the result in a Pmf:

```
row_sums = posterior.sum(axis=1)
marginal_B = Pmf(row_sums)
```

Here's what it looks like:



Let's put the code from this section in a function:

```
def marginal(joint, axis):
    """Compute a marginal distribution."""
    return Pmf(joint.sum(axis=axis))
```

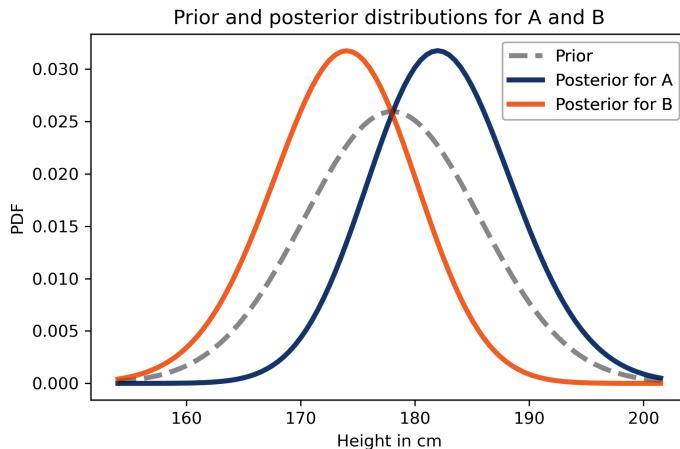
`marginal` takes as parameters a joint distribution and an axis number:

- If `axis=0`, it returns the marginal of the first variable (the one on the x -axis);
- If `axis=1`, it returns the marginal of the second variable (the one on the y -axis).

So we can compute both marginals like this:

```
marginal_A = marginal(posterior, axis=0)
marginal_B = marginal(posterior, axis=1)
```

Here's what they look like, along with the prior:



As you might expect, the posterior distribution for A is shifted to the right and the posterior distribution for B is shifted to the left.

We can summarize the results by computing the posterior means:

```
prior.mean()
177.99516026921506
print(marginal_A.mean(), marginal_B.mean())
182.3872812342168 173.6028600023339
```

Based on the observation that A is taller than B, we are inclined to believe that A is a little taller than average, and B is a little shorter.

Notice that the posterior distributions are a little narrower than the prior. We can quantify that by computing their standard deviations:

```
prior.std()
7.624924796641578
print(marginal_A.std(), marginal_B.std())
6.270461177645469 6.280513548175111
```

The standard deviations of the posterior distributions are a little smaller, which means we are more certain about the heights of A and B after we compare them.

Conditional Posteriors

Now suppose we measure A and find that he is 170 cm tall. What does that tell us about B?

In the joint distribution, each column corresponds to a possible height for A. We can select the column that corresponds to height 170 cm like this:

```
column_170 = posterior[170]
```

The result is a Series that represents possible heights for B and their relative likelihoods. These likelihoods are not normalized, but we can normalize them like this:

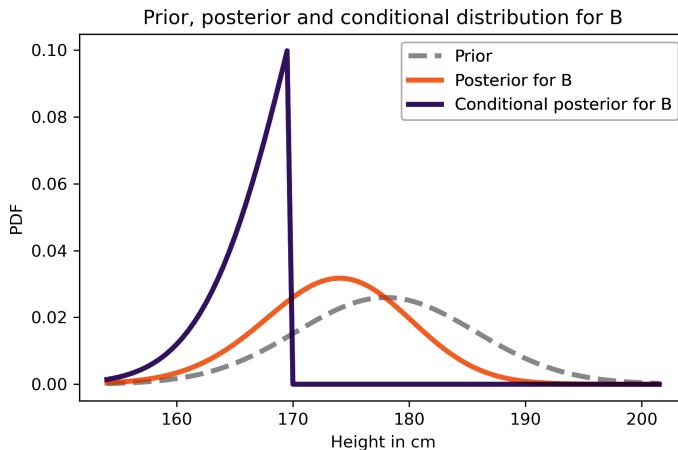
```
cond_B = Pmf(column_170)
```

```
cond_B.normalize()
```

```
0.004358061205454471
```

Making a Pmf copies the data by default, so we can normalize `cond_B` without affecting `column_170` or `posterior`. The result is the conditional distribution of height for B given that A is 170 cm tall.

Here's what it looks like:



The conditional posterior distribution is cut off at 170 cm, because we have established that B is shorter than A, and A is 170 cm.

Dependence and Independence

When we constructed the joint prior distribution, I said that the heights of A and B were independent, which means that knowing one of them provides no information about the other. In other words, the conditional probability $P(A_x | B_y)$ is the same as the unconditional probability $P(A_x)$.

But in the posterior distribution, A and B are not independent. If we know that A is taller than B, and we know how tall A is, that gives us information about B.

The conditional distribution we just computed demonstrates this dependence.

Summary

In this chapter we started with the “outer” operations, like outer product, which we used to construct a joint distribution.

In general, you cannot construct a joint distribution from two marginal distributions, but in the special case where the distributions are independent, you can.

We extended the Bayesian update process and applied it to a joint distribution. Then from the posterior joint distribution we extracted marginal posterior distributions and conditional posterior distributions.

As an exercise, you’ll have a chance to apply the same process to a problem that’s a little more difficult and a lot more useful, updating a chess player’s rating based on the outcome of a game.

Exercises

Exercise 11-1.

Based on the results of the previous example, compute the posterior conditional distribution for A given that B is 180 cm.

Hint: Use `loc` to select a row from a `DataFrame`.

Exercise 11-2.

Suppose we have established that A is taller than B, but we don’t know how tall B is. Now we choose a random woman, C, and find that she is shorter than A by at least 15 cm. Compute posterior distributions for the heights of A and C.

The average height for women in the US is 163 cm; the standard deviation is 7.3 cm.

Exercise 11-3.

The [Elo rating system](#) is a way to quantify the skill level of players for games like chess.

It is based on a model of the relationship between the ratings of players and the outcome of a game. Specifically, if R_A is the rating of player A and R_B is the rating of player B, the probability that A beats B is given by the [logistic function](#):

$$P(A \text{ beats } B) = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

The parameters 10 and 400 are arbitrary choices that determine the range of the ratings. In chess, the range is from 100 to 2,800.

Notice that the probability of winning depends only on the difference in rankings. As an example, if R_A exceeds R_B by 100 points, the probability that A wins is:

```
1 / (1 + 10**(-100/400))
```

```
0.6400649998028851
```

Suppose A has a current rating of 1,600, but we are not sure it is accurate. We could describe their true rating with a normal distribution with mean 1,600 and standard deviation 100, to indicate our uncertainty.

And suppose B has a current rating of 1,800, with the same level of uncertainty.

Then A and B play and A wins. How should we update their ratings?

CHAPTER 12

Classification

Classification might be the most well-known application of Bayesian methods, made famous in the 1990s as the basis of the first generation of **spam filters**.

In this chapter, I'll demonstrate Bayesian classification using data collected and made available by Dr. Kristen Gorman at the Palmer Long-Term Ecological Research Station in Antarctica (see Gorman, Williams, and Fraser, “[Ecological Sexual Dimorphism and Environmental Variability within a Community of Antarctic Penguins \(Genus *Pygoscelis*\)](#)”, March 2014). We'll use this data to classify penguins by species.

Penguin Data

I'll use pandas to load the data into a `DataFrame`:

```
import pandas as pd  
  
df = pd.read_csv('penguins_raw.csv')  
df.shape  
(344, 17)
```

The dataset contains one row for each penguin and one column for each variable.

Three species of penguins are represented in the dataset: Adélie, Chinstrap and Gentoo.

The measurements we'll use are:

- Body Mass in grams (g).
- Flipper Length in millimeters (mm).
- Culmen Length in millimeters.
- Culmen Depth in millimeters.

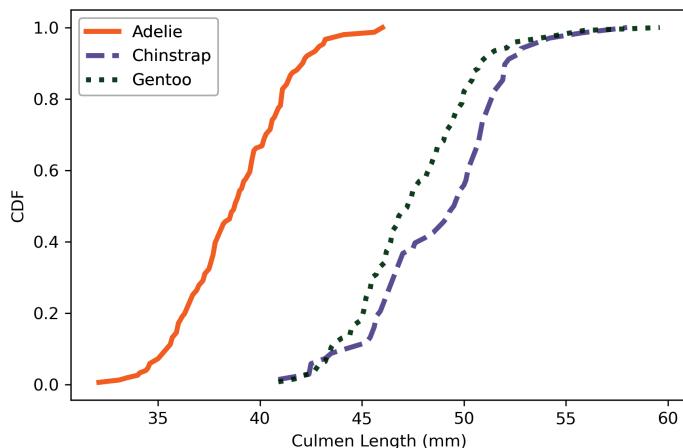
If you are not familiar with the word “culmen”, it refers to the **top margin of the beak**.

These measurements will be most useful for classification if there are substantial differences between species and small variation within species. To see whether that is true, and to what degree, I’ll plot cumulative distribution functions (CDFs) of each measurement for each species.

The following function takes the `DataFrame` and a column name. It returns a dictionary that maps from each species name to a `Cdf` of the values in the column named `colname`.

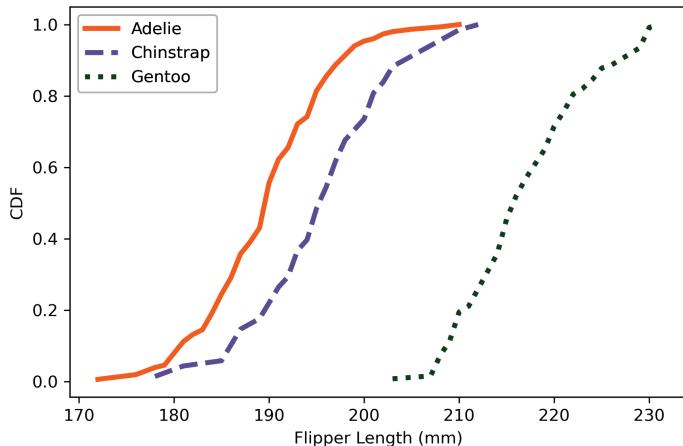
```
def make_cdf_map(df, colname, by='Species2'):
    """Make a CDF for each species."""
    cdf_map = {}
    grouped = df.groupby(by)[colname]
    for species, group in grouped:
        cdf_map[species] = Cdf.from_seq(group, name=species)
    return cdf_map
```

Here’s what the distributions look like for culmen length:



It looks like we can use culmen length to identify Adélie penguins, but the distributions for the other two species almost entirely overlap.

Here are the distributions for flipper length:



Using flipper length, we can distinguish Gentoo penguins from the other two species. So with just these two features, it seems like we should be able to classify penguins with some accuracy.

All of these CDFs show the sigmoid shape characteristic of the normal distribution; I will take advantage of that observation in the next section.

Normal Models

Let's use these features to classify penguins. We'll proceed in the usual Bayesian way:

1. Define a prior distribution with the three possible species and a prior probability for each,
2. Compute the likelihood of the data for each hypothetical species, and then
3. Compute the posterior probability of each hypothesis.

To compute the likelihood of the data under each hypothesis, I'll use the data to estimate the parameters of a normal distribution for each species.

The following function takes a `DataFrame` and a column name; it returns a dictionary that maps from each species name to a `norm` object.

`norm` is defined in SciPy; it represents a normal distribution with a given mean and standard deviation.

```
from scipy.stats import norm

def make_norm_map(df, colname, by='Species2'):
    """Make a map from species to norm object."""
    norm_map = {}
```

```
grouped = df.groupby(by)[colname]
for species, group in grouped:
    mean = group.mean()
    std = group.std()
    norm_map[species] = norm(mean, std)
return norm_map
```

For example, here's the dictionary of `norm` objects for flipper length:

```
flipper_map = make_norm_map(df, 'Flipper Length (mm)')
flipper_map.keys()

dict_keys(['Adelie', 'Chinstrap', 'Gentoo'])
```

Now suppose we measure a penguin and find that its flipper is 193 cm. What is the probability of that measurement under each hypothesis?

The `norm` object provides `pdf`, which computes the probability density function (PDF) of the normal distribution. We can use it to compute the likelihood of the observed data in a given distribution.

```
data = 193
flipper_map['Adelie'].pdf(data)

0.054732511875530694
```

The result is a probability density, so we can't interpret it as a probability. But it is proportional to the likelihood of the data, so we can use it to update the prior.

Here's how we compute the likelihood of the data in each distribution:

```
hypos = flipper_map.keys()
likelihood = [flipper_map[hypo].pdf(data) for hypo in hypos]
likelihood

[0.054732511875530694, 0.05172135615888162, 5.8660453661990634e-05]
```

Now we're ready to do the update.

The Update

As usual I'll use a `Pmf` to represent the prior distribution. For simplicity, let's assume that the three species are equally likely.

```
from empiricaldist import Pmf

prior = Pmf(1/3, hypos)
prior
```

probs	
Adelie	0.333333
Chinstrap	0.333333
Gentoo	0.333333

Now we can do the update in the usual way:

```
posterior = prior * likelihood
posterior.normalize()
posterior
```

probs	
Adelie	0.513860
Chinstrap	0.485589
Gentoo	0.000551

A penguin with a 193 mm flipper is unlikely to be a Gentoo, but might be either an Adélie or a Chinstrap (assuming that the three species were equally likely before the measurement).

The following function encapsulates the steps we just ran. It takes a `Pmf` representing the prior distribution, the observed data, and a map from each hypothesis to the distribution of the feature.

```
def update_penguin(prior, data, norm_map):
    """Update hypothetical species."""
    hypos = prior.qs
    likelihood = [norm_map[hypo].pdf(data) for hypo in hypos]
    posterior = prior * likelihood
    posterior.normalize()
    return posterior
```

The return value is the posterior distribution.

Here's the previous example again, using `update_penguin`:

```
posterior1 = update_penguin(prior, 193, flipper_map)
posterior1
```

probs	
Adelie	0.513860
Chinstrap	0.485589
Gentoo	0.000551

As we saw in the CDFs, flipper length does not distinguish strongly between Adélie and Chinstrap penguins.

But culmen length *can* make this distinction, so let's use it to do a second round of classification. First we estimate distributions of culmen length for each species like this:

```
culmen_map = make_norm_map(df, 'Culmen Length (mm)')
```

Now suppose we see a penguin with culmen length 48 mm. We can use this data to update the prior:

```
posterior2 = update_penguin(prior, 48, culmen_map)
posterior2
```

	probs
Adelie	0.001557
Chinstrap	0.474658
Gentoo	0.523785

A penguin with culmen length 48 mm is about equally likely to be a Chinstrap or a Gentoo.

Using one feature at a time, we can often rule out one species or another, but we generally can't identify species with confidence. We can do better using multiple features.

Naive Bayesian Classification

To make it easier to do multiple updates, I'll use the following function, which takes a prior Pmf, a sequence of measurements and a corresponding sequence of dictionaries containing estimated distributions.

```
def update_naive(prior, data_seq, norm_maps):
    """Naive Bayesian classifier

    prior: Pmf
    data_seq: sequence of measurements
    norm_maps: sequence of maps from species to distribution

    returns: Pmf representing the posterior distribution
    """
    posterior = prior.copy()
    for data, norm_map in zip(data_seq, norm_maps):
        posterior = update_penguin(posterior, data, norm_map)
    return posterior
```

It performs a series of updates, using one variable at a time, and returns the posterior Pmf. To test it, I'll use the same features we looked at in the previous section: culmen length and flipper length.

```
colnames = ['Flipper Length (mm)', 'Culmen Length (mm)']
norm_maps = [flipper_map, culmen_map]
```

Now suppose we find a penguin with flipper length 193 mm and culmen length 48. Here's the update:

```
data_seq = 193, 48
posterior = update_naive(prior, data_seq, norm_maps)
posterior
```

	probs
Adelie	0.003455
Chinstrap	0.995299
Gentoo	0.001246

It is almost certain to be a Chinstrap:

```
posterior.max_prob()
'Chinstrap'
```

We can loop through the dataset and classify each penguin with these two features:

```
import numpy as np

df['Classification'] = np.nan

for i, row in df.iterrows():
    data_seq = row[colnames]
    posterior = update_naive(prior, data_seq, norm_maps)
    df.loc[i, 'Classification'] = posterior.max_prob()
```

This loop adds a column called Classification to the DataFrame; it contains the species with the maximum posterior probability for each penguin.

So let's see how many we got right:

```
valid = df['Classification'].notna()
valid.sum()

342

same = df['Species2'] == df['Classification']
same.sum()

324
```

There are 344 penguins in the dataset, but two of them are missing measurements, so we have 342 valid cases. Of those, 324 are classified correctly, which is almost 95%:

```
same.sum() / valid.sum()

0.9473684210526315
```

The following function encapsulates these steps.

```

def accuracy(df):
    """Compute the accuracy of classification."""
    valid = df['Classification'].notna()
    same = df['Species2'] == df['Classification']
    return same.sum() / valid.sum()

```

The classifier we used in this section is called “naive” because it ignores correlations between the features. To see why that matters, I’ll make a less naive classifier: one that takes into account the joint distribution of the features.

Joint Distributions

I’ll start by making a scatter plot of the data:

```

import matplotlib.pyplot as plt

def scatterplot(df, var1, var2):
    """Make a scatter plot."""
    grouped = df.groupby('Species2')
    for species, group in grouped:
        plt.plot(group[var1], group[var2],
                 label=species, lw=0, alpha=0.3)

    decorate(xlabel=var1, ylabel=var2)

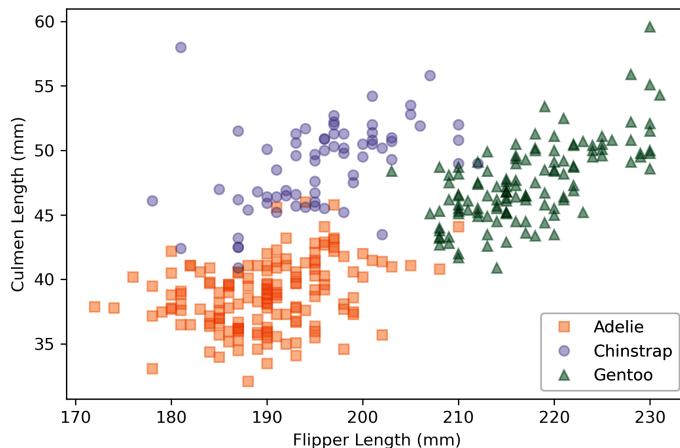
```

Here’s a scatter plot of culmen length and flipper length for the three species:

```

var1 = 'Flipper Length (mm)'
var2 = 'Culmen Length (mm)'
scatterplot(df, var1, var2)

```



Within each species, the joint distribution of these measurements forms an oval shape, at least roughly. The orientation of the ovals is along a diagonal, which indicates that there is a correlation between culmen length and flipper length.

If we ignore these correlations, we assume that the features are independent. To see what that looks like, I'll make a joint distribution for each species assuming independence.

The following function makes a discrete Pmf that approximates a normal distribution.

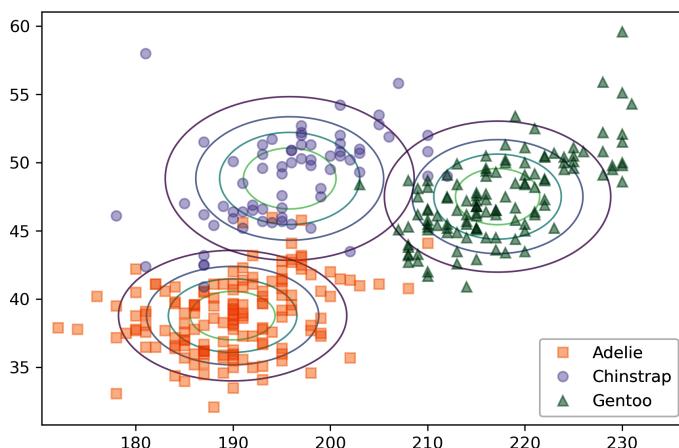
```
def make_pmf_norm(dist, sigmas=3, n=101):
    """Make a Pmf approximation to a normal distribution."""
    mean, std = dist.mean(), dist.std()
    low = mean - sigmas * std
    high = mean + sigmas * std
    qs = np.linspace(low, high, n)
    ps = dist.pdf(qs)
    pmf = Pmf(ps, qs)
    pmf.normalize()
    return pmf
```

We can use it, along with `make_joint`, to make a joint distribution of culmen length and flipper length for each species:

```
from utils import make_joint

joint_map = {}
for species in hypers:
    pmf1 = make_pmf_norm(flipper_map[species])
    pmf2 = make_pmf_norm(culmen_map[species])
    joint_map[species] = make_joint(pmf1, pmf2)
```

The following figure compares a scatter plot of the data to the contours of the joint distributions, assuming independence.



The contours of a joint normal distribution form ellipses. In this example, because the features are uncorrelated, the ellipses are aligned with the axes. But they are not well aligned with the data.

We can make a better model of the data, and use it to compute better likelihoods, with a multivariate normal distribution.

Multivariate Normal Distribution

As we have seen, a univariate normal distribution is characterized by its mean and standard deviation.

A multivariate normal distribution is characterized by the means of the features and the **covariance matrix**, which contains **variances**, which quantify the spread of the features, and the **covariances**, which quantify the relationships among them.

We can use the data to estimate the means and covariance matrix for the population of penguins. First I'll select the columns we want:

```
features = df[[var1, var2]]
```

And compute the means:

```
mean = features.mean()  
mean  
  
Flipper Length (mm)    200.915205  
Culmen Length (mm)    43.921930  
dtype: float64
```

We can also compute the covariance matrix:

```
cov = features.cov()  
cov
```

	Flipper Length (mm)	Culmen Length (mm)
Flipper Length (mm)	197.731792	50.375765
Culmen Length (mm)	50.375765	29.807054

The result is a `DataFrame` with one row and one column for each feature. The elements on the diagonal are the variances; the elements off the diagonal are covariances.

By themselves, variances and covariances are hard to interpret. We can use them to compute standard deviations and correlation coefficients, which are easier to interpret, but the details of that calculation are not important right now.

Instead, we'll pass the covariance matrix to `multivariate_normal`, which is a SciPy function that creates an object that represents a multivariate normal distribution.

As arguments it takes a sequence of means and a covariance matrix:

```
from scipy.stats import multivariate_normal  
  
multinorm = multivariate_normal(mean, cov)
```

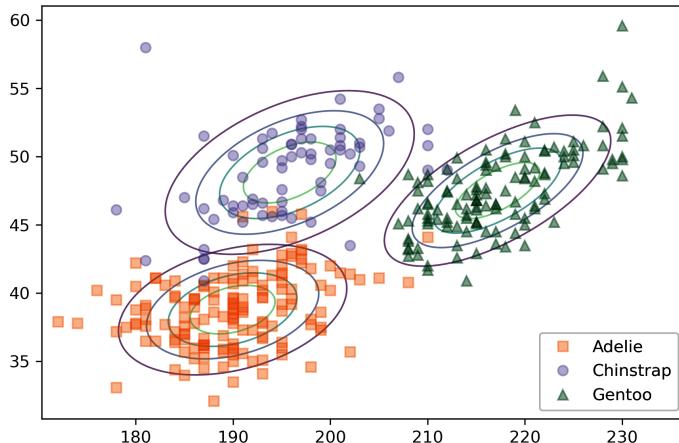
The following function makes a `multivariate_normal` object for each species.

```
def make_multinorm_map(df, colnames):  
    """Make a map from each species to a multivariate normal."""  
    multinorm_map = {}  
    grouped = df.groupby('Species2')  
    for species, group in grouped:  
        features = group[colnames]  
        mean = features.mean()  
        cov = features.cov()  
        multinorm_map[species] = multivariate_normal(mean, cov)  
    return multinorm_map
```

Here's how we make this map for the first two features, flipper length and culmen length:

```
multinorm_map = make_multinorm_map(df, [var1, var2])
```

The following figure shows a scatter plot of the data along with the contours of the multivariate normal distribution for each species:



Because the multivariate normal distribution takes into account the correlations between features, it is a better model for the data. And there is less overlap in the contours of the three distributions, which suggests that they should yield better classifications.

A Less Naive Classifier

In a previous section we used `update_penguin` to update a prior `pmf` based on observed data and a collection of `norm` objects that model the distribution of observations under each hypothesis. Here it is again:

```
def update_penguin(prior, data, norm_map):
    """Update hypothetical species."""
    hypos = prior.qs
    likelihood = [norm_map[hypo].pdf(data) for hypo in hypos]
    posterior = prior * likelihood
    posterior.normalize()
    return posterior
```

Last time we used this function, the values in `norm_map` were `norm` objects, but it also works if they are `multivariate_normal` objects.

We can use it to classify a penguin with flipper length 193 and culmen length 48:

```
data = 193, 48
update_penguin(prior, data, multinorm_map)
```

	probs
Adelie	0.002740
Chinstrap	0.997257
Gentoo	0.000003

A penguin with those measurements is almost certainly a Chinstrap.

Now let's see if this classifier does any better than the naive Bayesian classifier. I'll apply it to each penguin in the dataset:

```
df['Classification'] = np.nan

for i, row in df.iterrows():
    data = row[colnames]
    posterior = update_penguin(prior, data, multinorm_map)
    df.loc[i, 'Classification'] = posterior.idxmax()
```

And compute the accuracy:

```
accuracy(df)

0.9532163742690059
```

It turns out to be only a little better: the accuracy is 95.3%, compared to 94.7% for the naive Bayesian classifier.

Summary

In this chapter, we implemented a naive Bayesian classifier, which is “naive” in the sense that it assumes that the features it uses for classification are independent.

To see how bad that assumption is, we also implemented a classifier that uses the multivariate normal distribution to model the joint distribution of the features, which includes their dependencies.

In this example, the non-naive classifier is only marginally better. In one way, that’s disappointing. After all that work, it would have been nice to see a bigger improvement. But in another way, it’s good news. In general, a naive Bayesian classifier is easier to implement and requires less computation. If it works nearly as well as a more complex algorithm, it might be a good choice for practical purposes.

Speaking of practical purposes, you might have noticed that this example isn’t very useful. If we want to identify the species of a penguin, there are easier ways than measuring its flippers and beak.

But there *are* scientific uses for this type of classification. One of them is the subject of the research paper we started with: **sexual dimorphism**, that is, differences in shape between male and female animals.

In some species, like angler fish, males and females look very different. In other species, like mockingbirds, they are difficult to tell apart. And dimorphism is worth studying because it provides insight into social behavior, sexual selection, and evolution.

One way to quantify the degree of sexual dimorphism in a species is to use a classification algorithm like the one in this chapter. If you can find a set of features that makes it possible to classify individuals by sex with high accuracy, that’s evidence of high dimorphism.

As an exercise, you can use the dataset from this chapter to classify penguins by sex and see which of the three species is the most dimorphic.

Exercises

Exercise 12-1.

In my example I used culmen length and flipper length because they seemed to provide the most power to distinguish the three species. But maybe we can do better by using more features.

Make a naive Bayesian classifier that uses all four measurements in the dataset: culmen length and depth, flipper length, and body mass. Is it more accurate than the model with two features?

Exercise 12-2.

One of the reasons the penguin dataset was collected was to quantify sexual dimorphism in different penguin species, that is, physical differences between male and female penguins. One way to quantify dimorphism is to use measurements to classify penguins by sex. If a species is more dimorphic, we expect to be able to classify them more accurately.

As an exercise, pick a species and use a Bayesian classifier (naive or not) to classify the penguins by sex. Which features are most useful? What accuracy can you achieve?

Whenever people compare Bayesian inference with conventional approaches, one of the questions that comes up most often is something like, “What about p-values?” And one of the most common examples is the comparison of two groups to see if there is a difference in their means.

In classical statistical inference, the usual tool for this scenario is a Student’s *t*-test, and the result is a **p-value**. This process is an example of **null hypothesis significance testing**.

A Bayesian alternative is to compute the posterior distribution of the difference between the groups. Then we can use that distribution to answer whatever questions we are interested in, including the most likely size of the difference, a credible interval that’s likely to contain the true difference, the probability of superiority, or the probability that the difference exceeds some threshold.

To demonstrate this process, I’ll solve a problem borrowed from a statistical textbook: evaluating the effect of an educational “treatment” compared to a control.

Improving Reading Ability

We’ll use data from a **PhD dissertation in educational psychology** written in 1987, which was used as an example in a **statistics textbook** from 1989 and published on **DASL**, a web page that collects data stories.

Here's the description from DASL:

An educator conducted an experiment to test whether new directed reading activities in the classroom will help elementary school pupils improve some aspects of their reading ability. She arranged for a third grade class of 21 students to follow these activities for an 8-week period. A control classroom of 23 third graders followed the same curriculum without the activities. At the end of the 8 weeks, all students took a Degree of Reading Power (DRP) test, which measures the aspects of reading ability that the treatment is designed to improve.

The dataset is available [here](#). I'll use pandas to load the data into a DataFrame:

```
import pandas as pd

df = pd.read_csv('drp_scores.csv', skiprows=21, delimiter='\t')
df.head(3)
```

	Treatment	Response
0	Treated	24
1	Treated	43
2	Treated	58

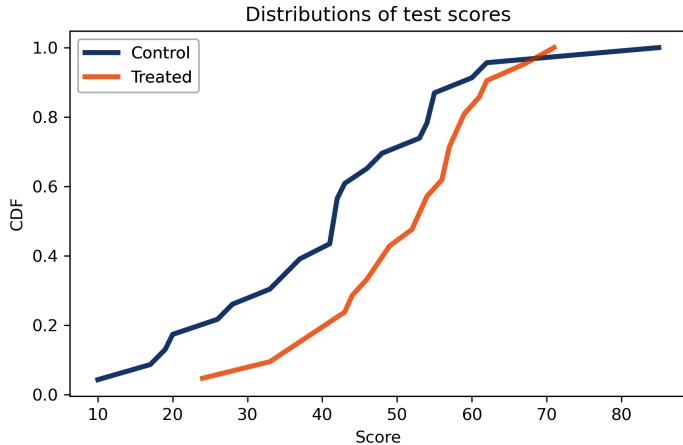
The Treatment column indicates whether each student was in the treated or control group. The Response is their score on the test.

I'll use groupby to separate the data for the Treated and Control groups:

```
grouped = df.groupby('Treatment')
responses = {}

for name, group in grouped:
    responses[name] = group['Response']
```

Here are CDFs of the scores for the two groups and summary statistics:



There is overlap between the distributions, but it looks like the scores are higher in the treated group. The distribution of scores is not exactly normal for either group, but it is close enough that the normal model is a reasonable choice.

So I'll assume that in the entire population of students (not just the ones in the experiment), the distribution of scores is well modeled by a normal distribution with unknown mean and standard deviation. I'll use `mu` and `sigma` to denote these unknown parameters, and we'll do a Bayesian update to estimate what they are.

Estimating Parameters

As always, we need a prior distribution for the parameters. Since there are two parameters, it will be a joint distribution. I'll construct it by choosing marginal distributions for each parameter and computing their outer product.

As a simple starting place, I'll assume that the prior distributions for `mu` and `sigma` are uniform. The following function makes a `Pmf` object that represents a uniform distribution.

```
from empiricaldist import Pmf

def make_uniform(qs, name=None, **options):
    """Make a Pmf that represents a uniform distribution."""
    pmf = Pmf(1.0, qs, **options)
    pmf.normalize()
    if name:
        pmf.index.name = name
    return pmf
```

`make_uniform` takes as parameters:

- An array of quantities, `qs`, and
- A string, `name`, which is assigned to the index so it appears when we display the `Pmf`.

Here's the prior distribution for `mu`:

```
import numpy as np

qs = np.linspace(20, 80, num=101)
prior_mu = make_uniform(qs, name='mean')
```

I chose the lower and upper bounds by trial and error. I'll explain how when we look at the posterior distribution.

Here's the prior distribution for `sigma`:

```
qs = np.linspace(5, 30, num=101)
prior_sigma = make_uniform(qs, name='std')
```

Now we can use `make_joint` to make the joint prior distribution:

```
from utils import make_joint

prior = make_joint(prior_mu, prior_sigma)
```

And we'll start by working with the data from the control group:

```
data = responses['Control']
data.shape

(23,)
```

In the next section we'll compute the likelihood of this data for each pair of parameters in the prior distribution.

Likelihood

We would like to know the probability of each score in the dataset for each hypothetical pair of values, `mu` and `sigma`. I'll do that by making a 3-dimensional grid with values of `mu` on the first axis, values of `sigma` on the second axis, and the scores from the dataset on the third axis:

```
mu_mesh, sigma_mesh, data_mesh = np.meshgrid(
    prior.columns, prior.index, data)

mu_mesh.shape

(101, 101, 23)
```

Now we can use `norm.pdf` to compute the probability density of each score for each hypothetical pair of parameters:

```
from scipy.stats import norm

densities = norm(mu_mesh, sigma_mesh).pdf(data_mesh)
densities.shape

(101, 101, 23)
```

The result is a 3-D array. To compute likelihoods, I'll multiply these densities along `axis=2`, which is the axis of the data:

```
likelihood = densities.prod(axis=2)
likelihood.shape

(101, 101)
```

The result is a 2-D array that contains the likelihood of the entire dataset for each hypothetical pair of parameters.

We can use this array to update the prior, like this:

```
from utils import normalize

posterior = prior * likelihood
normalize(posterior)
posterior.shape

(101, 101)
```

The result is a `DataFrame` that represents the joint posterior distribution.

The following function encapsulates these steps.

```
def update_norm(prior, data):
    """Update the prior based on data."""
    mu_mesh, sigma_mesh, data_mesh = np.meshgrid(
        prior.columns, prior.index, data)

    densities = norm(mu_mesh, sigma_mesh).pdf(data_mesh)
    likelihood = densities.prod(axis=2)

    posterior = prior * likelihood
    normalize(posterior)

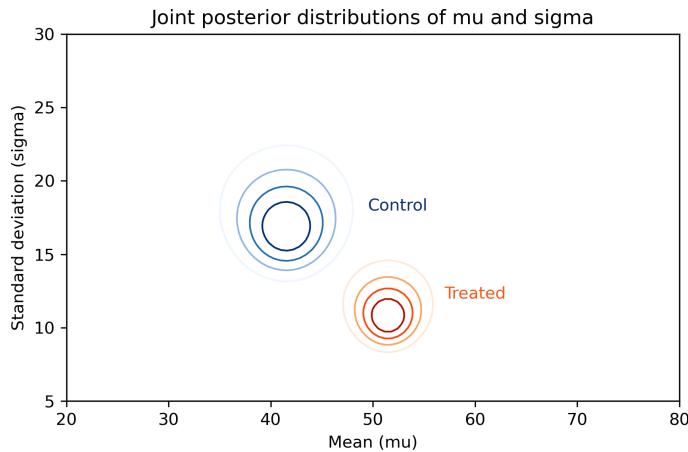
    return posterior
```

Here are the updates for the control and treatment groups:

```
data = responses['Control']
posterior_control = update_norm(prior, data)

data = responses['Treated']
posterior_treated = update_norm(prior, data)
```

And here's what they look like:



Along the x -axis, it looks like the mean score for the treated group is higher. Along the y -axis, it looks like the standard deviation for the treated group is lower.

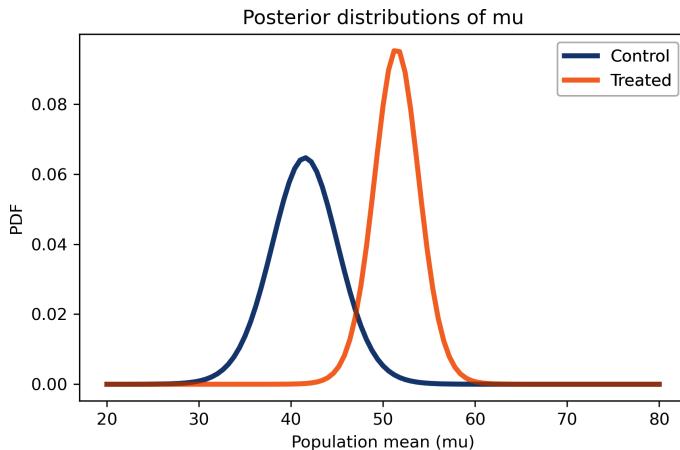
If we think the treatment causes these differences, the data suggest that the treatment increases the mean of the scores and decreases their spread. We can see these differences more clearly by looking at the marginal distributions for μ and σ .

Posterior Marginal Distributions

I'll use `marginal`, which we saw in “[Marginal Distributions](#)” on page 153, to extract the posterior marginal distributions for the population means:

```
from utils import marginal  
  
pmf_mean_control = marginal(posterior_control, 0)  
pmf_mean_treated = marginal(posterior_treated, 0)
```

Here's what they look like:



In both cases the posterior probabilities at the ends of the range are near zero, which means that the bounds we chose for the prior distribution are wide enough.

Comparing the marginal distributions for the two groups, it looks like the population mean in the treated group is higher. We can use `prob_gt` to compute the probability of superiority:

```
Pmf.prob_gt(pmf_mean_treated, pmf_mean_control)
```

```
0.980479025187326
```

There is a 98% chance that the mean in the treated group is higher.

Distribution of Differences

To quantify the magnitude of the difference between groups, we can use `sub_dist` to compute the distribution of the difference:

```
pmf_diff = Pmf.sub_dist(pmf_mean_treated, pmf_mean_control)
```

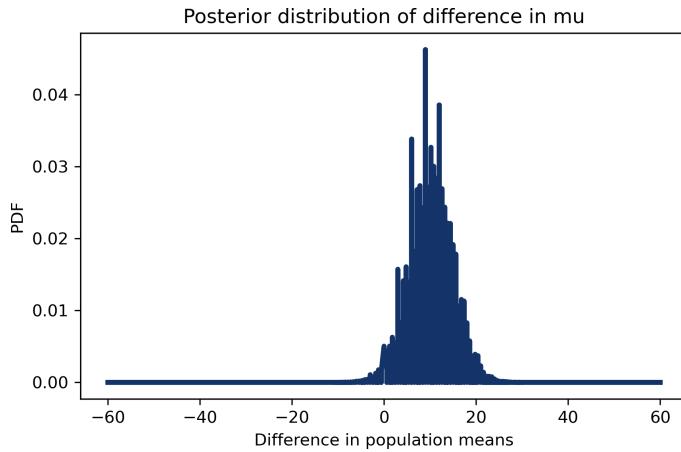
There are two things to be careful about when you use methods like `sub_dist`. The first is that the result usually contains more elements than the original `Pmf`. In this example, the original distributions have the same quantities, so the size increase is moderate.

```
len(pmf_mean_treated), len(pmf_mean_control), len(pmf_diff)
```

```
(101, 101, 879)
```

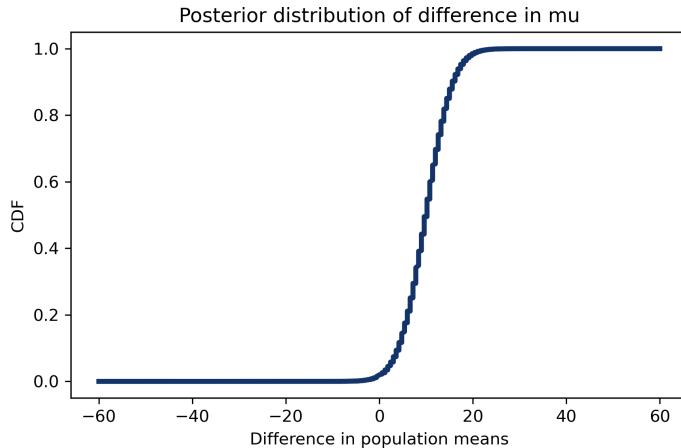
In the worst case, the size of the result can be the product of the sizes of the originals.

The other thing to be careful about is plotting the `pmf`. In this example, if we plot the distribution of differences, the result is pretty noisy:



There are two ways to work around that limitation. One is to plot the CDF, which smooths out the noise:

```
cdf_diff = pmf_diff.make_cdf()
```



The other option is to use kernel density estimation (KDE) to make a smooth approximation of the PDF on an equally-spaced grid, which is what this function does:

```

from scipy.stats import gaussian_kde

def kde_from_pmf(pmf, n=101):
    """Make a kernel density estimate for a PMF."""
    kde = gaussian_kde(pmf.qs, weights=pmf.ps)
    qs = np.linspace(pmf.qs.min(), pmf.qs.max(), n)
    ps = kde.evaluate(qs)
    pmf = Pmf(ps, qs)
    pmf.normalize()
    return pmf

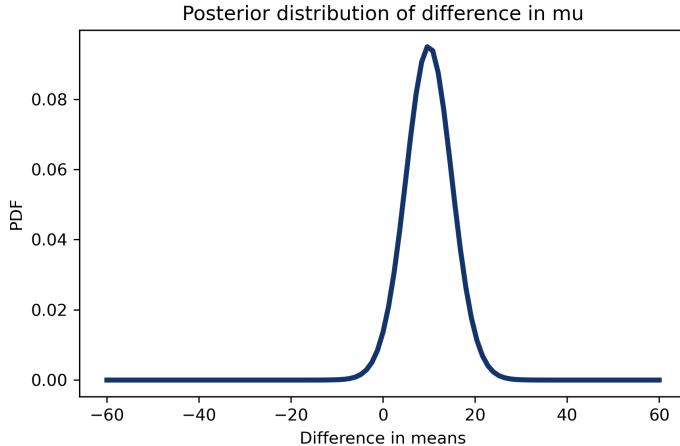
```

`kde_from_pmf` takes as parameters a `Pmf` and the number of places to evaluate the KDE.

It uses `gaussian_kde`, which we saw in “Kernel Density Estimation” on page 115, passing the probabilities from the `Pmf` as weights. This makes the estimated densities higher where the probabilities in the `Pmf` are higher.

Here’s what the kernel density estimate looks like for the `Pmf` of differences between the groups:

```
kde_diff = kde_from_pmf(pmf_diff)
```



The mean of this distribution is almost 10 points on a test where the mean is around 45, so the effect of the treatment seems to be substantial:

```

pmf_diff.mean()

9.954413088940848

```

We can use `credible_interval` to compute a 90% credible interval:

```
pmf_diff.credible_interval(0.9)  
array([ 2.4, 17.4])
```

Based on this interval, we are pretty sure the treatment improves test scores by 2 to 17 points.

Using Summary Statistics

In this example the dataset is not very big, so it doesn't take too long to compute the probability of every score under every hypothesis. But the result is a 3-D array; for larger datasets, it might be too big to compute practically.

Also, with larger datasets the likelihoods get very small, sometimes so small that we can't compute them with floating-point arithmetic. That's because we are computing the probability of a particular dataset; the number of possible datasets is astronomically big, so the probability of any of them is very small.

An alternative is to compute a summary of the dataset and compute the likelihood of the summary. For example, if we compute the mean and standard deviation of the data, we can compute the likelihood of those summary statistics under each hypothesis.

As an example, suppose we know that the actual mean of the population, μ , is 42 and the actual standard deviation, σ , is 17.

```
mu = 42  
sigma = 17
```

Now suppose we draw a sample from this distribution with sample size $n=20$, and compute the mean of the sample, which I'll call m , and the standard deviation of the sample, which I'll call s .

And suppose it turns out that:

```
n = 20  
m = 41  
s = 18
```

The summary statistics, m and s , are not too far from the parameters μ and σ , so it seems like they are not too unlikely.

To compute their likelihood, we will take advantage of three results from mathematical statistics:

- Given μ and σ , the distribution of m is normal with parameters μ and σ/\sqrt{n} ;
- The distribution of s is more complicated, but if we compute the transform $t = ns^2/\sigma^2$, the distribution of t is chi-squared with parameter $n - 1$; and
- According to **Basu's theorem**, m and s are independent.

So let's compute the likelihood of m and s given μ and σ .

First I'll create a `norm` object that represents the distribution of m :

```
dist_m = norm(mu, sigma/np.sqrt(n))
```

This is the “sampling distribution of the mean”. We can use it to compute the likelihood of the observed value of m , which is 41.

```
like1 = dist_m.pdf(m)
like1
0.10137915138497372
```

Now let's compute the likelihood of the observed value of s , which is 18. First, we compute the transformed value t :

```
t = n * s**2 / sigma**2
t
22.422145328719722
```

Then we create a `chi2` object to represent the distribution of t :

```
from scipy.stats import chi2
dist_s = chi2(n-1)
```

Now we can compute the likelihood of t :

```
like2 = dist_s.pdf(t)
like2
0.04736427909437004
```

Finally, because m and s are independent, their joint likelihood is the product of their likelihoods:

```
like = like1 * like2
like
0.004801750420548287
```

Now we can compute the likelihood of the data for any values of μ and σ , which we'll use in the next section to do the update.

Update with Summary Statistics

Now we're ready to do an update. I'll compute summary statistics for the two groups:

```
summary = {}

for name, response in responses.items():
    summary[name] = len(response), response.mean(), response.std()

summary

{'Control': (23, 41.52173913043478, 17.148733229699484),
 'Treated': (21, 51.476190476190474, 11.00735684721381)}
```

The result is a dictionary that maps from group name to a tuple that contains the sample size, n , the sample mean, \bar{m} , and the sample standard deviation s , for each group.

I'll demonstrate the update with the summary statistics from the control group:

```
n, m, s = summary['Control']
```

I'll make a mesh with hypothetical values of μ on the x -axis and values of σ on the y -axis:

```
mus, sigmas = np.meshgrid(prior.columns, prior.index)
mus.shape

(101, 101)
```

Now we can compute the likelihood of seeing the sample mean, \bar{m} , for each pair of parameters:

```
like1 = norm(mus, sigmas/np.sqrt(n)).pdf(m)
like1.shape

(101, 101)
```

And we can compute the likelihood of the sample standard deviation, s , for each pair of parameters:

```
ts = n * s**2 / sigmas**2
like2 = chi2(n-1).pdf(ts)
like2.shape

(101, 101)
```

Finally, we can do the update with both likelihoods:

```
posterior_control2 = prior * like1 * like2
normalize(posterior_control2)
```

To compute the posterior distribution for the treatment group, I'll put the previous steps in a function:

```

def update_norm_summary(prior, data):
    """Update a normal distribution using summary statistics."""
    n, m, s = data
    mu_mesh, sigma_mesh = np.meshgrid(prior.columns, prior.index)

    like1 = norm(mu_mesh, sigma_mesh/np.sqrt(n)).pdf(m)
    like2 = chi2(n-1).pdf(n * s**2 / sigma_mesh**2)

    posterior = prior * like1 * like2
    normalize(posterior)

    return posterior

```

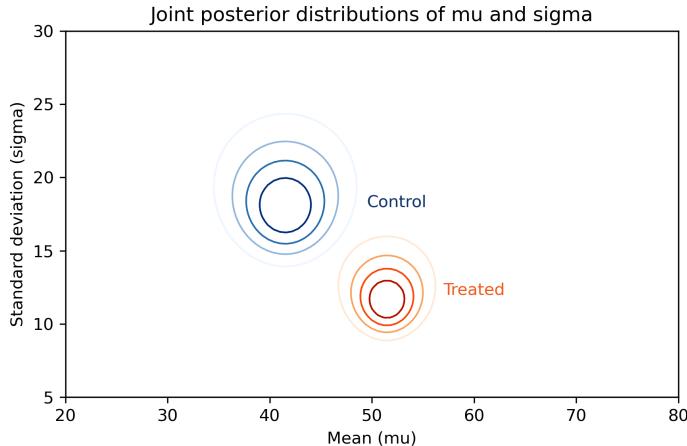
Here's the update for the treatment group:

```

data = summary['Treated']
posterior_treated2 = update_norm_summary(prior, data)

```

And here are the results:



Visually, these posterior joint distributions are similar to the ones we computed using the entire dataset, not just the summary statistics. But they are not exactly the same, as we can see by comparing the marginal distributions.

Comparing Marginals

Again, let's extract the marginal posterior distributions:

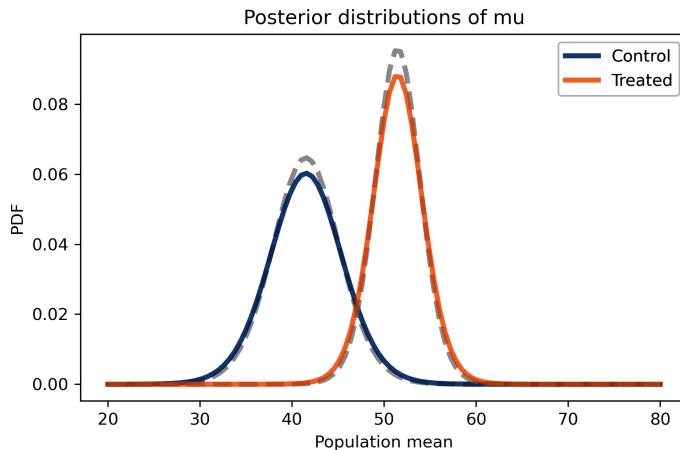
```

from utils import marginal

pmf_mean_control2 = marginal(posterior_control2, 0)
pmf_mean_treated2 = marginal(posterior_treated2, 0)

```

And compare them to results we got using the entire dataset (the dashed lines):



The posterior distributions based on summary statistics are similar to the posteriors we computed using the entire dataset, but in both cases they are shorter and a little wider.

That's because the update with summary statistics is based on the implicit assumption that the distribution of the data is normal. But it's not; as a result, when we replace the dataset with the summary statistics, we lose some information about the true distribution of the data. With less information, we are less certain about the parameters.

Summary

In this chapter we used a joint distribution to represent prior probabilities for the parameters of a normal distribution, μ and σ . And we updated that distribution two ways: first using the entire dataset and the normal PDF; then using summary statistics, the normal PDF, and the chi-square PDF. Using summary statistics is computationally more efficient, but it loses some information in the process.

Normal distributions appear in many domains, so the methods in this chapter are broadly applicable. The exercises at the end of the chapter will give you a chance to apply them.

Exercises

Exercise 13-1.

Looking again at the posterior joint distribution of `mu` and `sigma`, it seems like the standard deviation of the treated group might be lower; if so, that would suggest that the treatment is more effective for students with lower scores.

But before we speculate too much, we should estimate the size of the difference and see whether it might actually be 0.

Extract the marginal posterior distributions of `sigma` for the two groups. What is the probability that the standard deviation is higher in the control group?

Compute the distribution of the difference in `sigma` between the two groups. What is the mean of this difference? What is the 90% credible interval?

Exercise 13-2.

An **effect size** is a statistic intended to quantify the magnitude of a phenomenon. If the phenomenon is a difference in means between two groups, a common way to quantify it is Cohen's effect size, denoted d .

If the parameters for Group 1 are (μ_1, σ_1) , and the parameters for Group 2 are (μ_2, σ_2) , Cohen's effect size is

$$d = \frac{\mu_1 - \mu_2}{(\sigma_1 + \sigma_2)/2}$$

Use the joint posterior distributions for the two groups to compute the posterior distribution for Cohen's effect size.

Exercise 13-3.

This exercise is inspired by [a question that appeared on Reddit](#).

An instructor announces the results of an exam like this: “The average score on this exam was 81. Out of 25 students, 5 got more than 90, and I am happy to report that no one failed (got less than 60).”

Based on this information, what do you think the standard deviation of scores was?

You can assume that the distribution of scores is approximately normal. And let's assume that the sample mean, 81, is actually the population mean, so we only have to estimate `sigma`.

Hint: To compute the probability of a score greater than 90, you can use `norm.sf`, which computes the survival function, also known as the complementary CDF, or `1 - cdf(x)`.

Exercise 13-4.

The **Variability Hypothesis** is the observation that many physical traits are more variable among males than among females, in many species.

It has been a subject of controversy since the early 1800s, which suggests an exercise we can use to practice the methods in this chapter. Let's look at the distribution of heights for men and women in the U.S. and see who is more variable.

I used 2018 data from the CDC's **Behavioral Risk Factor Surveillance System** (BRFSS), which includes self-reported heights from 154,407 men and 254,722 women.

Here's what I found:

- The average height for men is 178 cm; the average height for women is 163 cm. So men are taller on average; no surprise there.
- For men the standard deviation is 8.27 cm; for women it is 7.75 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 0.0465; for women it is 0.0475. The coefficient of variation is higher for women, so this dataset provides evidence against the Variability Hypothesis. But we can use Bayesian methods to make that conclusion more precise.

Use these summary statistics to compute the posterior distribution of `mu` and `sigma` for the distributions of male and female height. Use `Pmf.div_dist` to compute posterior distributions of CV. Based on this dataset and the assumption that the distribution of height is normal, what is the probability that the coefficient of variation is higher for men? What is the most likely ratio of the CVs and what is the 90% credible interval for that ratio?

Survival Analysis

This chapter introduces “survival analysis”, which is a set of statistical methods used to answer questions about the time until an event. In the context of medicine it is literally about survival, but it can be applied to the time until any kind of event, or instead of time it can be about space or other dimensions.

Survival analysis is challenging because the data we have are often incomplete. But as we’ll see, Bayesian methods are particularly good at working with incomplete data.

As examples, we’ll consider two applications that are a little less serious than life and death: the time until light bulbs fail and the time until dogs in a shelter are adopted. To describe these “survival times”, we’ll use the Weibull distribution.

The Weibull Distribution

The [Weibull distribution](#) is often used in survival analysis because it is a good model for the distribution of lifetimes for manufactured products, at least over some parts of the range.

SciPy provides several versions of the Weibull distribution; the one we’ll use is called `weibull_min`. To make the interface consistent with our notation, I’ll wrap it in a function that takes as parameters λ , which mostly affects the location or “central tendency” of the distribution, and k , which affects the shape.

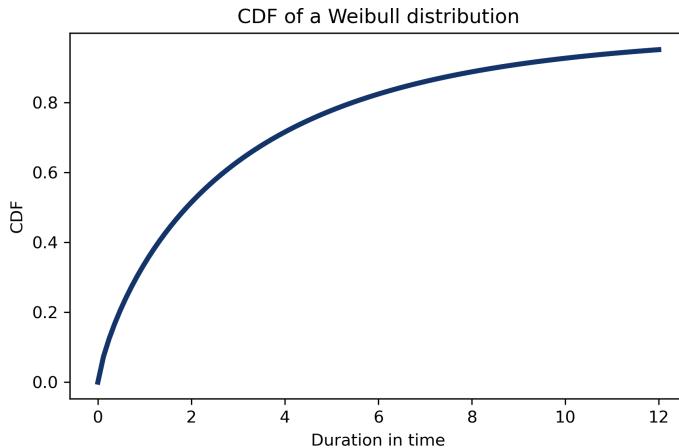
```
from scipy.stats import weibull_min

def weibull_dist(lam, k):
    return weibull_min(k, scale=lam)
```

As an example, here's a Weibull distribution with parameters $\lambda = 3$ and $k = 0.8$:

```
lam = 3
k = 0.8
actual_dist = weibull_dist(lam, k)
```

The result is an object that represents the distribution. Here's what the Weibull CDF looks like with those parameters:



`actual_dist` provides `rvs`, which we can use to generate a random sample from this distribution:

```
data = actual_dist.rvs(10)
data
array([0.80497283, 2.11577082, 0.43308797, 0.10862644, 5.17334866,
       3.25745053, 3.05555883, 2.47401062, 0.05340806, 1.08386395])
```

So, given the parameters of the distribution, we can generate a sample. Now let's see if we can go the other way: given the sample, we'll estimate the parameters.

Here's a uniform prior distribution for λ :

```
from utils import make_uniform

lams = np.linspace(0.1, 10.1, num=101)
prior_lam = make_uniform(lams, name='lambda')
```

And a uniform prior for k :

```
ks = np.linspace(0.1, 5.1, num=101)
prior_k = make_uniform(ks, name='k')
```

I'll use `make_joint` to make a joint prior distribution for the two parameters:

```
from utils import make_joint

prior = make_joint(prior_lam, prior_k)
```

The result is a `DataFrame` that represents the joint prior, with possible values of λ across the columns and values of k down the rows.

Now I'll use `meshgrid` to make a 3-D mesh with λ on the first axis (`axis=0`), k on the second axis (`axis=1`), and the data on the third axis (`axis=2`):

```
lam_mesh, k_mesh, data_mesh = np.meshgrid(
    prior.columns, prior.index, data)
```

Now we can use `weibull_dist` to compute the PDF of the Weibull distribution for each pair of parameters and each data point:

```
densities = weibull_dist(lam_mesh, k_mesh).pdf(data_mesh)
densities.shape

(101, 101, 10)
```

The likelihood of the data is the product of the probability densities along `axis=2`.

```
likelihood = densities.prod(axis=2)
likelihood.sum()

2.0938302958838208e-05
```

Now we can compute the posterior distribution in the usual way:

```
from utils import normalize

posterior = prior * likelihood
normalize(posterior)
```

The following function encapsulates these steps. It takes a joint prior distribution and the data, and returns a joint posterior distribution:

```
def update_weibull(prior, data):
    """Update the prior based on data."""
    lam_mesh, k_mesh, data_mesh = np.meshgrid(
        prior.columns, prior.index, data)

    densities = weibull_dist(lam_mesh, k_mesh).pdf(data_mesh)
    likelihood = densities.prod(axis=2)

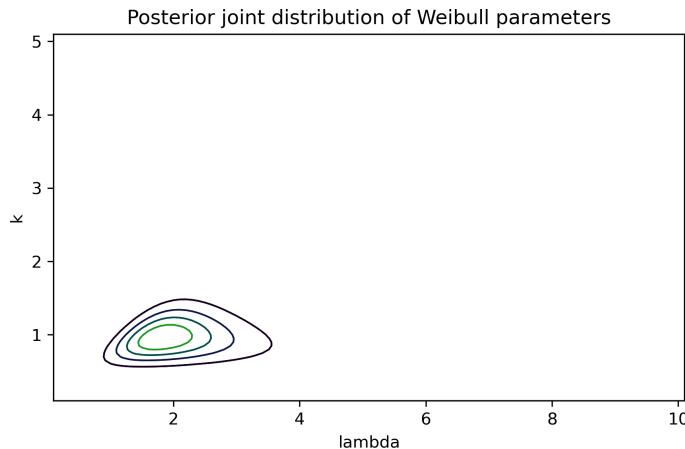
    posterior = prior * likelihood
    normalize(posterior)

    return posterior
```

Here's how we use it:

```
posterior = update_weibull(prior, data)
```

And here's a contour plot of the joint posterior distribution:



It looks like the range of likely values for λ is about 1 to 4, which contains the actual value we used to generate the data, 3. And the range for k is about 0.5 to 1.5, which contains the actual value, 0.8.

Incomplete Data

In the previous example we were given 10 random values from a Weibull distribution, and we used them to estimate the parameters (which we pretended we didn't know).

But in many real-world scenarios, we don't have complete data; in particular, when we observe a system at a point in time, we generally have information about the past, but not the future.

As an example, suppose you work at a dog shelter and you are interested in the time between the arrival of a new dog and when it is adopted. Some dogs might be snapped up immediately; others might have to wait longer. The people who operate the shelter might want to make inferences about the distribution of these residence times.

Suppose you monitor arrivals and departures over 8 weeks, and 10 dogs arrive during that interval. I'll assume that their arrival times are distributed uniformly, so I'll generate random values like this:

```
start = np.random.uniform(0, 8, size=10)
start
array([0.78026881, 6.08999773, 1.97550379, 1.1050535 , 2.65157251,
       0.66399652, 5.37581665, 6.45275039, 7.86193532, 5.08528588])
```

Now let's suppose that the residence times follow the Weibull distribution we used in the previous example. We can generate a sample from that distribution like this:

```

duration = actual_dist.rvs(10)
duration

array([0.80497283, 2.11577082, 0.43308797, 0.10862644, 5.17334866,
       3.25745053, 3.05555883, 2.47401062, 0.05340806, 1.08386395])

```

I'll use these values to construct a `DataFrame` that contains the arrival and departure times for each dog, called `start` and `end`:

```

import pandas as pd

d = dict(start=start, end=start+duration)
obs = pd.DataFrame(d)

```

For display purposes, I'll sort the rows of the `DataFrame` by arrival time:

```

obs = obs.sort_values(by='start', ignore_index=True)
obs

```

	start	end
0	0.663997	3.921447
1	0.780269	1.585242
2	1.105053	1.213680
3	1.975504	2.408592
4	2.651573	7.824921
5	5.085286	6.169150
6	5.375817	8.431375
7	6.089998	8.205769
8	6.452750	8.926761
9	7.861935	7.915343

Notice that several of the lifelines extend past the observation window of 8 weeks. So if we observed this system at the beginning of Week 8, we would have incomplete information. Specifically, we would not know the future adoption times for Dogs 6, 7, and 8.

I'll simulate this incomplete data by identifying the lifelines that extend past the observation window:

```
censored = obs['end'] > 8
```

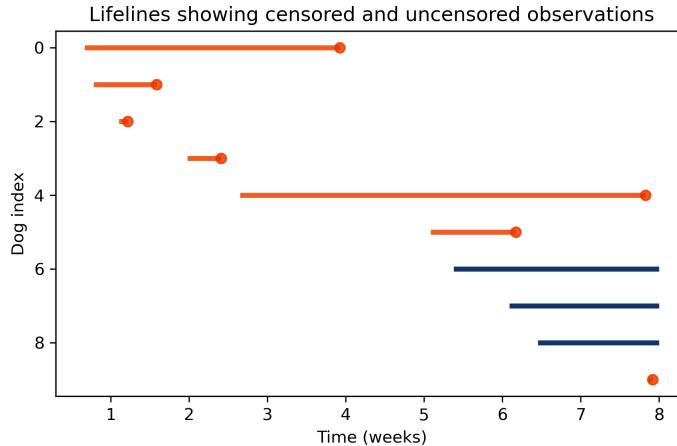
`censored` is a Boolean Series that is `True` for lifelines that extend past Week 8.

Data that is not available is sometimes called “censored” in the sense that it is hidden from us. But in this case it is hidden because we don't know the future, not because someone is censoring it.

For the lifelines that are censored, I'll modify `end` to indicate when they are last observed and `status` to indicate that the observation is incomplete:

```
obs.loc[censored, 'end'] = 8  
obs.loc[censored, 'status'] = 0
```

Now we can plot a “lifeline” for each dog, showing the arrival and departure times on a time line:



And I'll add one more column to the table, which contains the duration of the observed parts of the lifelines:

```
obs['T'] = obs['end'] - obs['start']
```

What we have simulated is the data that would be available at the beginning of Week 8.

Using Incomplete Data

Now, let's see how we can use both kinds of data, complete and incomplete, to infer the parameters of the distribution of residence times.

First I'll split the data into two sets: `data1` contains residence times for dogs whose arrival and departure times are known; `data2` contains incomplete residence times for dogs who were not adopted during the observation interval.

```
data1 = obs.loc[~censored, 'T']  
data2 = obs.loc[censored, 'T']
```

For the complete data, we can use `update_weibull`, which uses the PDF of the Weibull distribution to compute the likelihood of the data.

```
posterior1 = update_weibull(prior, data1)
```

For the incomplete data, we have to think a little harder. At the end of the observation interval, we don't know what the residence time will be, but we can put a lower bound on it; that is, we can say that the residence time will be greater than T .

And that means that we can compute the likelihood of the data using the survival function, which is the probability that a value from the distribution exceeds T .

The following function is identical to `update_weibull` except that it uses `sf`, which computes the survival function, rather than `pdf`.

```
def update_weibull_incomplete(prior, data):
    """Update the prior using incomplete data."""
    lam_mesh, k_mesh, data_mesh = np.meshgrid(
        prior.columns, prior.index, data)

    # evaluate the survival function
    probs = weibull_dist(lam_mesh, k_mesh).sf(data_mesh)
    likelihood = probs.prod(axis=2)

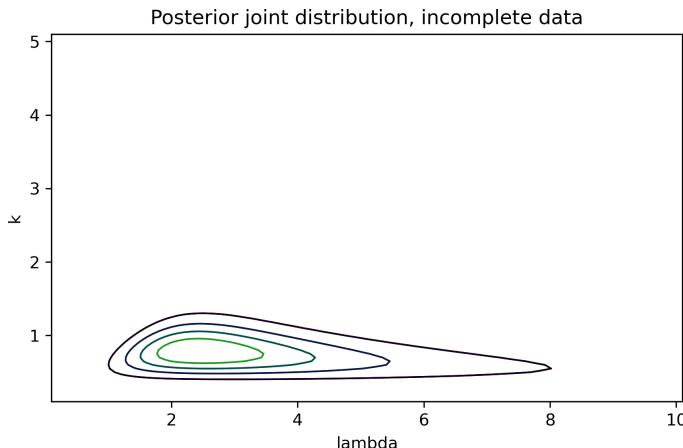
    posterior = prior * likelihood
    normalize(posterior)

    return posterior
```

Here's the update with the incomplete data:

```
posterior2 = update_weibull_incomplete(posterior1, data2)
```

And here's what the joint posterior distribution looks like after both updates:



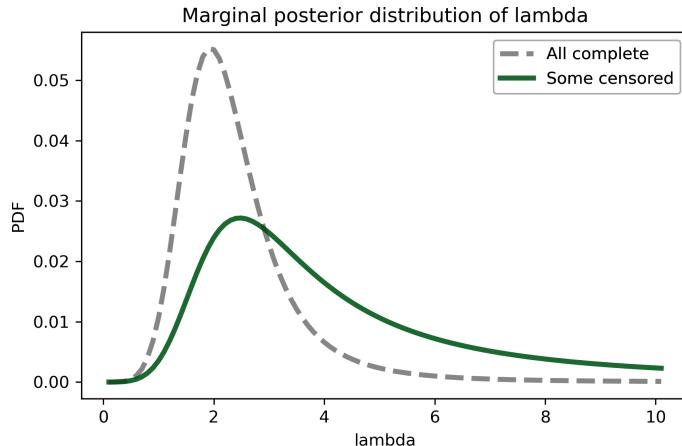
Compared to the previous contour plot, it looks like the range of likely values for λ is substantially wider. We can see that more clearly by looking at the marginal distributions.

```

posterior_lam2 = marginal(posterior2, 0)
posterior_k2 = marginal(posterior2, 1)

```

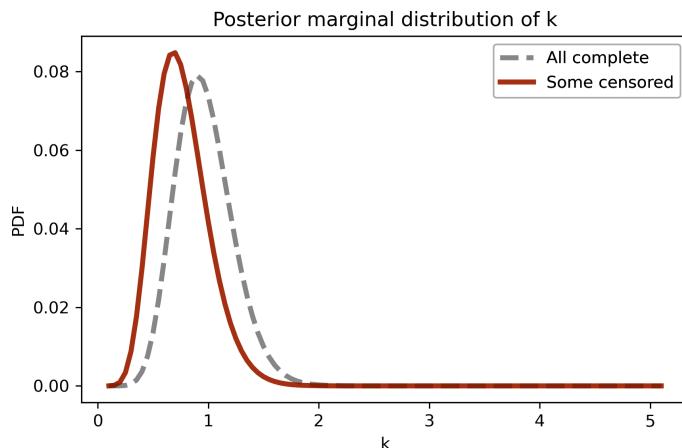
Here's the posterior marginal distribution for λ compared to the distribution we got using all complete data:



The distribution with some incomplete data is substantially wider.

As an aside, notice that the posterior distribution does not come all the way to 0 on the right side. That suggests that the range of the prior distribution is not wide enough to cover the most likely values for this parameter. If I were concerned about making this distribution more accurate, I would go back and run the update again with a wider prior.

Here's the posterior marginal distribution for k :



In this example, the marginal distribution is shifted to the left when we have incomplete data, but it is not substantially wider.

In summary, we have seen how to combine complete and incomplete data to estimate the parameters of a Weibull distribution, which is useful in many real-world scenarios where some of the data are censored.

In general, the posterior distributions are wider when we have incomplete data, because less information leads to more uncertainty.

This example is based on data I generated; in the next section we'll do a similar analysis with real data.

Light Bulbs

In 2007 [researchers ran an experiment](#) to characterize the distribution of lifetimes for light bulbs. Here is their description of the experiment:

An assembly of 50 new Philips (India) lamps with the rating 40 W, 220 V (AC) was taken and installed in the horizontal orientation and uniformly distributed over a lab area 11 m x 7 m.

The assembly was monitored at regular intervals of 12 h to look for failures. The instants of recorded failures were [recorded] and a total of 32 data points were obtained such that even the last bulb failed.

We can load the data into a `DataFrame` like this:

```
df = pd.read_csv('Lamps.csv', index_col=0)
df.head()
```

i	h	f	K
0	0	0	50
1	840	2	48
2	852	1	47
3	936	1	46
4	960	1	45

Column `h` contains the times when bulbs failed in hours; Column `f` contains the number of bulbs that failed at each time. We can represent these values and frequencies using a `Pmf`, like this:

```
from empiricaldist import Pmf

pmf_bulb = Pmf(df['f'].to_numpy(), df['h'])
pmf_bulb.normalize()

50
```

Because of the design of this experiment, we can consider the data to be a representative sample from the distribution of lifetimes, at least for light bulbs that are lit continuously.

Assuming that these data are well modeled by a Weibull distribution, let's estimate the parameters that fit the data. Again, I'll start with uniform priors for λ and k :

```
lams = np.linspace(1000, 2000, num=51)
prior_lam = make_uniform(lams, name='lambda')

ks = np.linspace(1, 10, num=51)
prior_k = make_uniform(ks, name='k')
```

For this example, there are 51 values in the prior distribution, rather than the usual 101. That's because we are going to use the posterior distributions to do some computationally intensive calculations. They will run faster with fewer values, but the results will be less precise.

As usual, we can use `make_joint` to make the prior joint distribution:

```
prior_bulb = make_joint(prior_lam, prior_k)
```

Although we have data for 50 light bulbs, there are only 32 unique lifetimes in the dataset. For the update, it is convenient to express the data in the form of 50 lifetimes, with each lifetime repeated the given number of times. We can use `np.repeat` to transform the data:

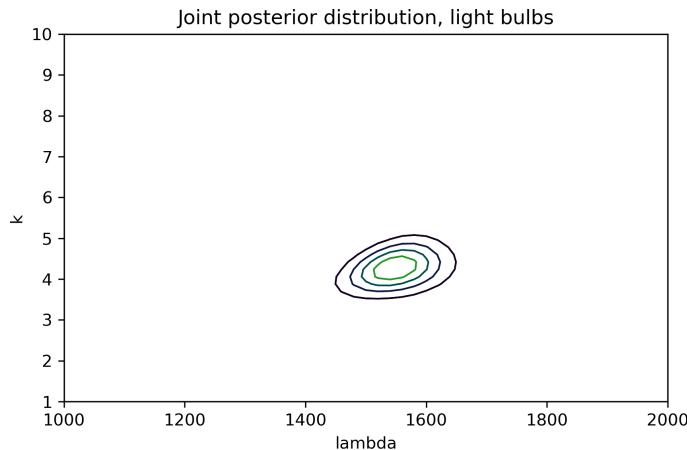
```
data_bulb = np.repeat(df['h'], df['f'])
len(data_bulb)

50
```

Now we can use `update_weibull` to do the update:

```
posterior_bulb = update_weibull(prior_bulb, data_bulb)
```

Here's what the posterior joint distribution looks like:



To summarize this joint posterior distribution, we'll compute the posterior mean lifetime.

Posterior Means

To compute the posterior mean of a joint distribution, we'll make a mesh that contains the values of λ and k :

```
lam_mesh, k_mesh = np.meshgrid(  
    prior_bulb.columns, prior_bulb.index)
```

Now for each pair of parameters we'll use `weibull_dist` to compute the mean:

```
means = weibull_dist(lam_mesh, k_mesh).mean()  
means.shape  
(51, 51)
```

The result is an array with the same dimensions as the joint distribution.

Now we need to weight each mean with the corresponding probability from the joint posterior:

```
prod = means * posterior_bulb
```

Finally we compute the sum of the weighted means:

```
prod.to_numpy().sum()  
1412.7242774305005
```

Based on the posterior distribution, we think the mean lifetime is about 1,413 hours.

The following function encapsulates these steps:

```
def joint_weibull_mean(joint):
    """Compute the mean of a joint distribution of Weibulls."""
    lam_mesh, k_mesh = np.meshgrid(
        joint.columns, joint.index)
    means = weibull_dist(lam_mesh, k_mesh).mean()
    prod = means * joint
    return prod.to_numpy().sum()
```

Posterior Predictive Distribution

Suppose you install 100 light bulbs of the kind in the previous section, and you come back to check on them after 1,000 hours. Based on the posterior distribution we just computed, what is the distribution of the number of bulbs you find dead?

If we knew the parameters of the Weibull distribution for sure, the answer would be a binomial distribution.

For example, if we know that $\lambda = 1550$ and $k = 4.25$, we can use `weibull_dist` to compute the probability that a bulb dies before you return:

```
lam = 1550
k = 4.25
t = 1000

prob_dead = weibull_dist(lam, k).cdf(t)
prob_dead

0.14381685899960547
```

If there are 100 bulbs and each has this probability of dying, the number of dead bulbs follows a binomial distribution.

```
from utils import make_binomial

n = 100
p = prob_dead
dist_num_dead = make_binomial(n, p)
```

But that's based on the assumption that we know λ and k , and we don't. Instead, we have a posterior distribution that contains possible values of these parameters and their probabilities.

So the posterior predictive distribution is not a single binomial; instead it is a mixture of binomials, weighted with the posterior probabilities.

We can use `make_mixture` to compute the posterior predictive distribution.

It doesn't work with joint distributions, but we can convert the `DataFrame` that represents a joint distribution to a `Series`, like this:

```

posterior_series = posterior_bulb.stack()
posterior_series.head()

k      lambda
1.0    1000.0    8.146763e-25
       1020.0    1.210486e-24
       1040.0    1.738327e-24
       1060.0    2.418201e-24
       1080.0    3.265549e-24
dtype: float64

```

The result is a `Series` with a `MultiIndex` that contains two “levels”: the first level contains the values of `k`; the second contains the values of `lam`.

With the posterior in this form, we can iterate through the possible parameters and compute a predictive distribution for each pair:

```

pmf_seq = []
for (k, lam) in posterior_series.index:
    prob_dead = weibull_dist(lam, k).cdf(t)
    pmf = make_binomial(n, prob_dead)
    pmf_seq.append(pmf)

```

Now we can use `make_mixture`, passing as parameters the posterior probabilities in `posterior_series` and the sequence of binomial distributions in `pmf_seq`:

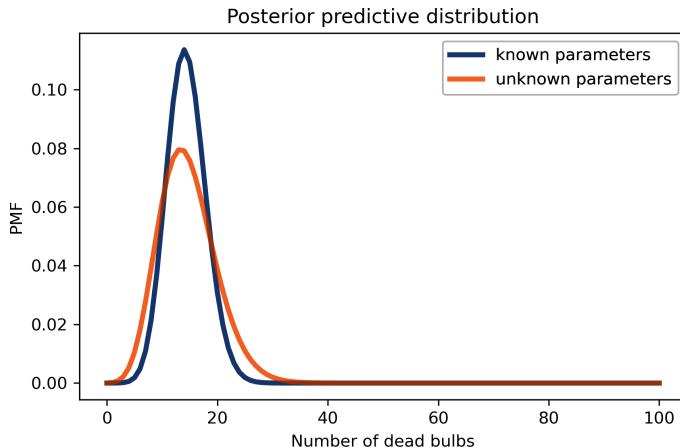
```

from utils import make_mixture

post_pred = make_mixture(posterior_series, pmf_seq)

```

Here’s what the posterior predictive distribution looks like, compared to the binomial distribution we computed with known parameters:



The posterior predictive distribution is wider because it represents our uncertainty about the parameters as well as our uncertainty about the number of dead bulbs.

Summary

This chapter introduces survival analysis, which is used to answer questions about the time until an event, and the Weibull distribution, which is a good model for “lifetimes” (broadly interpreted) in a number of domains.

We used joint distributions to represent prior probabilities for the parameters of the Weibull distribution, and we updated them three ways: knowing the exact duration of a lifetime, knowing a lower bound, and knowing that a lifetime fell in a given interval.

These examples demonstrate a feature of Bayesian methods: they can be adapted to handle incomplete, or “censored”, data with only small changes. As an exercise, you’ll have a chance to work with one more type of censored data, when we are given an upper bound on a lifetime.

The methods in this chapter work with any distribution with two parameters. In the exercises, you’ll have a chance to estimate the parameters of a two-parameter gamma distribution, which is used to describe a variety of natural phenomena.

And in the next chapter we’ll move on to models with three parameters!

Exercises

Exercise 14-1.

Using data about the lifetimes of light bulbs, we computed the posterior distribution from the parameters of a Weibull distribution, λ and k , and the posterior predictive distribution for the number of dead bulbs, out of 100, after 1,000 hours.

Now suppose you do the experiment: You install 100 light bulbs, come back after 1,000 hours, and find 20 dead light bulbs. Update the posterior distribution based on this data. How much does it change the posterior mean?

Exercise 14-2.

In this exercise, we’ll use one month of data to estimate the parameters of a distribution that describes daily rainfall in Seattle. Then we’ll compute the posterior predictive distribution for daily rainfall and use it to estimate the probability of a rare event, like more than 1.5 inches of rain in a day.

According to hydrologists, the distribution of total daily rainfall (for days with rain) is well modeled by a two-parameter gamma distribution.

When we worked with the one-parameter gamma distribution in “[The Gamma Distribution](#)” on page 101, we used the Greek letter α for the parameter.

For the two-parameter gamma distribution, we will use k for the “shape parameter”, which determines the shape of the distribution, and the Greek letter θ or theta for the “scale parameter”.

I suggest you proceed in the following steps:

1. Construct a prior distribution for the parameters of the gamma distribution.
Note that k and θ must be greater than 0.
2. Use the observed rainfalls to update the distribution of parameters.
3. Compute the posterior predictive distribution of rainfall, and use it to estimate the probability of getting more than 1.5 inches of rain in one day.

CHAPTER 15

Mark and Recapture

This chapter introduces “mark and recapture” experiments, in which we sample individuals from a population, mark them somehow, and then take a second sample from the same population. Seeing how many individuals in the second sample are marked, we can estimate the size of the population.

Experiments like this were originally used in ecology, but turn out to be useful in many other fields. Examples in this chapter include software engineering and epidemiology.

Also, in this chapter we’ll work with models that have three parameters, so we’ll extend the joint distributions we’ve been using to three dimensions.

But first, grizzly bears.

The Grizzly Bear Problem

In 1996 and 1997 researchers deployed bear traps in locations in British Columbia and Alberta, Canada, in an effort to estimate the population of grizzly bears. They describe the experiment in [this article](#).

The “trap” consists of a lure and several strands of barbed wire intended to capture samples of hair from bears that visit the lure. Using the hair samples, the researchers use DNA analysis to identify individual bears.

During the first session, the researchers deployed traps at 76 sites. Returning 10 days later, they obtained 1,043 hair samples and identified 23 different bears. During a second 10-day session they obtained 1,191 samples from 19 different bears, where 4 of the 19 were from bears they had identified in the first batch.

To estimate the population of bears from this data, we need a model for the probability that each bear will be observed during each session. As a starting place, we'll make the simplest assumption, that every bear in the population has the same (unknown) probability of being sampled during each session.

With these assumptions we can compute the probability of the data for a range of possible populations.

As an example, let's suppose that the actual population of bears is 100.

After the first session, 23 of the 100 bears have been identified. During the second session, if we choose 19 bears at random, what is the probability that 4 of them were previously identified?

I'll define:

- N : actual population size, 100.
- K : number of bears identified in the first session, 23.
- n : number of bears observed in the second session, 19 in the example.
- k : number of bears in the second session that were previously identified, 4.

For given values of N , K , and n , the probability of finding k previously-identified bears is given by the **hypergeometric distribution**:

$$\binom{K}{k} \binom{N-K}{n-k} / \binom{N}{n}$$

where the **binomial coefficient**, $\binom{K}{k}$, is the number of subsets of size k we can choose from a population of size K .

To understand why, consider:

- The denominator, $\binom{N}{n}$, is the number of subsets of n we could choose from a population of N bears.
- The numerator is the number of subsets that contain k bears from the previously identified K and $n - k$ from the previously unseen $N - K$.

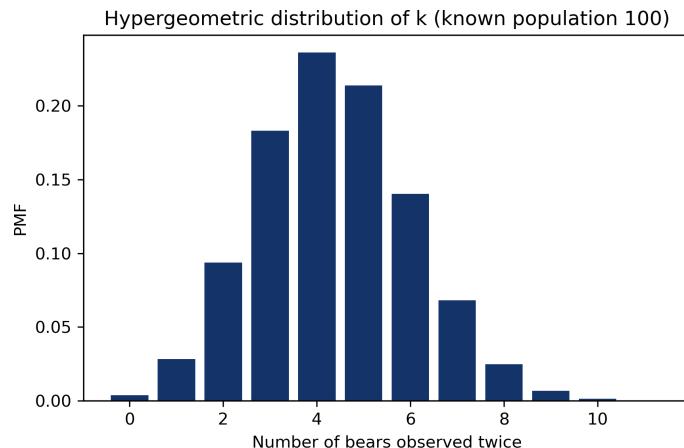
SciPy provides `hypergeom`, which we can use to compute this probability for a range of values of k :

```
import numpy as np
from scipy.stats import hypergeom

N = 100
K = 23
n = 19
```

```
ks = np.arange(12)
ps = hypergeom(N, K, n).pmf(ks)
```

The result is the distribution of k with given parameters N , K , and n . Here's what it looks like:



The most likely value of k is 4, which is the value actually observed in the experiment. That suggests that $N = 100$ is a reasonable estimate of the population, given this data.

We've computed the distribution of k given N , K , and n . Now let's go the other way: given K , n , and k , how can we estimate the total population, N ?

The Update

As a starting place, let's suppose that, prior to this study, an expert estimates that the local bear population is between 50 and 500, and equally likely to be any value in that range.

I'll use `make_uniform` to make a uniform distribution of integers in this range:

```
import numpy as np
from utils import make_uniform

qs = np.arange(50, 501)
prior_N = make_uniform(qs, name='N')
prior_N.shape

(451,)
```

So that's our prior.

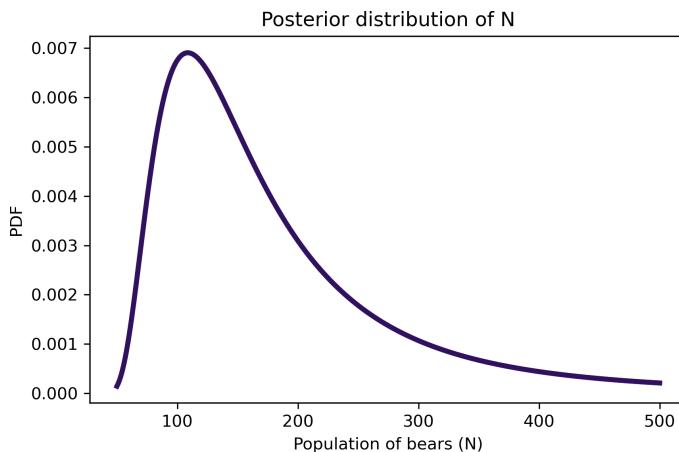
To compute the likelihood of the data, we can use `hypergeom` with constants K and n, and a range of values of N:

```
Ns = prior_N.qs  
K = 23  
n = 19  
k = 4  
  
likelihood = hypergeom(Ns, K, n).pmf(k)
```

We can compute the posterior in the usual way:

```
posterior_N = prior_N * likelihood  
posterior_N.normalize()  
  
0.07755224277106727
```

And here's what it looks like:



The most likely value is 109:

```
posterior_N.max_prob()  
109
```

But the distribution is skewed to the right, so the posterior mean is substantially higher:

```
posterior_N.mean()  
173.79880627085637
```

And the credible interval is quite wide:

```
posterior_N.credible_interval(0.9)  
array([ 77., 363.])
```

This solution is relatively simple, but it turns out we can do a little better if we model the unknown probability of observing a bear explicitly.

Two-Parameter Model

Next we'll try a model with two parameters: the number of bears, N , and the probability of observing a bear, p .

We'll assume that the probability is the same in both rounds, which is probably reasonable in this case because it is the same kind of trap in the same place.

We'll also assume that the probabilities are independent; that is, the probability a bear is observed in the second round does not depend on whether it was observed in the first round. This assumption might be less reasonable, but for now it is a necessary simplification.

Here are the counts again:

```
K = 23  
n = 19  
k = 4
```

For this model, I'll express the data in a notation that will make it easier to generalize to more than two rounds:

- k_{10} is the number of bears observed in the first round but not the second,
- k_{01} is the number of bears observed in the second round but not the first, and
- k_{11} is the number of bears observed in both rounds.

Here are their values:

```
k10 = 23 - 4  
k01 = 19 - 4  
k11 = 4
```

Suppose we know the actual values of N and p . We can use them to compute the likelihood of this data.

For example, suppose we know that $N=100$ and $p=0.2$. We can use N to compute k_{00} , which is the number of unobserved bears:

```
N = 100  
  
observed = k01 + k10 + k11  
k00 = N - observed  
k00
```

For the update, it will be convenient to store the data as a list that represents the number of bears in each category:

```
x = [k00, k01, k10, k11]
x
[62, 15, 19, 4]
```

Now, if we know $p=0.2$, we can compute the probability a bear falls in each category. For example, the probability of being observed in both rounds is $p*p$, and the probability of being unobserved in both rounds is $q*q$ (where $q=1-p$).

```
p = 0.2
q = 1-p
y = [q*q, q*p, p*q, p*p]
y
[0.6400000000000001,
 0.1600000000000003,
 0.1600000000000003,
 0.0400000000000001]
```

Now the probability of the data is given by the **multinomial distribution**:

$$\frac{N!}{\prod x_i!} \prod y_i^{x_i}$$

where N is actual population, x is a sequence with the counts in each category, and y is a sequence of probabilities for each category.

SciPy provides `multinomial`, which provides `pmf`, which computes this probability. Here is the probability of the data for these values of N and p :

```
from scipy.stats import multinomial

likelihood = multinomial.pmf(x, N, y)
likelihood
0.0016664011988507257
```

That's the likelihood if we know N and p , but of course we don't. So we'll choose prior distributions for N and p , and use the likelihoods to update it.

The Prior

We'll use `prior_N` again for the prior distribution of N , and a uniform prior for the probability of observing a bear, p :

```
qs = np.linspace(0, 0.99, num=100)
prior_p = make_uniform(qs, name='p')
```

We can make a joint distribution in the usual way:

```
from utils import make_joint

joint_prior = make_joint(prior_p, prior_N)
joint_prior.shape

(451, 100)
```

The result is a pandas `DataFrame` with values of `N` down the rows and values of `p` across the columns. However, for this problem it will be convenient to represent the prior distribution as a 1-D `Series` rather than a 2-D `DataFrame`. We can convert from one format to the other using `stack`:

```
from empiricaldist import Pmf

joint_pmf = Pmf(joint_prior.stack())
joint_pmf.head(3)
```

probs		
N	p	
50	0.00	0.000022
	0.01	0.000022
	0.02	0.000022

The result is a `Pmf` whose index is a `MultiIndex`. A `MultiIndex` can have more than one column; in this example, the first column contains values of `N` and the second column contains values of `p`.

The `Pmf` has one row (and one prior probability) for each possible pair of parameters `N` and `p`. So the total number of rows is the product of the lengths of `prior_N` and `prior_p`.

Now we have to compute the likelihood of the data for each pair of parameters.

The Update

To allocate space for the likelihoods, it is convenient to make a copy of `joint_pmf`:

```
likelihood = joint_pmf.copy()
```

As we loop through the pairs of parameters, we compute the likelihood of the data as in the previous section, and then store the result as an element of `likelihood`:

```
observed = k01 + k10 + k11

for N, p in joint_pmf.index:
    k00 = N - observed
    x = [k00, k01, k10, k11]
```

```

q = 1-p
y = [q*q, q*p, p*q, p*p]
likelihood[N, p] = multinomial.pmf(x, N, y)

```

Now we can compute the posterior in the usual way:

```

posterior_pmf = joint_pmf * likelihood
posterior_pmf.normalize()

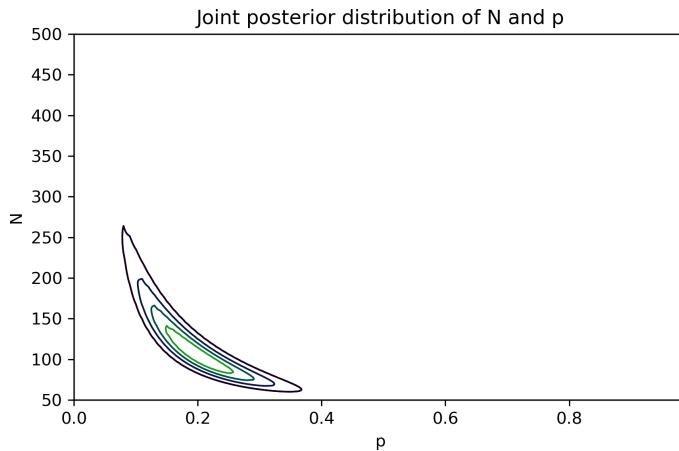
```

We'll use `plot_contour` again to visualize the joint posterior distribution. But remember that the posterior distribution we just computed is represented as a `Pmf`, which is a `Series`, and `plot_contour` expects a `DataFrame`.

Since we used `stack` to convert from a `DataFrame` to a `Series`, we can use `unstack` to go the other way:

```
joint_posterior = posterior_pmf.unstack()
```

And here's what the result looks like:



The most likely values of N are near 100, as in the previous model. The most likely values of p are near 0.2.

The shape of this contour indicates that these parameters are correlated. If p is near the low end of the range, the most likely values of N are higher; if p is near the high end of the range, N is lower.

Now that we have a posterior `DataFrame`, we can extract the marginal distributions in the usual way:

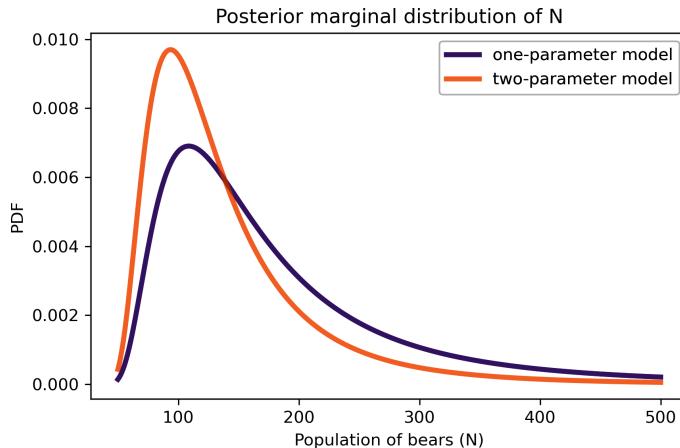
```

from utils import marginal

posterior2_p = marginal(joint_posterior, 0)
posterior2_N = marginal(joint_posterior, 1)

```

Here's the posterior distribution for N based on the two-parameter model, along with the posterior we got using the one-parameter (hypergeometric) model:



With the two-parameter model, the mean is a little lower and the 90% credible interval is a little narrower.

The Lincoln Index Problem

In an excellent blog post, John D. Cook wrote about the Lincoln index, which is a way to estimate the number of errors in a document (or program) by comparing results from two independent testers. Here's his presentation of the problem:

Suppose you have a tester who finds 20 bugs in your program. You want to estimate how many bugs are really in the program. You know there are at least 20 bugs, and if you have supreme confidence in your tester, you may suppose there are around 20 bugs. But maybe your tester isn't very good. Maybe there are hundreds of bugs. How can you have any idea how many bugs there are? There's no way to know with one tester. But if you have two testers, you can get a good idea, even if you don't know how skilled the testers are.

Suppose the first tester finds 20 bugs, the second finds 15, and they find 3 in common; how can we estimate the number of bugs?

This problem is similar to the Grizzly Bear Problem, so I'll represent the data in the same way:

```
k10 = 20 - 3  
k01 = 15 - 3  
k11 = 3
```

But in this case it is probably not reasonable to assume that the testers have the same probability of finding a bug. So I'll define two parameters, p_0 for the probability that

the first tester finds a bug, and p_1 for the probability that the second tester finds a bug.

I will continue to assume that the probabilities are independent, which is like assuming that all bugs are equally easy to find. That might not be a good assumption, but let's stick with it for now.

As an example, suppose we know that the probabilities are 0.2 and 0.15.

```
p0, p1 = 0.2, 0.15
```

We can compute the array of probabilities, y , like this:

```
def compute_probs(p0, p1):
    """Computes the probability for each of 4 categories."""
    q0 = 1-p0
    q1 = 1-p1
    return [q0*q1, q0*p1, p0*q1, p0*p1]

y = compute_probs(p0, p1)
y

[0.68, 0.12, 0.17, 0.03]
```

With these probabilities, there is a 68% chance that neither tester finds the bug and a 3% chance that both do.

Pretending that these probabilities are known, we can compute the posterior distribution for N . Here's a prior distribution that's uniform from 32 to 350 bugs:

```
qs = np.arange(32, 350, step=5)
prior_N = make_uniform(qs, name='N')
prior_N.head(3)
```

	probs
N	
32	0.015625
37	0.015625
42	0.015625

I'll put the data in an array, with 0 as a place-keeper for the unknown value k_{00} :

```
data = np.array([0, k01, k10, k11])
```

And here are the likelihoods for each value of N , with ps as a constant:

```
likelihood = prior_N.copy()
observed = data.sum()
x = data.copy()

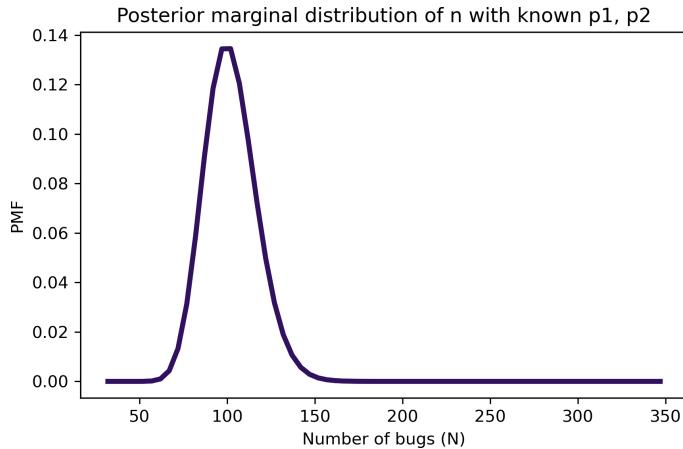
for N in prior_N.qs:
```

```
x[0] = N - observed  
likelihood[N] = multinomial.pmf(x, N, y)
```

We can compute the posterior in the usual way:

```
posterior_N = prior_N * likelihood  
posterior_N.normalize()  
0.0003425201572557094
```

And here's what it looks like:



With the assumption that p_0 and p_1 are known to be 0.2 and 0.15, the posterior mean is 102 with 90% credible interval (77, 127). But this result is based on the assumption that we know the probabilities, and we don't.

Three-Parameter Model

What we need is a model with three parameters: N , p_0 , and p_1 . We'll use `prior_N` again for the prior distribution of N , and here are the priors for p_0 and p_1 :

```
qs = np.linspace(0, 1, num=51)  
prior_p0 = make_uniform(qs, name='p0')  
prior_p1 = make_uniform(qs, name='p1')
```

Now we have to assemble them into a joint prior with three dimensions. I'll start by putting the first two into a `DataFrame`:

```
joint2 = make_joint(prior_p0, prior_N)  
joint2.shape  
(64, 51)
```

Now I'll stack them, as in the previous example, and put the result in a `Pmf`:

```
joint2_pmf = Pmf(joint2.stack())
joint2_pmf.head(3)
```

probs		
N	p0	
32	0.00	0.000306
	0.02	0.000306
	0.04	0.000306

We can use `make_joint` again to add in the third parameter:

```
joint3 = make_joint(prior_p1, joint2_pmf)
joint3.shape
(3264, 51)
```

The result is a `DataFrame` with values of `N` and `p0` in a `MultiIndex` that goes down the rows and values of `p1` in an index that goes across the columns.

Now I'll apply `stack` again:

```
joint3_pmf = Pmf(joint3.stack())
joint3_pmf.head(3)
```

probs		
N	p0	p1
32	0.0	0.00 0.000006
		0.02 0.000006
		0.04 0.000006

The result is a `Pmf` with a three-column `MultiIndex` containing all possible triplets of parameters.

The number of rows is the product of the number of values in all three priors, which is almost 170,000:

```
joint3_pmf.shape
(166464,)
```

That's still small enough to be practical, but it will take longer to compute the likelihoods than in the previous examples.

Here's the loop that computes the likelihoods; it's similar to the one in the previous section:

```
likelihood = joint3_pmf.copy()
observed = data.sum()
x = data.copy()

for N, p0, p1 in joint3_pmf.index:
    x[0] = N - observed
    y = compute_probs(p0, p1)
    likelihood[N, p0, p1] = multinomial.pmf(x, N, y)
```

We can compute the posterior in the usual way:

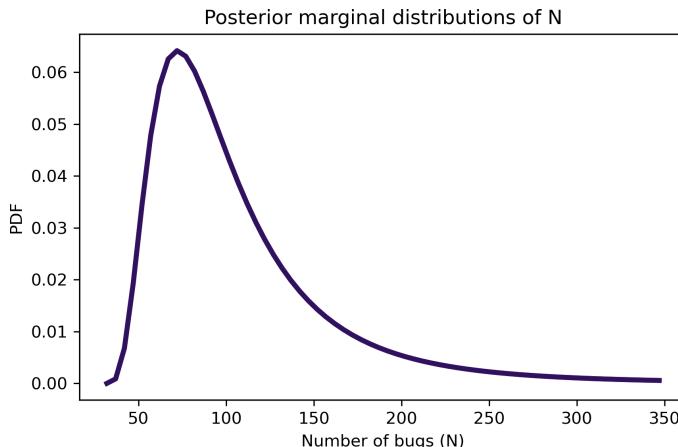
```
posterior_pmf = joint3_pmf * likelihood
posterior_pmf.normalize()

8.941088283758206e-06
```

Now, to extract the marginal distributions, we could unstack the joint posterior as we did in the previous section. But `Pmf` provides a version of `marginal` that works with a `Pmf` rather than a `DataFrame`. Here's how we use it to get the posterior distribution for `N`:

```
posterior_N = posterior_pmf.marginal(0)
```

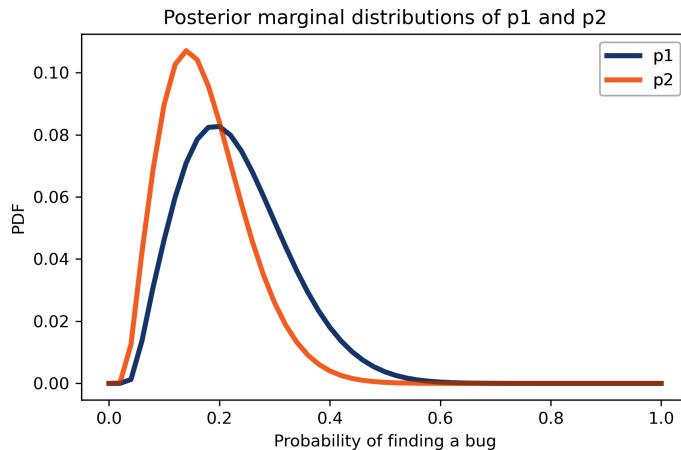
And here's what it looks like:



105.7656173219623

The posterior mean is 105 bugs, which suggests that there are still many bugs the testers have not found.

Here are the posteriors for p_0 and p_1 :



Comparing the posterior distributions, the tester who found more bugs probably has a higher probability of finding bugs. The posterior means are about 23% and 18%. But the distributions overlap, so we should not be too sure.

This is the first example we've seen with three parameters. As the number of parameters increases, the number of combinations increases quickly. The method we've been using so far, enumerating all possible combinations, becomes impractical if the number of parameters is more than 3 or 4.

However, there are other methods that can handle models with many more parameters, as we'll see in [Chapter 19](#).

Summary

The problems in this chapter are examples of [mark and recapture](#) experiments, which are used in ecology to estimate animal populations. They also have applications in engineering, as in the Lincoln Index Problem. And in the exercises you'll see that they are used in epidemiology, too.

This chapter introduces two new probability distributions:

- The hypergeometric distribution is a variation of the binomial distribution in which samples are drawn from the population without replacement.
- The multinomial distribution is a generalization of the binomial distribution where there are more than two possible outcomes.

Also in this chapter, we saw the first example of a model with three parameters. We'll see more in subsequent chapters.

Exercises

Exercise 15-1.

In an excellent paper, Anne Chao explains how mark and recapture experiments are used in epidemiology to estimate the prevalence of a disease in a human population based on multiple incomplete lists of cases.

One of the examples in that paper is a study “to estimate the number of people who were infected by hepatitis in an outbreak that occurred in and around a college in northern Taiwan from April to July 1995.”

Three lists of cases were available:

1. 135 cases identified using a serum test.
2. 122 cases reported by local hospitals.
3. 126 cases reported on questionnaires collected by epidemiologists.

In this exercise, we’ll use only the first two lists; in the next exercise we’ll bring in the third list.

Make a joint prior and update it using this data, then compute the posterior mean of N and a 90% credible interval.

Exercise 15-2.

Now let’s do the version of the problem with all three lists. Here’s the data from Chou’s paper:

Hepatitis A virus list			
P	Q	E	Data
1	1	1	k111 =28
1	1	0	k110 =21
1	0	1	k101 =17
1	0	0	k100 =69
0	1	1	k011 =18
0	1	0	k010 =55
0	0	1	k001 =63
0	0	0	k000 =??

Write a loop that computes the likelihood of the data for each pair of parameters, then update the prior and compute the posterior mean of N . How does it compare to the results using only the first two lists?

Logistic Regression

This chapter introduces two related topics: log odds and logistic regression.

In “[Bayes’s Rule](#)” on page 70, we rewrote Bayes’s theorem in terms of odds and derived Bayes’s rule, which can be a convenient way to do a Bayesian update on paper or in your head. In this chapter, we’ll look at Bayes’s rule on a logarithmic scale, which provides insight into how we accumulate evidence through successive updates.

That leads directly to logistic regression, which is based on a linear model of the relationship between evidence and the log odds of a hypothesis. As an example, we’ll use data from the Space Shuttle to explore the relationship between temperature and the probability of damage to the O-rings.

As an exercise, you’ll have a chance to model the relationship between a child’s age when they start school and their probability of being diagnosed with attention deficit hyperactivity disorder (ADHD).

Log Odds

When I was in grad school, I signed up for a class on the Theory of Computation. On the first day of class, I was the first to arrive. A few minutes later, another student arrived.

At the time, about 83% of the students in the computer science program [were male](#), so I was mildly surprised to note that the other student was female.

When another female student arrived a few minutes later, I started to think I was in the wrong room. When a third female student arrived, I was confident I was in the wrong room. And as it turned out, I was.

I'll use this anecdote to demonstrate Bayes's rule on a logarithmic scale and show how it relates to logistic regression.

Using H to represent the hypothesis that I was in the right room, and F to represent the observation that the first other student was female, we can write Bayes's rule like this:

$$O(H|F) = O(H) \frac{P(F|H)}{P(F|notH)}$$

Before I saw the other students, I was confident I was in the right room, so I might assign prior odds of 10:1 in favor:

$$O(H) = 10$$

If I was in the right room, the likelihood of the first female student was about 17%. If I was not in the right room, the likelihood of the first female student was more like 50%:

$$\frac{P(F|H)}{P(F|notH)} = 17/50$$

So the likelihood ratio is close to 1/3. Applying Bayes's rule, the posterior odds were

$$O(H|F) = 10/3$$

After two students, the posterior odds were

$$O(H|FF) = 10/9$$

And after three students:

$$O(H|FFF) = 10/27$$

At that point, I was right to suspect I was in the wrong room.

The following table shows the odds after each update, the corresponding probabilities, and the change in probability after each step, expressed in percentage points.

	odds	prob	prob diff
prior	10.000000	0.909091	--
1 student	3.333333	0.769231	-13.986014
2 students	1.111111	0.526316	-24.291498
3 students	0.370370	0.270270	-25.604552

Each update uses the same likelihood, but the changes in probability are not the same. The first update decreases the probability by about 14 percentage points, the second by 24, and the third by 26. That's normal for this kind of update, and in fact it's necessary; if the changes were the same size, we would quickly get into negative probabilities.

The odds follow a more obvious pattern. Because each update multiplies the odds by the same likelihood ratio, the odds form a geometric sequence. And that brings us to consider another way to represent uncertainty: **log odds**, which is the logarithm of odds, usually expressed using the natural log (base e).

Adding log odds to the table:

	odds	prob	prob diff	log odds	log odds diff
prior	10.000000	0.909091	--	2.302585	--
1 student	3.333333	0.769231	-13.986014	1.203973	-1.098612
2 students	1.111111	0.526316	-24.291498	0.105361	-1.098612
3 students	0.370370	0.270270	-25.604552	-0.993252	-1.098612

You might notice:

- When probability is greater than 0.5, odds are greater than 1, and log odds are positive.
- When probability is less than 0.5, odds are less than 1, and log odds are negative.

You might also notice that the log odds are equally spaced. The change in log odds after each update is the logarithm of the likelihood ratio.

```
np.log(1/3)
-1.0986122886681098
```

That's true in this example, and we can show that it's true in general by taking the log of both sides of Bayes's rule:

$$\log O(H|F) = \log O(H) + \log \frac{P(F|H)}{P(F|notH)}$$

On a log odds scale, a Bayesian update is additive. So if F^x means that x female students arrive while I am waiting, the posterior log odds that I am in the right room are:

$$\log O(H|F^x) = \log O(H) + x \log \frac{P(F|H)}{P(F|notH)}$$

This equation represents a linear relationship between the log likelihood ratio and the posterior log odds.

In this example the linear equation is exact, but even when it's not, it is common to use a linear function to model the relationship between an explanatory variable, x , and a dependent variable expressed in log odds, like this:

$$\log O(H|x) = \beta_0 + \beta_1 x$$

where β_0 and β_1 are unknown parameters:

- The intercept, β_0 , is the log odds of the hypothesis when x is 0.
- The slope, β_1 , is the log of the likelihood ratio.

This equation is the basis of logistic regression.

The Space Shuttle Problem

As an example of logistic regression, I'll solve a problem from Cameron Davidson-Pilon's book, *Bayesian Methods for Hackers*. He writes:

On January 28, 1986, the twenty-fifth flight of the US space shuttle program ended in disaster when one of the rocket boosters of the shuttle Challenger exploded shortly after lift-off, killing all 7 crew members. The presidential commission on the accident concluded that it was caused by the failure of an O-ring in a field joint on the rocket booster, and that this failure was due to a faulty design that made the O-ring unacceptably sensitive to a number of factors including outside temperature. Of the previous 24 flights, data were available on failures of O-rings on 23 (one was lost at sea), and these data were discussed on the evening preceding the Challenger launch, but unfortunately only the data corresponding to the 7 flights on which there was a damage incident were considered important and these were thought to show no obvious trend.

The dataset is originally from [this paper](#), but also available from [Davidson-Pilon](#).

Here are the first few rows:

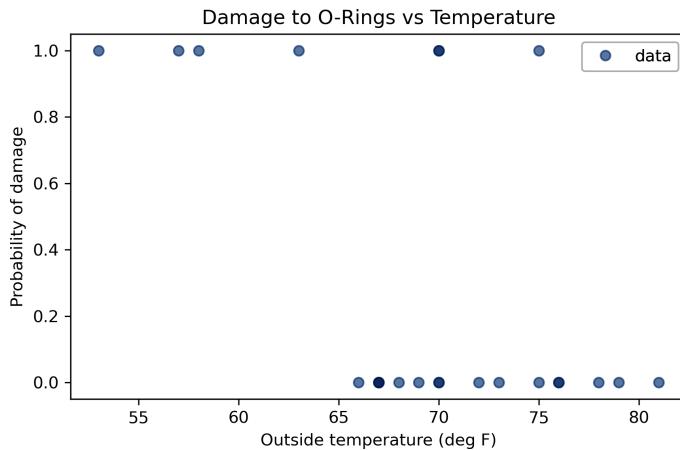
	Date	Temperature	Damage
0	1981-04-12	66	0
1	1981-11-12	70	1
2	1982-03-22	69	0
4	1982-01-11	68	0
5	1983-04-04	67	0

The columns are:

- **Date:** The date of launch,
- **Temperature:** Outside temperature in Fahrenheit (F), and
- **Damage:** 1 if there was a damage incident and 0 otherwise.

There are 23 launches in the dataset, 7 with damage incidents.

The following figure shows the relationship between damage and temperature:



When the outside temperature was below 65 degrees F, there was always damage to the O-rings. When the temperature was above 65 degrees F, there was usually no damage.

Based on this figure, it seems plausible that the probability of damage is related to temperature. If we assume this probability follows a logistic model, we can write

$$\log O(H|x) = \beta_0 + \beta_1 x$$

where H is the hypothesis that the O-rings will be damaged, x is temperature, and β_0 and β_1 are the parameters we will estimate. For reasons I'll explain soon, I'll define x to be temperature shifted by an offset so its mean is 0:

```
offset = data['Temperature'].mean().round()
data['x'] = data['Temperature'] - offset
offset
70.0
```

And for consistency I'll create a copy of the `Damage` columns called `y`:

```
data['y'] = data['Damage']
```

Before doing a Bayesian update, I'll use `statsmodels` to run a conventional (non-Bayesian) logistic regression:

```
import statsmodels.formula.api as smf

formula = 'y ~ x'
results = smf.logit(formula, data=data).fit(disp=False)
results.params

Intercept    -1.208490
x             -0.232163
dtype: float64
```

`results` contains a “point estimate” for each parameter, that is, a single value rather than a posterior distribution.

The intercept is about -1.2, and the estimated slope is about -0.23. To see what these parameters mean, I'll use them to compute probabilities for a range of temperatures. Here's the range:

```
inter = results.params['Intercept']
slope = results.params['x']
xs = np.arange(53, 83) - offset
```

We can use the logistic regression equation to compute log odds:

```
log_odds = inter + slope * xs
```

And then convert to probabilities:

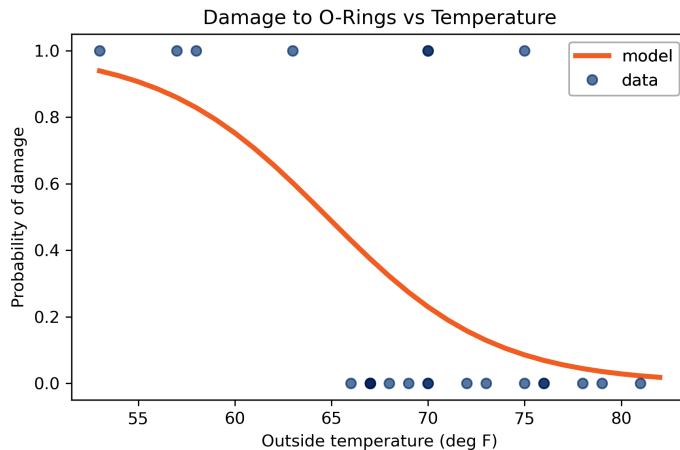
```
odds = np.exp(log_odds)
ps = odds / (odds + 1)
```

Converting log odds to probabilities is a common enough operation that it has a name, `expit`, and SciPy provides a function that computes it:

```
from scipy.special import expit

ps = expit(inter + slope * xs)
```

Here's what the logistic model looks like with these estimated parameters:



At low temperatures, the probability of damage is high; at high temperatures, it drops off to near 0.

But that's based on conventional logistic regression. Now we'll do the Bayesian version.

Prior Distribution

I'll use uniform distributions for both parameters, using the point estimates from the previous section to help me choose the upper and lower bounds:

```
from utils import make_uniform

qs = np.linspace(-5, 1, num=101)
prior_inter = make_uniform(qs, 'Intercept')

qs = np.linspace(-0.8, 0.1, num=101)
prior_slope = make_uniform(qs, 'Slope')
```

We can use `make_joint` to construct the joint prior distribution:

```
from utils import make_joint

joint = make_joint(prior_inter, prior_slope)
```

The values of `intercept` run across the columns, and the values of `slope` run down the rows.

For this problem, it will be convenient to “stack” the prior so the parameters are levels in a `MultiIndex`:

```
from empiricaldist import Pmf

joint_pmf = Pmf(joint.stack())
joint_pmf.head()
```

		probs
Slope	Intercept	
-0.8	-5.00	0.000098
	-4.94	0.000098
	-4.88	0.000098

`joint_pmf` is a `Pmf` with two levels in the index, one for each parameter. That makes it easy to loop through possible pairs of parameters, as we'll see in the next section.

Likelihood

To do the update, we have to compute the likelihood of the data for each possible pair of parameters.

To make that easier, I'm going to group the data by temperature, `x`, and count the number of launches and damage incidents at each temperature:

```
grouped = data.groupby('x')['y'].agg(['count', 'sum'])
grouped.head()
```

	count	sum
x		
-17.0	1	1
-13.0	1	1
-12.0	1	1
-7.0	1	1
-4.0	1	0

The result is a `DataFrame` with two columns: `count` is the number of launches at each temperature; `sum` is the number of damage incidents. To be consistent with the parameters of the binomial distributions, I'll assign them to variables named `ns` and `ks`:

```
ns = grouped['count']
ks = grouped['sum']
```

To compute the likelihood of the data, let's assume temporarily that the parameters we just estimated, `slope` and `inter`, are correct.

We can use them to compute the probability of damage at each launch temperature, like this:

```
xs = grouped.index  
ps = expit(inter + slope * xs)
```

ps contains the probability of damage for each launch temperature, according to the model.

Now, for each temperature we have ns, ps, and ks; we can use the binomial distribution to compute the likelihood of the data.

```
from scipy.stats import binom  
  
likes = binom.pmf(ks, ns, ps)  
likes  
  
array([0.93924781, 0.85931657, 0.82884484, 0.60268105, 0.56950687,  
       0.24446388, 0.67790595, 0.72637895, 0.18815003, 0.8419509 ,  
       0.87045398, 0.15645171, 0.86667894, 0.95545945, 0.96435859,  
       0.97729671])
```

Each element of likes is the probability of seeing k damage incidents in n launches if the probability of damage is p. The likelihood of the whole dataset is the product of this array:

```
likes.prod()  
0.0004653644508250066
```

That's how we compute the likelihood of the data for a particular pair of parameters. Now we can compute the likelihood of the data for all possible pairs:

```
likelihood = joint_pmf.copy()  
for slope, inter in joint_pmf.index:  
    ps = expit(inter + slope * xs)  
    likes = binom.pmf(ks, ns, ps)  
    likelihood[slope, inter] = likes.prod()
```

To initialize likelihood, we make a copy of joint_pmf, which is a convenient way to make sure that likelihood has the same type, index, and data type as joint_pmf.

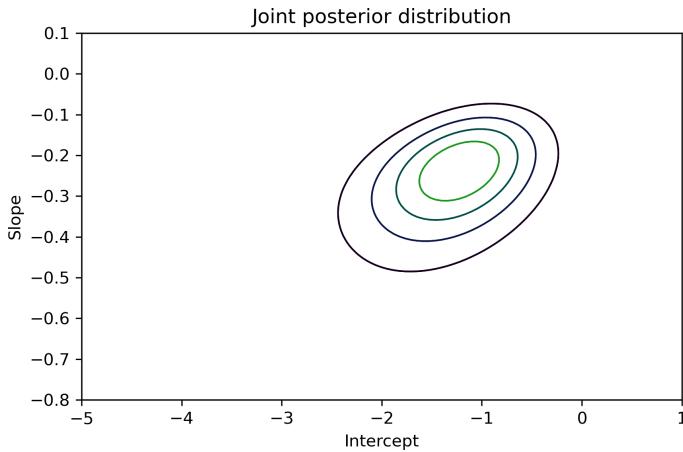
The loop iterates through the parameters. For each possible pair, it uses the logistic model to compute ps, computes the likelihood of the data, and assigns the result to a row in likelihood.

The Update

Now we can compute the posterior distribution in the usual way:

```
posterior_pmf = joint_pmf * likelihood  
posterior_pmf.normalize()
```

If we unstack the posterior `Pmf` we can make a contour plot of the joint posterior distribution:



The ovals in the contour plot are aligned along a diagonal, which indicates that there is some correlation between `slope` and `inter` in the posterior distribution.

But the correlation is weak, which is one of the reasons we subtracted off the mean launch temperature when we computed `x`; centering the data minimizes the correlation between the parameters.

Exercise 16-1.

To see why this matters, go back and set `offset=60` and run the analysis again. The slope should be the same, but the intercept will be different. And if you plot the joint distribution, the contours you get will be elongated, indicating stronger correlation between the estimated parameters.

In theory, this correlation is not a problem, but in practice it is. With uncentered data, the posterior distribution is more spread out, so it's harder to cover with the joint prior distribution. Centering the data maximizes the precision of the estimates; with uncentered data, we have to do more computation to get the same precision.

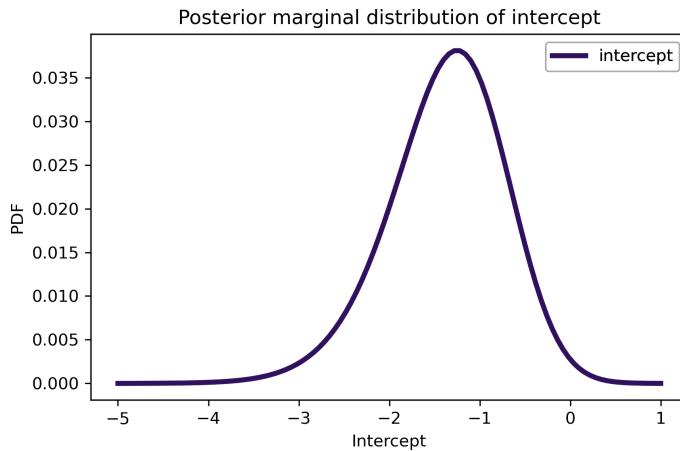
Marginal Distributions

Finally, we can extract the marginal distributions:

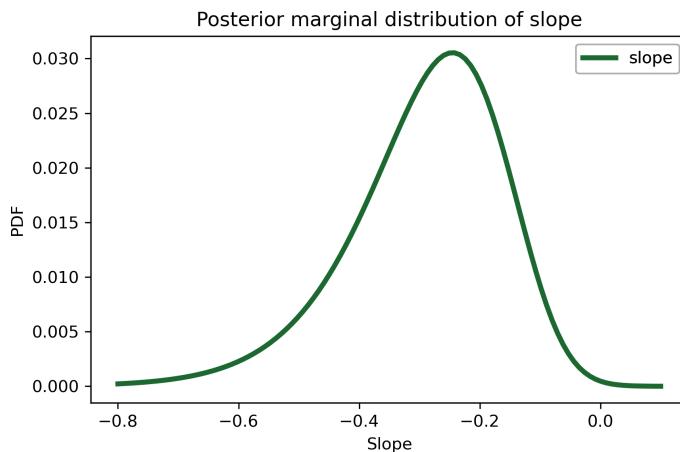
```
from utils import marginal

marginal_inter = marginal(joint_posterior, 0)
marginal_slope = marginal(joint_posterior, 1)
```

Here's the posterior distribution of `intercept`:



And here's the posterior distribution of `slope`:



Transforming Distributions

Let's interpret these parameters. Recall that the intercept is the log odds of the hypothesis when x is 0, which is when temperature is about 70 degrees F (the value of `offset`). So we can interpret the quantities in `marginal_inter` as log odds.

To convert them to probabilities, I'll use the following function, which transforms the quantities in a `Pmf` by applying a given function:

```

def transform(pmf, func):
    """Transform the quantities in a Pmf."""
    ps = pmf.ps
    qs = func(pmf.qs)
    return Pmf(ps, qs, copy=True)

```

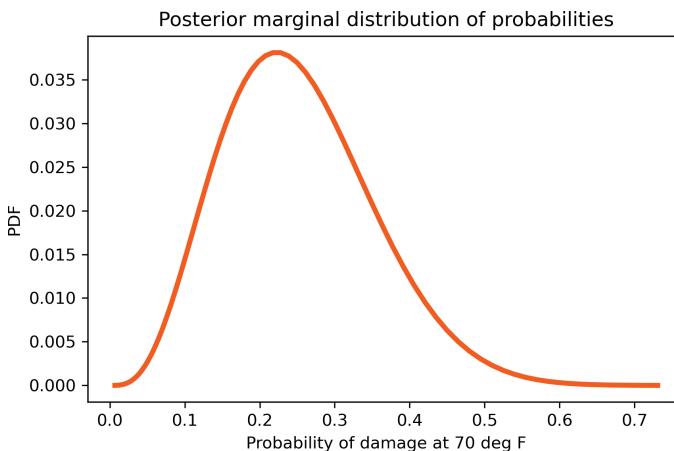
If we call `transform` and pass `expit` as a parameter, it transforms the log odds in `marginal_inter` into probabilities and returns the posterior distribution of `inter` expressed in terms of probabilities:

```
marginal_probs = transform(marginal_inter, expit)
```

`Pmf` provides a `transform` method that does the same thing:

```
marginal_probs = marginal_inter.transform(expit)
```

Here's the posterior distribution for the probability of damage at 70 degrees F:



The mean of this distribution is about 22%, which is the probability of damage at 70 degrees F, according to the model.

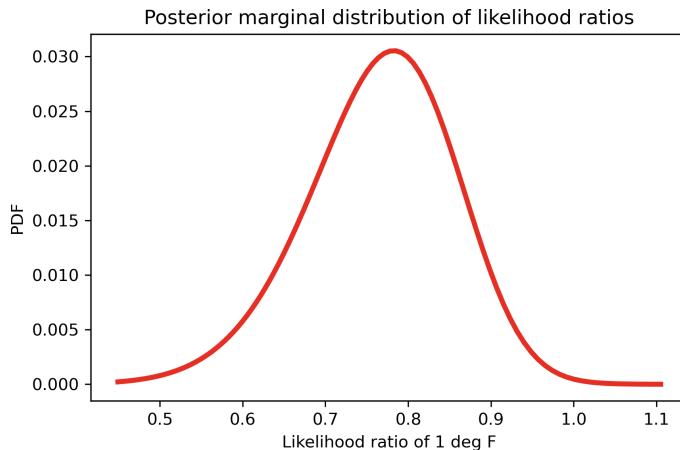
This result shows the second reason I defined `x` to be zero when temperature is 70 degrees F; this way, the intercept corresponds to the probability of damage at a relevant temperature, rather than 0 degrees F.

Now let's look more closely at the estimated slope. In the logistic model, the parameter β_1 is the log of the likelihood ratio.

So we can interpret the quantities in `marginal_slope` as log likelihood ratios, and we can use `exp` to transform them to likelihood ratios (also known as Bayes factors):

```
marginal_lr = marginal_slope.transform(np.exp)
```

The result is the posterior distribution of likelihood ratios; here's what it looks like:



The mean of this distribution is about 0.75, which means that each additional degree Fahrenheit provides evidence against the possibility of damage, with a likelihood ratio (Bayes factor) of 0.75.

Notice:

- I computed the posterior mean of the probability of damage at 70 degrees F by transforming the marginal distribution of the intercept to the marginal distribution of probability, and then computing the mean.
- I computed the posterior mean of the likelihood ratio by transforming the marginal distribution of slope to the marginal distribution of likelihood ratios, and then computing the mean.

This is the correct order of operations, as opposed to computing the posterior means first and then transforming them.

Predictive Distributions

In the logistic model, the parameters are interpretable, at least after transformation. But often what we care about are predictions, not parameters. In the Space Shuttle Problem, the most important prediction is, “What is the probability of O-ring damage if the outside temperature is 31 degrees F?”

To make that prediction, I'll draw a sample of parameter pairs from the posterior distribution:

```
sample = posterior_pmf.choice(101)
```

The result is an array of 101 tuples, each representing a possible pair of parameters. I chose this sample size to make the computation fast. Increasing it would not change the results much, but they would be a little more precise.

To generate predictions, I'll use a range of temperatures from 31 degrees F (the temperature when the Challenger launched) to 82 degrees F (the highest observed temperature):

```
temps = np.arange(31, 83)
xs = temps - offset
```

The following loop uses `xs` and the sample of parameters to construct an array of predicted probabilities:

```
pred = np.empty((len(sample), len(xs)))

for i, (slope, inter) in enumerate(sample):
    pred[i] = expit(inter + slope * xs)
```

The result has one column for each value in `xs` and one row for each element of `sample`.

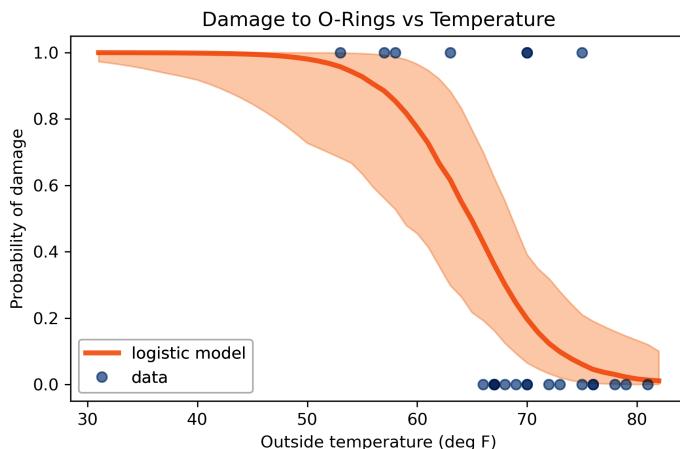
In each column, I'll compute the median to quantify the central tendency and a 90% credible interval to quantify the uncertainty.

`np.percentile` computes the given percentiles; with the argument `axis=0`, it computes them for each column:

```
low, median, high = np.percentile(pred, [5, 50, 95], axis=0)
```

The results are arrays containing predicted probabilities for the lower bound of the 90% CI, the median, and the upper bound of the CI.

Here's what they look like:



According to these results, the probability of damage to the O-rings at 80 degrees F is near 2%, but there is some uncertainty about that prediction; the upper bound of the CI is around 10%.

At 60 degrees F, the probability of damage is near 80%, but the CI is even wider, from 48% to 97%.

But the primary goal of the model is to predict the probability of damage at 31 degrees F, and the answer is at least 97%, and more likely to be more than 99.9%.

One conclusion we might draw is this: if the people responsible for the Challenger launch had taken into account all of the data, and not just the seven damage incidents, they could have predicted that the probability of damage at 31 degrees F was nearly certain. If they had, it seems likely they would have postponed the launch.

At the same time, if they considered the previous figure, they might have realized that the model makes predictions that extend far beyond the data. When we extrapolate like that, we have to remember not just the uncertainty quantified by the model, which we expressed as a credible interval; we also have to consider the possibility that the model itself is unreliable.

This example is based on a logistic model, which assumes that each additional degree of temperature contributes the same amount of evidence in favor of (or against) the possibility of damage. Within a narrow range of temperatures, that might be a reasonable assumption, especially if it is supported by data. But over a wider range, and beyond the bounds of the data, reality has no obligation to stick to the model.

Empirical Bayes

In this chapter I used `statsmodels` to compute the parameters that maximize the probability of the data, and then used those estimates to choose the bounds of the uniform prior distributions. It might have occurred to you that this process uses the data twice, once to choose the priors and again to do the update. If that bothers you, you are not alone. The process I used is an example of what's called the [Empirical Bayes method](#), although I don't think that's a particularly good name for it.

Although it might seem problematic to use the data twice, in these examples, it is not. To see why, consider an alternative: instead of using the estimated parameters to choose the bounds of the prior distribution, I could have used uniform distributions with much wider ranges. In that case, the results would be the same; the only difference is that I would spend more time computing likelihoods for parameters where the posterior probabilities are negligibly small.

So you can think of this version of Empirical Bayes as an optimization that minimizes computation by putting the prior distributions where the likelihood of the data is worth computing. This optimization doesn't affect the results, so it doesn't "double-count" the data.

Summary

So far we have seen three ways to represent degrees of confidence in a hypothesis: probability, odds, and log odds. When we write Bayes's rule in terms of log odds, a Bayesian update is the sum of the prior and the likelihood; in this sense, Bayesian statistics is the arithmetic of hypotheses and evidence.

This form of Bayes's theorem is also the foundation of logistic regression, which we used to infer parameters and make predictions. In the Space Shuttle Problem, we modeled the relationship between temperature and the probability of damage, and showed that the Challenger disaster might have been predictable. But this example is also a warning about the hazards of using a model to extrapolate far beyond the data.

In the exercises below you'll have a chance to practice the material in this chapter, using log odds to evaluate a political pundit and using logistic regression to model diagnosis rates for attention deficit hyperactivity disorder (ADHD).

In the next chapter we'll move from logistic regression to linear regression, which we will use to model changes over time in temperature, snowfall, and the marathon world record.

More Exercises

Exercise 16-2.

Suppose a political pundit claims to be able to predict the outcome of elections, but instead of picking a winner, they give each candidate a probability of winning. With that kind of prediction, it can be hard to say whether it is right or wrong.

For example, suppose the pundit says that Alice has a 70% chance of beating Bob, and then Bob wins the election. Does that mean the pundit was wrong?

One way to answer this question is to consider two hypotheses:

- H : The pundit's algorithm is legitimate; the probabilities it produces are correct in the sense that they accurately reflect the candidates' probabilities of winning.
- $\text{not } H$: The pundit's algorithm is bogus; the probabilities it produces are random values with a mean of 50%.

If the pundit says Alice has a 70% chance of winning, and she does, that provides evidence in favor of H with likelihood ratio 70/50.

If the pundit says Alice has a 70% chance of winning, and she loses, that's evidence against H with a likelihood ratio of 50/30.

Suppose we start with some confidence in the algorithm, so the prior odds are 4 to 1. And suppose the pundit generates predictions for three elections:

- In the first election, the pundit says Alice has a 70% chance of winning and she does.
- In the second election, the pundit says Bob has a 30% chance of winning and he does.
- In the third election, the pundit says Carol has a 90% chance of winning and she does.

What is the log likelihood ratio for each of these outcomes? Use the log-odds form of Bayes's rule to compute the posterior log odds for H after these outcomes. In total, do these outcomes increase or decrease your confidence in the pundit?

If you are interested in this topic, you can [read more about it in this blog post](#).

Exercise 16-3.

An article in the *New England Journal of Medicine* reports results from a study that looked at the diagnosis rate of attention deficit hyperactivity disorder (ADHD) as a function of birth month: “[Attention Deficit–Hyperactivity Disorder and Month of School Enrollment](#)”.

They found that children born in June, July, and August were substantially more likely to be diagnosed with ADHD, compared to children born in September, but only in states that use a September cutoff for children to enter kindergarten. In these states, children born in August start school almost a year younger than children born in September. The authors of the study suggest that the cause is “age-based variation in behavior that may be attributed to ADHD rather than to the younger age of the children”.

Use the methods in this chapter to estimate the probability of diagnosis as a function of birth month. The notebook for this chapter provides the data and some suggestions for getting started.

CHAPTER 17

Regression

In the previous chapter we saw several examples of logistic regression, which is based on the assumption that the likelihood of an outcome, expressed in the form of log odds, is a linear function of some quantity (continuous or discrete).

In this chapter we'll work on examples of simple linear regression, which models the relationship between two quantities. Specifically, we'll look at changes over time in snowfall and the marathon world record.

The models we'll use have three parameters, so you might want to review the tools we used for the three-parameter model in [Chapter 15](#).

More Snow?

I am under the impression that we don't get as much snow around here as we used to. By "around here" I mean Norfolk County, Massachusetts, where I was born, grew up, and currently live. And by "used to" I mean compared to when I was young, like in 1978 when we got [27 inches of snow](#) and I didn't have to go to school for a couple of weeks.

Fortunately, we can test my conjecture with data. Norfolk County happens to be the location of the [Blue Hill Meteorological Observatory](#), which keeps the oldest continuous weather record in North America.

Data from this and many other weather stations is available from the [National Oceanic and Atmospheric Administration](#) (NOAA). I collected data from the Blue Hill Observatory from May 11, 1967 to May 11, 2020.

We can use pandas to read the data into `DataFrame`:

```
import pandas as pd  
  
df = pd.read_csv('2239075.csv', parse_dates=[2])
```

The columns we'll use are:

- DATE, which is the date of each observation,
- SNOW, which is the total snowfall in inches.

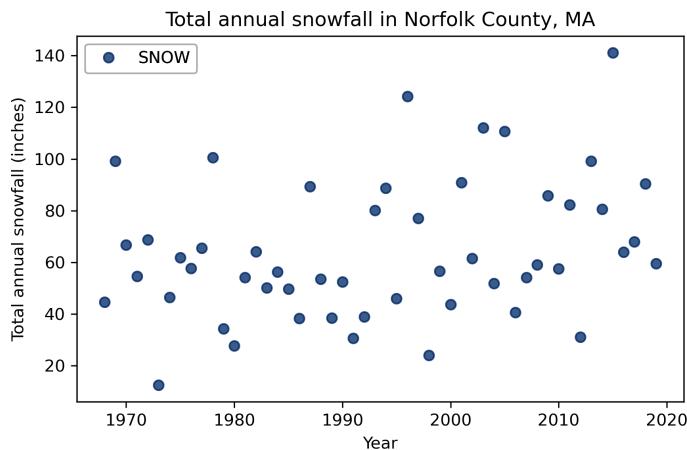
I'll add a column that contains just the year part of the dates:

```
df['YEAR'] = df['DATE'].dt.year
```

And use `groupby` to add up the total snowfall in each year:

```
snow = df.groupby('YEAR')['SNOW'].sum()
```

The following figure shows total snowfall during each of the complete years in my lifetime:



Looking at this plot, it's hard to say whether snowfall is increasing, decreasing, or unchanged. In the last decade, we've had several years with more snow than 1978, including 2015, which was the snowiest winter in the Boston area in modern history, with a total of 141 inches.

This kind of question—looking at noisy data and wondering whether it is going up or down—is precisely the question we can answer with Bayesian regression.

Regression Model

The foundation of regression (Bayesian or not) is the assumption that a time series like this is the sum of two parts:

1. A linear function of time, and
2. A series of random values drawn from a distribution that is not changing over time.

Mathematically, the regression model is

$$y = ax + b + \epsilon$$

where y is the series of measurements (snowfall in this example), x is the series of times (years) and ϵ is the series of random values.

a and b are the slope and intercept of the line through the data. They are unknown parameters, so we will use the data to estimate them.

We don't know the distribution of ϵ , so we'll make the additional assumption that it is a normal distribution with mean 0 and unknown standard deviation, σ .

To see whether this assumption is reasonable, I'll plot the distribution of total snowfall and a normal model with the same mean and standard deviation.

Here's a `Pmf` object that represents the distribution of snowfall:

```
from empiricaldist import Pmf

pmf_snowfall = Pmf.from_seq(snow)
```

And here are the mean and standard deviation of the data:

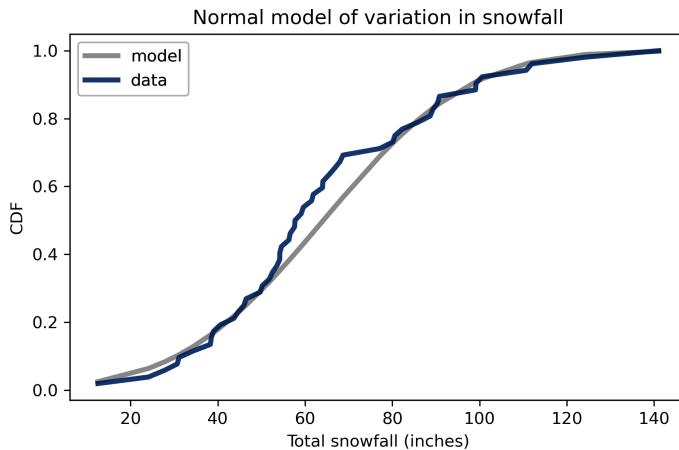
```
mean, std = pmf_snowfall.mean(), pmf_snowfall.std()
mean, std
(64.19038461538462, 26.288021984395684)
```

I'll use the `norm` object from SciPy to compute the CDF of a normal distribution with the same mean and standard deviation:

```
from scipy.stats import norm

dist = norm(mean, std)
qs = pmf_snowfall.qs
ps = dist.cdf(qs)
```

Here's what the distribution of the data looks like compared to the normal model:



We've had more winters below the mean than expected, but overall this looks like a reasonable model.

Least Squares Regression

Our regression model has three parameters: slope, intercept, and standard deviation of ϵ . Before we can estimate them, we have to choose priors. To help with that, I'll use `statsmodels` to fit a line to the data by **least squares regression**.

First, I'll use `reset_index` to convert `snow`, which is a `Series`, to a `DataFrame`:

```
data = snow.reset_index()
data.head(3)
```

	YEAR	SNOW
0	1968	44.7
1	1969	99.2
2	1970	66.8

The result is a `DataFrame` with two columns, `YEAR` and `SNOW`, in a format we can use with `statsmodels`.

As we did in the previous chapter, I'll center the data by subtracting off the mean:

```
offset = data['YEAR'].mean().round()
data['x'] = data['YEAR'] - offset
offset
1994.0
```

And I'll add a column to `data` so the dependent variable has a standard name:

```
data['y'] = data['SNOW']
```

Now, we can use `statsmodels` to compute the least squares fit to the data and estimate `slope` and `intercept`:

```
import statsmodels.formula.api as smf

formula = 'y ~ x'
results = smf.ols(formula, data=data).fit()
results.params

Intercept    64.446325
x            0.511880
dtype: float64
```

The intercept, about 64 inches, is the expected snowfall when $x=0$, which is the beginning of 1994. The estimated slope indicates that total snowfall is increasing at a rate of about 0.5 inches per year.

`results` also provides `resid`, which is an array of residuals, that is, the differences between the data and the fitted line. The standard deviation of the residuals is an estimate of `sigma`:

```
results.resid.std()

25.385680731210616
```

We'll use these estimates to choose prior distributions for the parameters.

Priors

I'll use uniform distributions for all three parameters:

```
import numpy as np
from utils import make_uniform

qs = np.linspace(-0.5, 1.5, 51)
prior_slope = make_uniform(qs, 'Slope')

qs = np.linspace(54, 75, 41)
prior_inter = make_uniform(qs, 'Intercept')

qs = np.linspace(20, 35, 31)
prior_sigma = make_uniform(qs, 'Sigma')
```

I made the prior distributions different lengths for two reasons. First, if we make a mistake and use the wrong distribution, it will be easier to catch the error if they are all different lengths.

Second, it provides more precision for the most important parameter, `slope`, and spends less computational effort on the least important, `sigma`.

In “Three-Parameter Model” on page 217 we made a joint distribution with three parameters. I’ll wrap that process in a function:

```
from utils import make_joint

def make_joint3(pmf1, pmf2, pmf3):
    """Make a joint distribution with three parameters."""
    joint2 = make_joint(pmf2, pmf1).stack()
    joint3 = make_joint(pmf3, joint2).stack()
    return Pmf(joint3)
```

And use it to make a `Pmf` that represents the joint distribution of the three parameters:

```
prior = make_joint3(prior_slope, prior_inter, prior_sigma)
prior.head(3)
```

probs			
Slope	Intercept	Sigma	
-0.5	54.0	20.0	0.000015
		20.5	0.000015
		21.0	0.000015

The index of `Pmf` has three columns, containing values of `slope`, `inter`, and `sigma`, in that order.

With three parameters, the size of the joint distribution starts to get big. Specifically, it is the product of the lengths of the prior distributions. In this example, the prior distributions have 51, 41, and 31 values, so the length of the joint prior is 64,821.

Likelihood

Now we’ll compute the likelihood of the data. To demonstrate the process, let’s assume temporarily that the parameters are known.

```
inter = 64
slope = 0.51
sigma = 25
```

I’ll extract the `xs` and `ys` from `data` as `Series` objects:

```
xs = data['x']
ys = data['y']
```

And compute the “residuals”, which are the differences between the actual values, `ys`, and the values we expect based on `slope` and `inter`:

```
expected = slope * xs + inter
resid = ys - expected
```

According to the model, the residuals should follow a normal distribution with mean 0 and standard deviation `sigma`. So we can compute the likelihood of each residual value using `norm` from SciPy:

```
densities = norm(0, sigma).pdf(resid)
```

The result is an array of probability densities, one for each element of the dataset; their product is the likelihood of the data.

```
likelihood = densities.prod()  
likelihood  
1.3551948769061074e-105
```

As we saw in the previous chapter, the likelihood of any particular dataset tends to be small. If it's too small, we might exceed the limits of floating-point arithmetic. When that happens, we can avoid the problem by computing likelihoods under a log transform. But in this example that's not necessary.

The Update

Now we're ready to do the update. First, we need to compute the likelihood of the data for each possible set of parameters:

```
likelihood = prior.copy()  
  
for slope, inter, sigma in prior.index:  
    expected = slope * xs + inter  
    resid = ys - expected  
    densities = norm.pdf(resid, 0, sigma)  
    likelihood[slope, inter, sigma] = densities.prod()
```

This computation takes longer than many of the previous examples. We are approaching the limit of what we can do with grid approximations.

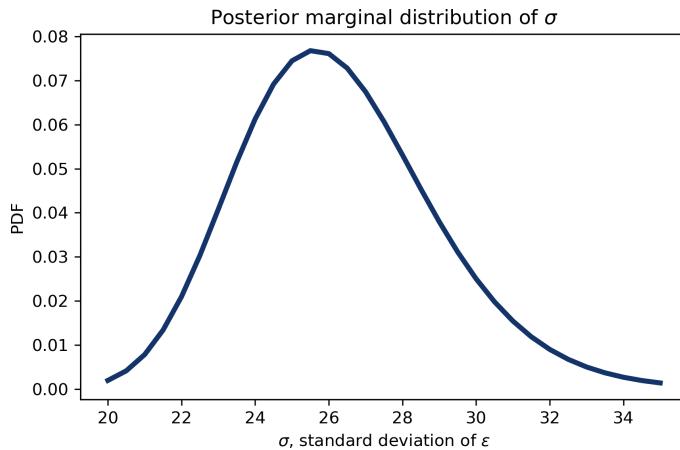
Nevertheless, we can do the update in the usual way:

```
posterior = prior * likelihood  
posterior.normalize()
```

The result is a `Pmf` with a three-level index containing values of `slope`, `inter`, and `sigma`. To get the marginal distributions from the joint posterior, we can use `Pmf.marginal`, which we saw in “[Three-Parameter Model](#)” on page 217:

```
posterior_slope = posterior.marginal(0)  
posterior_inter = posterior.marginal(1)  
posterior_sigma = posterior.marginal(2)
```

Here's the posterior distribution for `sigma`:



The most likely values for `sigma` are near 26 inches, which is consistent with our estimate based on the standard deviation of the data.

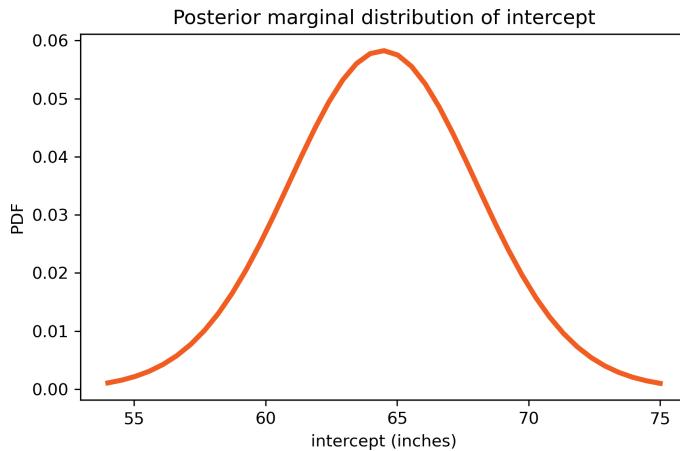
However, to say whether snowfall is increasing or decreasing, we don't really care about `sigma`. It is a "nuisance parameter", so-called because we have to estimate it as part of the model, but we don't need it to answer the questions we are interested in.

Nevertheless, it is good to check the marginal distributions to make sure

- The location is consistent with our expectations, and
- The posterior probabilities are near 0 at the extremes of the range, which indicates that the prior distribution covers all parameters with non-negligible probability.

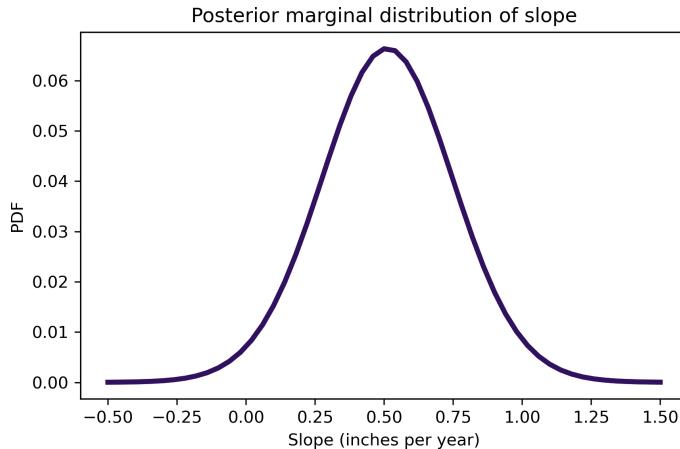
In this example, the posterior distribution of `sigma` looks fine.

Here's the posterior distribution of `inter`:



The posterior mean is about 64 inches, which is the expected amount of snow during the year at the midpoint of the range, 1994.

And finally, here's the posterior distribution of `slope`:



The posterior mean is about 0.51 inches, which is consistent with the estimate we got from least squared regression.

The 90% credible interval is from 0.1 to 0.9, which indicates that our uncertainty about this estimate is pretty high. In fact, there is still a small posterior probability (about 2%) that the slope is negative.

However, it is more likely that my conjecture was wrong: we are actually getting more snow around here than we used to, increasing at a rate of about a half-inch per year, which is substantial. On average, we get an additional 25 inches of snow per year than we did when I was young.

This example shows that with slow-moving trends and noisy data, your instincts can be misleading.

Now, you might suspect that I overestimate the amount of snow when I was young because I enjoyed it, and underestimate it now because I don't. But you would be mistaken.

During the Blizzard of 1978, we did not have a snowblower and my brother and I had to shovel. My sister got a pass for no good reason. Our driveway was about 60 feet long and three cars wide near the garage. And we had to shovel Mr. Crocker's driveway, too, for which we were not allowed to accept payment. Furthermore, as I recall it was during this excavation that I accidentally hit my brother with a shovel on the head, and it bled a lot because, you know, scalp wounds.

Anyway, the point is that I don't think I overestimate the amount of snow when I was young because I have fond memories of it.

Marathon World Record

For many running events, if you plot the world record pace over time, the result is a remarkably straight line. People, [including me](#), have speculated about possible reasons for this phenomenon.

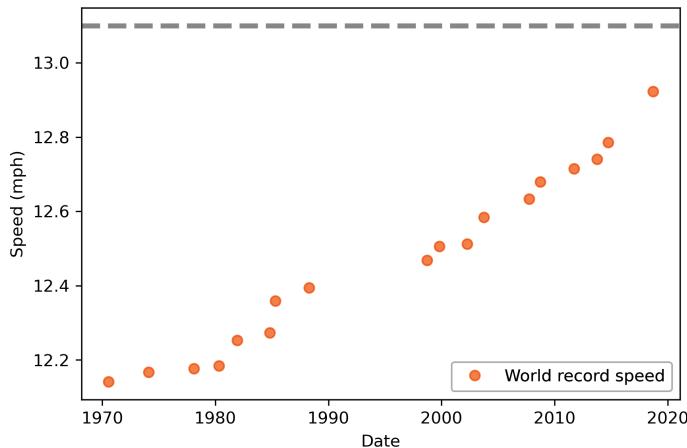
People have also speculated about when, if ever, the world record time for the marathon will be less than two hours. (Note: In 2019, Eliud Kipchoge ran the marathon distance in under two hours, which is an astonishing achievement that I fully appreciate, but for several reasons it did not count as a world record.)

So, as a second example of Bayesian regression, we'll consider the world record progression for the marathon (for male runners), estimate the parameters of a linear model, and use the model to predict when a runner will break the two-hour barrier.

In the notebook for this chapter, you can see how I loaded and cleaned the data. The result is a `DataFrame` that contains the following columns (and additional information we won't use):

- `date`, which is a pandas `Timestamp` representing the date when the world record was broken, and
- `speed`, which records the record-breaking pace in mph.

Here's what the results look like, starting in 1970:



The data points fall approximately on a line, although it's possible that the slope is increasing.

To prepare the data for regression, I'll subtract away the approximate midpoint of the time interval, 1995:

```
offset = pd.to_datetime('1995')
timedelta = table['date'] - offset
```

When we subtract two `Timestamp` objects, the result is a “time delta”, which we can convert to seconds and then to years:

```
data['x'] = timedelta.dt.total_seconds() / 3600 / 24 / 365.24
```

As in the previous example, I'll use least squares regression to compute point estimates for the parameters, which will help with choosing priors:

```
import statsmodels.formula.api as smf

formula = 'y ~ x'
results = smf.ols(formula, data=data).fit()
results.params

Intercept    12.460507
x            0.015464
dtype: float64
```

The estimated intercept is about 12.5 mph, which is the interpolated world record pace for 1995. The estimated slope is about 0.015 mph per year, which is the rate the world record pace is increasing, according to the model.

Again, we can use the standard deviation of the residuals as a point estimate for sigma:

```
results.resid.std()  
0.04139961220193225
```

These parameters give us a good idea where we should put the prior distributions.

The Priors

Here are the prior distributions I chose for slope, intercept, and sigma:

```
qs = np.linspace(0.012, 0.018, 51)  
prior_slope = make_uniform(qs, 'Slope')  
  
qs = np.linspace(12.4, 12.5, 41)  
prior_inter = make_uniform(qs, 'Intercept')  
  
qs = np.linspace(0.01, 0.21, 31)  
prior_sigma = make_uniform(qs, 'Sigma')
```

And here's the joint prior distribution:

```
prior = make_joint3(prior_slope, prior_inter, prior_sigma)  
prior.head()
```

probs		
Slope	Intercept	Sigma
0.012	12.4	0.010000 0.000015
		0.016667 0.000015
		0.023333 0.000015

Now we can compute likelihoods as in the previous example:

```
xs = data['x']  
ys = data['y']  
likelihood = prior.copy()  
  
for slope, inter, sigma in prior.index:  
    expected = slope * xs + inter  
    resid = ys - expected  
    densities = norm.pdf(resid, 0, sigma)  
    likelihood[slope, inter, sigma] = densities.prod()
```

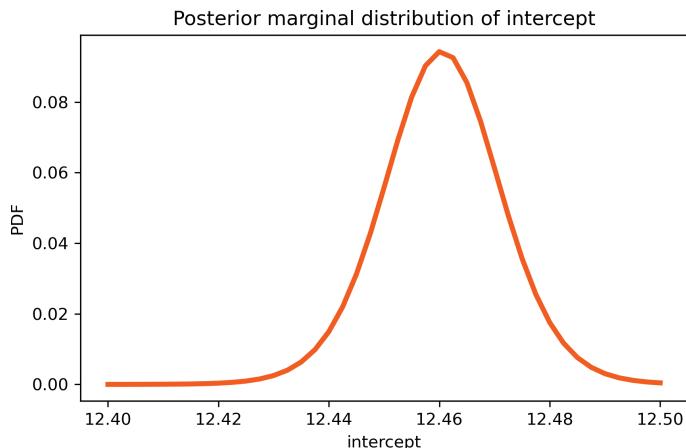
Now we can do the update in the usual way:

```
posterior = prior * likelihood  
posterior.normalize()
```

And unpack the marginals:

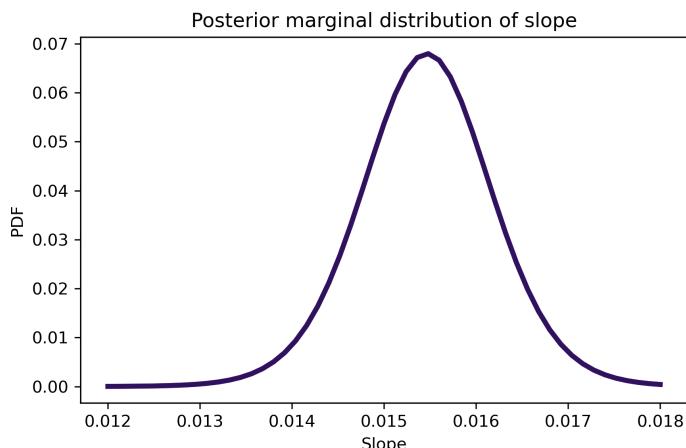
```
posterior_slope = posterior.marginal(0)
posterior_inter = posterior.marginal(1)
posterior_sigma = posterior.marginal(2)
```

Here's the posterior distribution of `inter`:



The posterior mean is about 12.5 mph, which is the world record marathon pace the model predicts for the midpoint of the date range, 1994.

And here's the posterior distribution of `slope`:



The posterior mean is about 0.015 mph per year, or 0.15 mph per decade.

That's interesting, but it doesn't answer the question we're interested in: when will there be a two-hour marathon? To answer that, we have to make predictions.

Prediction

To generate predictions, I'll draw a sample from the posterior distribution of parameters, then use the regression equation to combine the parameters with the data.

Pmf provides `choice`, which we can use to draw a random sample with replacement, using the posterior probabilities as weights:

```
sample = posterior.choice(101)
```

The result is an array of tuples. Looping through the sample, we can use the regression equation to generate predictions for a range of `xs`:

```
xs = np.arange(-25, 50, 2)
pred = np.empty((len(sample), len(xs)))

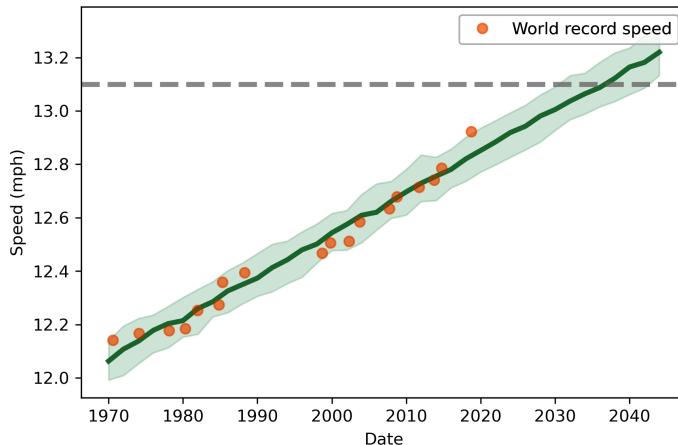
for i, (slope, inter, sigma) in enumerate(sample):
    epsilon = norm(0, sigma).rvs(len(xs))
    pred[i] = inter + slope * xs + epsilon
```

Each prediction is an array with the same length as `xs`, which I store as a row in `pred`. So the result has one row for each sample and one column for each value of `x`.

We can use `percentile` to compute the 5th, 50th, and 95th percentiles in each column:

```
low, median, high = np.percentile(pred, [5, 50, 95], axis=0)
```

To show the results, I'll plot the median of the predictions as a line and the 90% credible interval as a shaded area:



The dashed line shows the two-hour marathon pace, which is 13.1 miles per hour. Visually we can estimate that the prediction line hits the target pace between 2030 and 2040.

To make this more precise, we can use interpolation to see when the predictions cross the finish line. SciPy provides `interp1d`, which does linear interpolation by default.

```
from scipy.interpolate import interp1d

future = np.array([interp1d(high, xs)(13.1),
                   interp1d(median, xs)(13.1),
                   interp1d(low, xs)(13.1)])
```

The median prediction is 2036, with a 90% credible interval from 2032 to 2043. So there is about a 5% chance we'll see a two-hour marathon before 2032.

Summary

This chapter introduces Bayesian regression, which is based on the same model as least squares regression; the difference is that it produces a posterior distribution for the parameters rather than point estimates.

In the first example, we looked at changes in snowfall in Norfolk County, Massachusetts, and concluded that we get more snowfall now than when I was young, contrary to my expectation.

In the second example, we looked at the progression of world record pace for the men's marathon, computed the joint posterior distribution of the regression parameters, and used it to generate predictions for the next 20 years.

These examples have three parameters, so it takes a little longer to compute the likelihood of the data. With more than three parameters, it becomes impractical to use grid algorithms.

In the next few chapters, we'll explore other algorithms that reduce the amount of computation we need to do a Bayesian update, which makes it possible to use models with more parameters.

But first, you might want to work on these exercises.

Exercises

Exercise 17-1.

I am under the impression that it is warmer around here than it used to be. In this exercise, you can put my conjecture to the test.

We'll use the same dataset we used to model snowfall; it also includes daily low and high temperatures in Norfolk County, Massachusetts, during my lifetime. The details are in the notebook for this chapter.

1. Use `statsmodels` to generate point estimates for the regression parameters.
2. Choose priors for `slope`, `intercept`, and `sigma` based on these estimates, and use `make_joint3` to make a joint prior distribution.
3. Compute the likelihood of the data and compute the posterior distribution of the parameters.
4. Extract the posterior distribution of `slope`. How confident are we that temperature is increasing?
5. Draw a sample of parameters from the posterior distribution and use it to generate predictions up to 2067.
6. Plot the median of the predictions and a 90% credible interval along with the observed data.

Does the model fit the data well? How much do we expect annual average temperatures to increase over my (expected) lifetime?

Conjugate Priors

In the previous chapters we have used grid approximations to solve a variety of problems. One of my goals has been to show that this approach is sufficient to solve many real-world problems. And I think it's a good place to start because it shows clearly how the methods work.

However, as we saw in the previous chapter, grid methods will only get you so far. As we increase the number of parameters, the number of points in the grid grows (literally) exponentially. With more than 3-4 parameters, grid methods become impractical.

So, in the remaining three chapters, I will present three alternatives:

1. In this chapter, we'll use **conjugate priors** to speed up some of the computations we've already done.
2. In the next chapter, I'll present Markov chain Monte Carlo (MCMC) methods, which can solve problems with tens of parameters, or even hundreds, in a reasonable amount of time.
3. And in the last chapter, we'll use Approximate Bayesian Computation (ABC) for problems that are hard to model with simple distributions.

We'll start with the World Cup Problem.

The World Cup Problem Revisited

In [Chapter 8](#), we solved the World Cup Problem using a Poisson process to model goals in a soccer game as random events that are equally likely to occur at any point during a game.

We used a gamma distribution to represent the prior distribution of λ , the goal-scoring rate. And we used a Poisson distribution to compute the probability of k , the number of goals scored.

Here's a gamma object that represents the prior distribution:

```
from scipy.stats import gamma

alpha = 1.4
dist = gamma(alpha)
```

And here's a grid approximation:

```
import numpy as np
from utils import pmf_from_dist

lams = np.linspace(0, 10, 101)
prior = pmf_from_dist(dist, lams)
```

Here's the likelihood of scoring 4 goals for each possible value of λ :

```
from scipy.stats import poisson

k = 4
likelihood = poisson(lams).pmf(k)
```

And here's the update:

```
posterior = prior * likelihood
posterior.normalize()

0.05015532557804499
```

So far, this should be familiar. Now we'll solve the same problem using the conjugate prior.

The Conjugate Prior

In “[The Gamma Distribution](#)” on page 101, I presented three reasons to use a gamma distribution for the prior and said there was a fourth reason I would reveal later. Well, now is the time.

The other reason I chose the gamma distribution is that it is the “conjugate prior” of the Poisson distribution, so-called because the two distributions are connected or coupled, which is what “conjugate” means.

In the next section I'll explain *how* they are connected, but first I'll show you the consequence of this connection, which is that there is a remarkably simple way to compute the posterior distribution.

However, to demonstrate it, we have to switch from the one-parameter version of the gamma distribution to the two-parameter version. Since the first parameter is called `alpha`, you might guess that the second parameter is called `beta`.

The following function takes `alpha` and `beta` and makes an object that represents a gamma distribution with those parameters:

```
def make_gamma_dist(alpha, beta):
    """Makes a gamma object."""
    dist = gamma(alpha, scale=1/beta)
    dist.alpha = alpha
    dist.beta = beta
    return dist
```

Here's the prior distribution with `alpha=1.4` again and `beta=1`:

```
alpha = 1.4
beta = 1

prior_gamma = make_gamma_dist(alpha, beta)
prior_gamma.mean()

1.4
```

Now I claim without proof that we can do a Bayesian update with `k` goals just by making a gamma distribution with parameters `alpha+k` and `beta+1`:

```
def update_gamma(prior, data):
    """Update a gamma prior."""
    k, t = data
    alpha = prior.alpha + k
    beta = prior.beta + t
    return make_gamma_dist(alpha, beta)
```

Here's how we update it with `k=4` goals in `t=1` game:

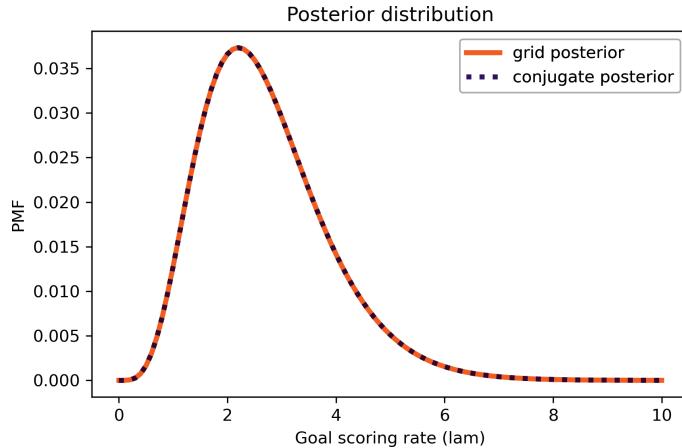
```
data = 4, 1
posterior_gamma = update_gamma(prior_gamma, data)
```

After all the work we did with the grid, it might seem absurd that we can do a Bayesian update by adding two pairs of numbers. So let's confirm that it works.

I'll make a `Pmf` with a discrete approximation of the posterior distribution:

```
posterior_conjugate = pmf_from_dist(posterior_gamma, lams)
```

The following figure shows the result along with the posterior we computed using the grid algorithm:



They are the same other than small differences due to floating-point approximations.

What the Actual?

To understand how that works, we'll write the PDF of the gamma prior and the PMF of the Poisson likelihood, then multiply them together, because that's what the Bayesian update does. We'll see that the result is a gamma distribution, and we'll derive its parameters.

Here's the PDF of the gamma prior, which is the probability density for each value of λ , given parameters α and β :

$$\lambda^{\alpha-1} e^{-\lambda\beta}$$

I have omitted the normalizing factor; since we are planning to normalize the posterior distribution anyway, we don't really need it.

Now suppose a team scores k goals in t games. The probability of this data is given by the PMF of the Poisson distribution, which is a function of k with λ and t as parameters:

$$\lambda^k e^{-\lambda t}$$

Again, I have omitted the normalizing factor, which makes it clearer that the gamma and Poisson distributions have the same functional form. When we multiply them together, we can pair up the factors and add up the exponents. The result is the unnormalized posterior distribution,

$$\lambda^\alpha - 1 + k e^{-\lambda(\beta + t)}$$

which we can recognize as an unnormalized gamma distribution with parameters $\alpha + k$ and $\beta + t$.

This derivation provides insight into what the parameters of the posterior distribution mean: α reflects the number of events that have occurred; β reflects the elapsed time.

Binomial Likelihood

As a second example, let's look again at the Euro Problem. When we solved it with a grid algorithm, we started with a uniform prior:

```
from utils import make_uniform

xs = np.linspace(0, 1, 101)
uniform = make_uniform(xs, 'uniform')
```

We used the binomial distribution to compute the likelihood of the data, which was 140 heads out of 250 attempts:

```
from scipy.stats import binom

k, n = 140, 250
xs = uniform.qs
likelihood = binom.pmf(k, n, xs)
```

Then we computed the posterior distribution in the usual way:

```
posterior = uniform * likelihood
posterior.normalize()
```

We can solve this problem more efficiently using the conjugate prior of the binomial distribution, which is the beta distribution.

The beta distribution is bounded between 0 and 1, so it works well for representing the distribution of a probability like x . It has two parameters, called `alpha` and `beta`, that determine the shape of the distribution.

SciPy provides an object called `beta` that represents a beta distribution. The following function takes `alpha` and `beta` and returns a new `beta` object:

```
import scipy.stats

def make_beta(alpha, beta):
    """Makes a beta object."""
    dist = scipy.stats.beta(alpha, beta)
    dist.alpha = alpha
    dist.beta = beta
    return dist
```

It turns out that the uniform distribution, which we used as a prior, is the beta distribution with parameters `alpha=1` and `beta=1`. So we can make a `beta` object that represents a uniform distribution, like this:

```
alpha = 1
beta = 1

prior_beta = make_beta(alpha, beta)
```

Now let's figure out how to do the update. As in the previous example, we'll write the PDF of the prior distribution and the PMF of the likelihood function, and multiply them together. We'll see that the product has the same form as the prior, and we'll derive its parameters.

Here is the PDF of the beta distribution, which is a function of x with α and β as parameters:

$$x^{\alpha-1}(1-x)^{\beta-1}$$

Again, I have omitted the normalizing factor, which we don't need because we are going to normalize the distribution after the update.

And here's the PMF of the binomial distribution, which is a function of k with n and x as parameters:

$$x^k(1-x)^{n-k}$$

Again, I have omitted the normalizing factor. Now when we multiply the beta prior and the binomial likelihood, the result is

$$x^{\alpha-1+k}(1-x)^{\beta-1+n-k}$$

which we recognize as an unnormalized beta distribution with parameters $\alpha + k$ and $\beta + n - k$.

So if we observe k successes in n trials, we can do the update by making a beta distribution with parameters `alpha+k` and `beta+n-k`. That's what this function does:

```
def update_beta(prior, data):
    """Update a beta distribution."""
    k, n = data
    alpha = prior.alpha + k
    beta = prior.beta + n - k
    return make_beta(alpha, beta)
```

Again, the conjugate prior gives us insight into the meaning of the parameters; α is related to the number of observed successes; β is related to the number of failures.

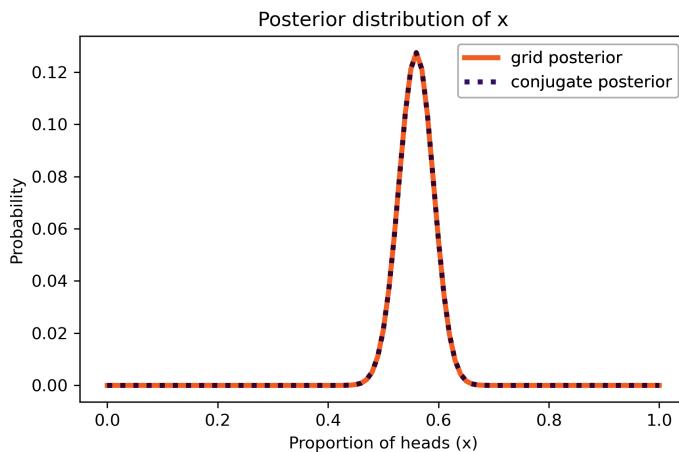
Here's how we do the update with the observed data:

```
data = 140, 250
posterior_beta = update_beta(prior_beta, data)
```

To confirm that it works, I'll evaluate the posterior distribution for the possible values of x s and put the results in a PMF:

```
posterior_conjugate = pmf_from_dist(posterior_beta, xs)
```

And we can compare the posterior distribution we just computed with the results from the grid algorithm:



They are the same other than small differences due to floating-point approximations.

The examples so far are problems we have already solved, so let's try something new.

Lions and Tigers and Bears

Suppose we visit a wild animal preserve where we know that the only animals are lions and tigers and bears, but we don't know how many of each there are. During the tour, we see three lions, two tigers, and one bear. Assuming that every animal had an equal chance to appear in our sample, what is the probability that the next animal we see is a bear?

To answer this question, we'll use the data to estimate the prevalence of each species, that is, what fraction of the animals belong to each species. If we know the prevalences, we can use the multinomial distribution to compute the probability of the data.

For example, suppose we know that the fraction of lions, tigers, and bears is 0.4, 0.3, and 0.3, respectively.

In that case the probability of the data is:

```
from scipy.stats import multinomial

data = 3, 2, 1
n = np.sum(data)
ps = 0.4, 0.3, 0.3

multinomial.pmf(data, n, ps)

0.10368
```

Now, we could choose a prior for the prevalences and do a Bayesian update using the multinomial distribution to compute the probability of the data.

But there's an easier way, because the multinomial distribution has a conjugate prior: the Dirichlet distribution.

The Dirichlet Distribution

The Dirichlet distribution is a multivariate distribution, like the multivariate normal distribution we used in “[Multivariate Normal Distribution](#)” on page 170 to describe the distribution of penguin measurements.

In that example, the quantities in the distribution are pairs of flipper length and culmen length, and the parameters of the distribution are a vector of means and a matrix of covariances.

In a Dirichlet distribution, the quantities are vectors of probabilities, \mathbf{x} , and the parameter is a vector, α .

An example will make that clearer. SciPy provides a `dirichlet` object that represents a Dirichlet distribution. Here's an instance with $\alpha = 1, 2, 3$:

```
from scipy.stats import dirichlet

alpha = 1, 2, 3
dist = dirichlet(alpha)
```

Since we provided three parameters, the result is a distribution of three variables. Suppose we draw a random value from this distribution, like this:

```
dist.rvs()

array([[0.46414019, 0.16853117, 0.36732863]])
```

The result is an array of three values. They are bounded between 0 and 1, and they always add up to 1, so they can be interpreted as the probabilities of a set of outcomes that are mutually exclusive and collectively exhaustive.

Let's see what the distributions of these values look like. I'll draw 1,000 random vectors from this distribution, like this:

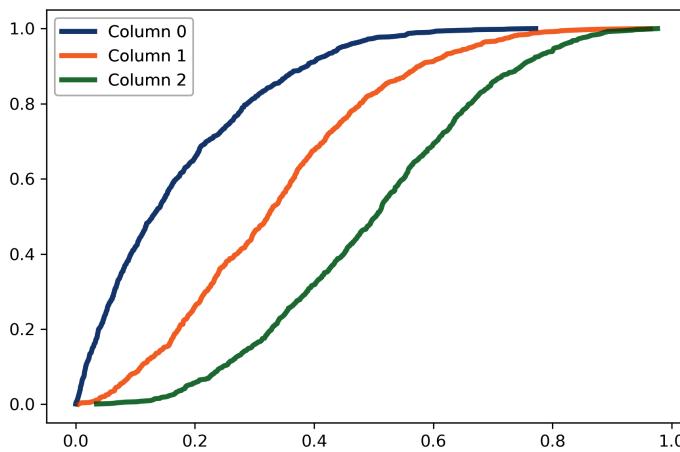
```
sample = dist.rvs(1000)
```

The result is an array with 1,000 rows and three columns. I'll compute the Cdf of the values in each column:

```
from empiricaldist import Cdf

cdfs = [Cdf.from_seq(col)
        for col in sample.transpose()]
```

The result is a list of Cdf objects that represent the marginal distributions of the three variables. Here's what they look like:



Column 0, which corresponds to the lowest parameter, contains the lowest probabilities. Column 2, which corresponds to the highest parameter, contains the highest probabilities.

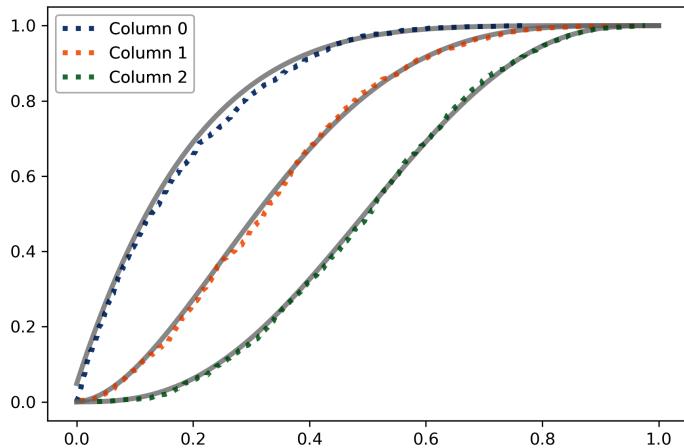
As it turns out, these marginal distributions are beta distributions. The following function takes a sequence of parameters, `alpha`, and computes the marginal distribution of variable `i`:

```
def marginal_beta(alpha, i):
    """Compute the ith marginal of a Dirichlet distribution."""
    total = np.sum(alpha)
    return make_beta(alpha[i], total-alpha[i])
```

We can use it to compute the marginal distribution for the three variables:

```
marginals = [marginal_beta(alpha, i)
              for i in range(len(alpha))]
```

The following plot shows the CDF of these distributions as gray lines and compares them to the CDFs of the samples:



This confirms that the marginals of the Dirichlet distribution are beta distributions. And that's useful because the Dirichlet distribution is the conjugate prior for the multinomial likelihood function.

If the prior distribution is Dirichlet with parameter vector `alpha` and the data is a vector of observations, `data`, the posterior distribution is Dirichlet with parameter vector `alpha + data`.

As an exercise at the end of this chapter, you can use this method to solve the Lions and Tigers and Bears problem.

Summary

After reading this chapter, if you feel like you've been tricked, I understand. It turns out that many of the problems in this book can be solved with just a few arithmetic operations. So why did we go to all the trouble of using grid algorithms?

Sadly, there are only a few problems we can solve with conjugate priors; in fact, this chapter includes most of the ones that are useful in practice.

For the vast majority of problems, there is no conjugate prior and no shortcut to compute the posterior distribution. That's why we need grid algorithms and the methods in the next two chapters, Approximate Bayesian Computation (ABC) and Markov chain Monte Carlo methods (MCMC).

Exercises

Exercise 18-1.

In the second version of the World Cup Problem, the data we use for the update is not the number of goals in a game, but the time until the first goal. So the probability of the data is given by the exponential distribution rather than the Poisson distribution.

But it turns out that the gamma distribution is *also* the conjugate prior of the exponential distribution, so there is a simple way to compute this update, too. The PDF of the exponential distribution is a function of t with λ as a parameter:

$$\lambda e^{-\lambda t}$$

Multiply the PDF of the gamma prior by this likelihood, confirm that the result is an unnormalized gamma distribution, and see if you can derive its parameters.

Write a few lines of code to update `prior_gamma` with the data from this version of the problem, which was a first goal after 11 minutes and a second goal after an additional 12 minutes.

Exercise 18-2.

For problems like the Euro Problem where the likelihood function is binomial, we can do a Bayesian update with just a few arithmetic operations, but only if the prior is a beta distribution.

If we want a uniform prior, we can use a beta distribution with `alpha=1` and `beta=1`. But what can we do if the prior distribution we want is not a beta distribution? For example, in “[Triangle Prior](#)” on page 49 we also solved the Euro Problem with a triangle prior, which is not a beta distribution.

In these cases, we can often find a beta distribution that is a good-enough approximation for the prior we want. See if you can find a beta distribution that fits the triangle prior, then update it using `update_beta`.

Use `pmf_from_dist` to make a `PMF` that approximates the posterior distribution and compare it to the posterior we just computed using a grid algorithm. How big is the largest difference between them?

Exercise 18-3.

3Blue1Brown is a YouTube channel about math; if you are not already aware of it, I recommend it highly. In [this video](#) the narrator presents this problem:

You are buying a product online and you see three sellers offering the same product at the same price. One of them has a 100% positive rating, but with only 10 reviews. Another has a 96% positive rating with 50 total reviews. And yet another has a 93% positive rating, but with 200 total reviews.

Which one should you buy from?

Let's think about how to model this scenario. Suppose each seller has some unknown probability, x , of providing satisfactory service and getting a positive rating, and we want to choose the seller with the highest value of x .

This is not the only model for this scenario, and it is not necessarily the best. An alternative would be something like item response theory, where sellers have varying ability to provide satisfactory service and customers have varying difficulty of being satisfied.

But the first model has the virtue of simplicity, so let's see where it gets us.

1. As a prior, I suggest a beta distribution with `alpha=8` and `beta=2`. What does this prior look like and what does it imply about sellers?
2. Use the data to update the prior for the three sellers and plot the posterior distributions. Which seller has the highest posterior mean?
3. How confident should we be about our choice? That is, what is the probability that the seller with the highest posterior mean actually has the highest value of x ?
4. Consider a beta prior with `alpha=0.7` and `beta=0.5`. What does this prior look like and what does it imply about sellers?
5. Run the analysis again with this prior and see what effect it has on the results.

Exercise 18-4.

Use a Dirichlet prior with parameter vector `alpha = [1, 1, 1]` to solve the Lions and Tigers and Bears problem:

Suppose we visit a wild animal preserve where we know that the only animals are lions and tigers and bears, but we don't know how many of each there are.

During the tour, we see three lions, two tigers, and one bear. Assuming that every animal had an equal chance to appear in our sample, estimate the prevalence of each species.

What is the probability that the next animal we see is a bear?

For most of this book we've been using grid methods to approximate posterior distributions. For models with one or two parameters, grid algorithms are fast and the results are precise enough for most practical purposes. With three parameters, they start to be slow, and with more than three they are usually not practical.

In the previous chapter we saw that we can solve some problems using conjugate priors. But the problems we can solve this way tend to be the same ones we can solve with grid algorithms.

For problems with more than a few parameters, the most powerful tool we have is MCMC, which stands for "Markov chain Monte Carlo". In this context, "Monte Carlo" refers to methods that generate random samples from a distribution. Unlike grid methods, MCMC methods don't try to compute the posterior distribution; they sample from it instead.

It might seem strange that you can generate a sample without ever computing the distribution, but that's the magic of MCMC.

To demonstrate, we'll start by solving the World Cup Problem. Yes, again.

The World Cup Problem

In [Chapter 8](#) we modeled goal scoring in football (soccer) as a Poisson process characterized by a goal-scoring rate, denoted λ .

We used a gamma distribution to represent the prior distribution of λ , then we used the outcome of the game to compute the posterior distribution for both teams.

To answer the first question, we used the posterior distributions to compute the "probability of superiority" for France.

To answer the second question, we computed the posterior predictive distributions for each team, that is, the distribution of goals we expect in a rematch.

In this chapter we'll solve this problem again using PyMC3, which is a library that provide implementations of several MCMC methods. But we'll start by reviewing the grid approximation of the prior and the prior predictive distribution.

Grid Approximation

As we did in “[The Gamma Distribution](#)” on page 101 we'll use a gamma distribution with parameter $\alpha = 1.4$ to represent the prior:

```
from scipy.stats import gamma

alpha = 1.4
prior_dist = gamma(alpha)
```

I'll use `linspace` to generate possible values for λ , and `pmf_from_dist` to compute a discrete approximation of the prior:

```
import numpy as np
from utils import pmf_from_dist

lams = np.linspace(0, 10, 101)
prior_pmf = pmf_from_dist(prior_dist, lams)
```

We can use the Poisson distribution to compute the likelihood of the data; as an example, we'll use 4 goals:

```
from scipy.stats import poisson

data = 4
likelihood = poisson.pmf(data, lams)
```

Now we can do the update in the usual way:

```
posterior = prior_pmf * likelihood
posterior.normalize()

0.05015532557804499
```

Soon we will solve the same problem with PyMC3, but first it will be useful to introduce something new: the prior predictive distribution.

Prior Predictive Distribution

We have seen the posterior predictive distribution in previous chapters; the prior predictive distribution is similar except that (as you might have guessed) it is based on the prior.

To estimate the prior predictive distribution, we'll start by drawing a sample from the prior:

```
sample_prior = prior_dist.rvs(1000)
```

The result is an array of possible values for the goal-scoring rate, λ . For each value in `sample_prior`, I'll generate one value from a Poisson distribution:

```
from scipy.stats import poisson

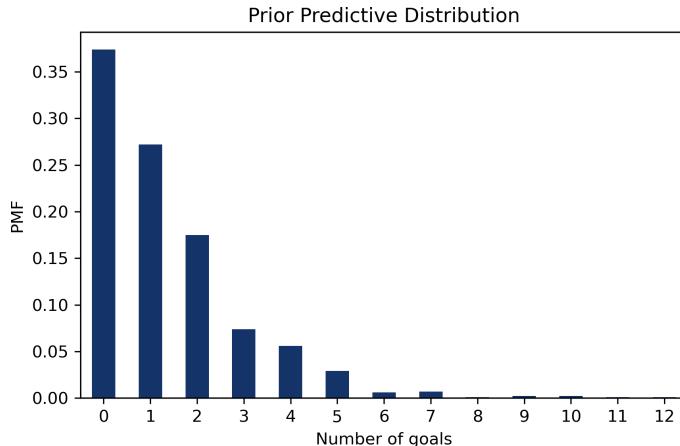
sample_prior_pred = poisson.rvs(sample_prior)
```

`sample_prior_pred` is a sample from the prior predictive distribution. To see what it looks like, we'll compute the PMF of the sample:

```
from empiricaldist import Pmf

pmf_prior_pred = Pmf.from_seq(sample_prior_pred)
```

And here's what it looks like:



One reason to compute the prior predictive distribution is to check whether our model of the system seems reasonable. In this case, the distribution of goals seems consistent with what we know about World Cup football.

But in this chapter we have another reason: computing the prior predictive distribution is a first step toward using MCMC.

Introducing PyMC3

PyMC3 is a Python library that provides several MCMC methods. To use PyMC3, we have to specify a model of the process that generates the data. In this example, the model has two steps:

- First we draw a goal-scoring rate from the prior distribution,
- Then we draw a number of goals from a Poisson distribution.

Here's how we specify this model in PyMC3:

```
import pymc3 as pm

with pm.Model() as model:
    lam = pm.Gamma('lam', alpha=1.4, beta=1.0)
    goals = pm.Poisson('goals', lam)
```

After importing `pymc3`, we create a `Model` object named `model`.

If you are not familiar with the `with` statement in Python, it is a way to associate a block of statements with an object. In this example, the two indented statements are associated with the new `Model` object. As a result, when we create the distribution objects, `Gamma` and `Poisson`, they are added to the `Model`.

Inside the `with` statement:

- The first line creates the prior, which is a gamma distribution with the given parameters.
- The second line creates the prior predictive, which is a Poisson distribution with the parameter `lam`.

The first parameter of `Gamma` and `Poisson` is a string variable name.

Sampling the Prior

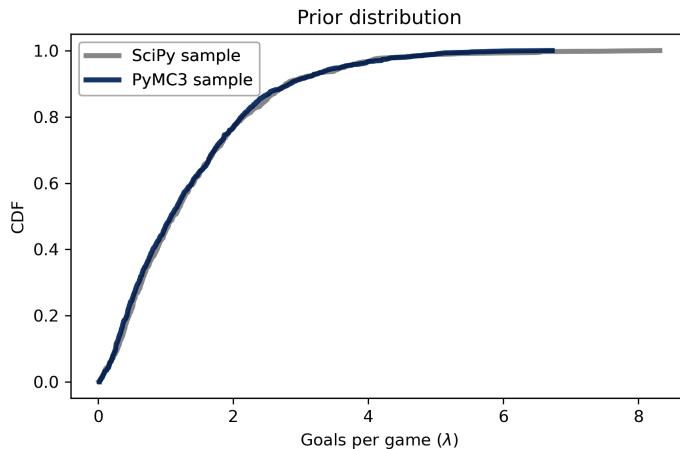
PyMC3 provides a function that generates samples from the prior and prior predictive distributions. We can use a `with` statement to run this function in the context of the model:

```
with model:
    trace = pm.sample_prior_predictive(1000)
```

The result is a dictionary-like object that maps from the variables, `lam` and `goals`, to the samples. We can extract the sample of `lam` like this:

```
sample_prior_pymc = trace['lam']
sample_prior_pymc.shape
(1000,)
```

The following figure compares the CDF of this sample to the CDF of the sample we generated using the `gamma` object from SciPy:



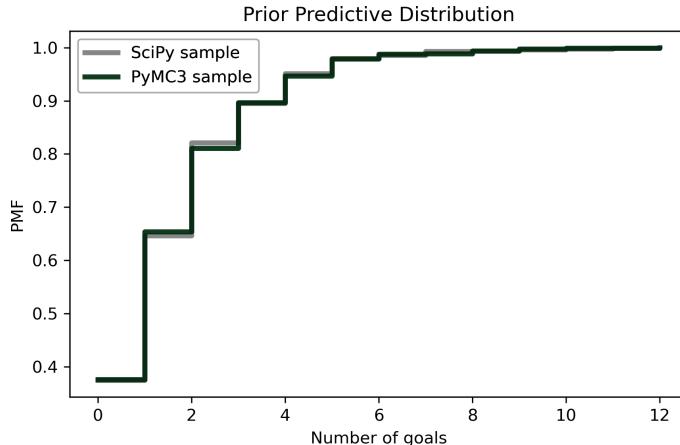
The results are similar, which confirms that the specification of the model is correct and the sampler works as advertised.

From the trace we can also extract `goals`, which is a sample from the prior predictive distribution:

```
sample_prior_pred_pymc = trace['goals']
sample_prior_pred_pymc.shape
(1000,)
```

And we can compare it to the sample we generated using the `poisson` object from SciPy.

Because the quantities in the posterior predictive distribution are discrete (number of goals) I'll plot the CDFs as step functions:



Again, the results are similar, so we have some confidence we are using PyMC3 right.

When Do We Get to Inference?

Finally, we are ready for actual inference. We just have to make one small change. Here is the model we used to generate the prior predictive distribution:

```
with pm.Model() as model:  
    lam = pm.Gamma('lam', alpha=1.4, beta=1.0)  
    goals = pm.Poisson('goals', lam)
```

And here is the model we'll use to compute the posterior distribution:

```
with pm.Model() as model2:  
    lam = pm.Gamma('lam', alpha=1.4, beta=1.0)  
    goals = pm.Poisson('goals', lam, observed=4)
```

The difference is that we mark `goals` as `observed` and provide the observed data, 4.

And instead of calling `sample_prior_predictive`, we'll call `sample`, which is understood to sample from the posterior distribution of `lam`:

```
options = dict(return_inferencedata=False)  
  
with model2:  
    trace2 = pm.sample(500, **options)
```

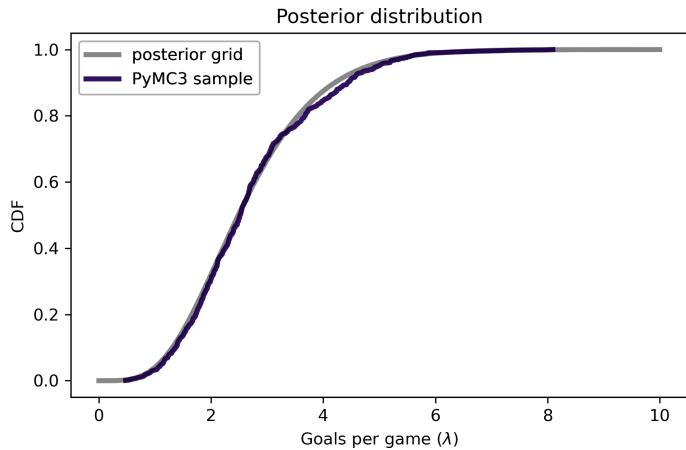
Although the specification of these models is similar, the sampling process is very different. I won't go into the details of how PyMC3 works, but here are a few things you should be aware of:

- Depending on the model, PyMC3 uses one of several MCMC methods; in this example, it uses the **No U-Turn Sampler** (NUTS), which is one of the most efficient and reliable methods we have.
- When the sampler starts, the first values it generates are usually not a representative sample from the posterior distribution, so these values are discarded. This process is called "tuning".
- Instead of using a single Markov chain, PyMC3 uses multiple chains. Then we can compare results from multiple chains to make sure they are consistent.

Although we asked for a sample of 500, PyMC3 generated two samples of 1,000, discarded half of each, and returned the remaining 1,000. From `trace2` we can extract a sample from the posterior distribution, like this:

```
sample_post_pymc = trace2['lam']
```

And we can compare the CDF of this sample to the posterior we computed by grid approximation:



The results from PyMC3 are consistent with the results from the grid approximation.

Posterior Predictive Distribution

Finally, to sample from the posterior predictive distribution, we can use `sample_posterior_predictive`:

```
with model2:  
    post_pred = pm.sample_posterior_predictive(trace2)
```

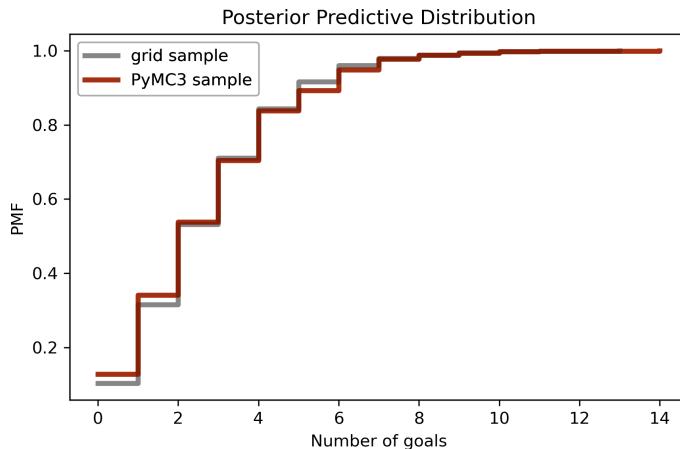
The result is a dictionary that contains a sample of `goals`:

```
sample_post_pred_pymc = post_pred['goals']
```

I'll also generate a sample from the posterior distribution we computed by grid approximation:

```
sample_post = posterior.sample(1000)  
sample_post_pred = poisson(sample_post).rvs()
```

And we can compare the two samples:



Again, the results are consistent. So we've established that we can compute the same results using a grid approximation or PyMC3.

But it might not be clear why. In this example, the grid algorithm requires less computation than MCMC, and the result is a pretty good approximation of the posterior distribution, rather than a sample.

However, this is a simple model with just one parameter. In fact, we could have solved it with even less computation, using a conjugate prior. The power of PyMC3 will be clearer with a more complex model.

Happiness

Recently I read “[Happiness and Life Satisfaction](#)” by Esteban Ortiz-Ospina and Max Roser, which discusses (among many other things) the relationship between income and happiness, both between countries, within countries, and over time. It cites the “[World Happiness Report](#)”, which includes [results of a multiple regression analysis](#) that explores the relationship between happiness and six potentially predictive factors:

- Income as represented by per capita GDP
- Social support
- Healthy life expectancy at birth
- Freedom to make life choices
- Generosity
- Perceptions of corruption

The dependent variable is the national average of responses to the “Cantril ladder question” used by the [Gallup World Poll](#):

Please imagine a ladder with steps numbered from zero at the bottom to 10 at the top. The top of the ladder represents the best possible life for you and the bottom of the ladder represents the worst possible life for you. On which step of the ladder would you say you personally feel you stand at this time?

I'll refer to the responses as “happiness”, but it might be more precise to think of them as a measure of satisfaction with quality of life.

In the next few sections we'll replicate the analysis in this report using Bayesian regression.

We can use pandas to read the data into a `DataFrame`:

```
import pandas as pd

filename = 'WHR20_DataForFigure2.1.xls'
df = pd.read_excel(filename)
```

The `DataFrame` has one row for each of 153 countries and one column for each of 20 variables.

The column called '`Ladder score`' contains the measurements of happiness we will try to predict.

```
score = df['Ladder score']
```

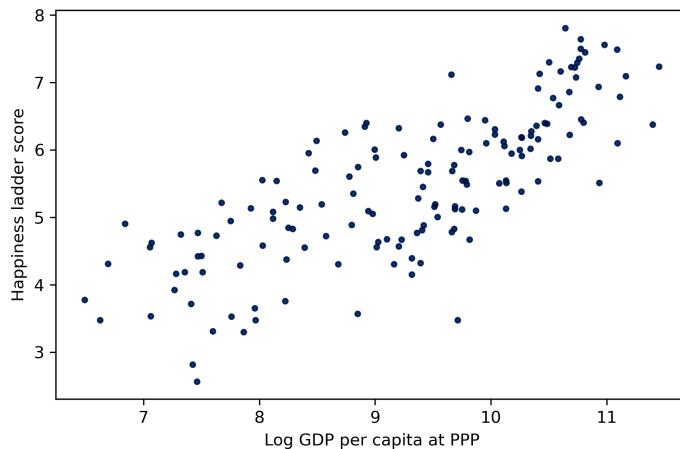
Simple Regression

To get started, let's look at the relationship between happiness and income as represented by gross domestic product (GDP) per person.

The column named '`Logged GDP per capita`' represents the natural logarithm of GDP for each country, divided by population, corrected for [purchasing power parity](#) (PPP):

```
log_gdp = df['Logged GDP per capita']
```

The following figure is a scatter plot of `score` versus `log_gdp`, with one marker for each country:



It's clear that there is a relationship between these variables: people in countries with higher GDP generally report higher levels of happiness.

We can use `linregress` from SciPy to compute a simple regression of these variables:

```
from scipy.stats import linregress
result = linregress(log_gdp, score)
```

And here are the results:

Slope	0.717738
Intercept	-1.198646

The estimated slope is about 0.72, which suggests that an increase of one unit in log-GDP, which is a factor of $e \approx 2.7$ in GDP, is associated with an increase of 0.72 units on the happiness ladder.

Now let's estimate the same parameters using PyMC3. We'll use the same regression model as in “[Regression Model](#)” on page 243,

$$y = ax + b + \epsilon$$

where y is the dependent variable (ladder score), x is the predictive variable (log GDP) and ϵ is a series of values from a normal distribution with standard deviation σ .

a and b are the slope and intercept of the regression line. They are unknown parameters, so we will use the data to estimate them.

The following is the PyMC3 specification of this model:

```
x_data = log_gdp
y_data = score

with pm.Model() as model3:
    a = pm.Uniform('a', 0, 4)
    b = pm.Uniform('b', -4, 4)
    sigma = pm.Uniform('sigma', 0, 2)

    y_est = a * x_data + b
    y = pm.Normal('y',
                  mu=y_est, sd=sigma,
                  observed=y_data)
```

The prior distributions for the parameters `a`, `b`, and `sigma` are uniform with ranges that are wide enough to cover the posterior distributions.

`y_est` is the estimated value of the dependent variable, based on the regression equation. And `y` is a normal distribution with mean `y_est` and standard deviation `sigma`.

Notice how the data are included in the model:

- The values of the predictive variable, `x_data`, are used to compute `y_est`.
- The values of the dependent variable, `y_data`, are provided as the observed values of `y`.

Now we can use this model to generate a sample from the posterior distribution:

```
with model3:
    trace3 = pm.sample(500, **options)
```

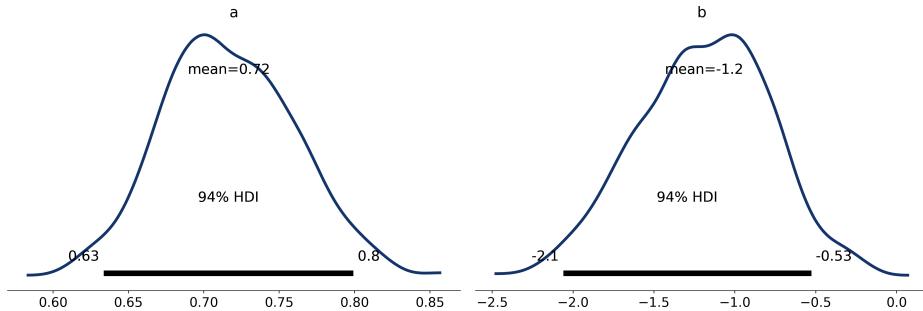
When you run the sampler, you might get warning messages about “divergences” and the “acceptance probability”. You can ignore them for now.

The result is an object that contains samples from the joint posterior distribution of `a`, `b`, and `sigma`.

ArviZ provides `plot_posterior`, which we can use to plot the posterior distributions of the parameters. Here are the posterior distributions of slope, `a`, and intercept, `b`:

```
import arviz as az

with model3:
    az.plot_posterior(trace3, var_names=['a', 'b']);
```



The graphs show the distributions of the samples, estimated by KDE, and 94% credible intervals. In the figure, “HDI” stands for [“highest-density interval”](#).

The means of these samples are consistent with the parameters we estimated with `linregress`.

The simple regression model has only three parameters, so we could have used a grid algorithm. But the regression model in the happiness report has six predictive variables, so it has eight parameters in total, including the intercept and `sigma`.

It is not practical to compute a grid approximation for a model with eight parameters. Even a coarse grid, with 20 points along each dimension, would have more than 25 billion points. And with 153 countries, we would have to compute almost 4 trillion likelihoods.

But PyMC3 can handle a model with eight parameters comfortably, as we’ll see in the next section.

Multiple Regression

Before we implement the multiple regression model, I’ll select the columns we need from the `DataFrame`:

```
columns = ['Ladder score',
           'Logged GDP per capita',
           'Social support',
           'Healthy life expectancy',
           'Freedom to make life choices',
           'Generosity',
           'Perceptions of corruption']

subset = df[columns]
```

The predictive variables have different units: log-GDP is in log-dollars, life expectancy is in years, and the other variables are on arbitrary scales. To make these factors comparable, I'll standardize the data so that each variable has mean 0 and standard deviation 1.

```
standardized = (subset - subset.mean()) / subset.std()
```

Now let's build the model. I'll extract the dependent variable:

```
y_data = standardized['Ladder score']
```

And the dependent variables:

```
x1 = standardized[columns[1]]
x2 = standardized[columns[2]]
x3 = standardized[columns[3]]
x4 = standardized[columns[4]]
x5 = standardized[columns[5]]
x6 = standardized[columns[6]]
```

And here's the model. b_0 is the intercept; b_1 through b_6 are the parameters associated with the predictive variables:

```
with pm.Model() as model4:
    b0 = pm.Uniform('b0', -4, 4)
    b1 = pm.Uniform('b1', -4, 4)
    b2 = pm.Uniform('b2', -4, 4)
    b3 = pm.Uniform('b3', -4, 4)
    b4 = pm.Uniform('b4', -4, 4)
    b5 = pm.Uniform('b5', -4, 4)
    b6 = pm.Uniform('b6', -4, 4)
    sigma = pm.Uniform('sigma', 0, 2)

    y_est = b0 + b1*x1 + b2*x2 + b3*x3 + b4*x4 + b5*x5 + b6*x6
    y = pm.Normal('y',
                  mu=y_est,
                  sd=sigma,
                  observed=y_data)
```

We could express this model more concisely using a vector of predictive variables and a vector of parameters, but I decided to keep it simple.

Now we can sample from the joint posterior distribution:

```
with model4:
    trace4 = pm.sample(500, **options)
```

From `trace4` we can extract samples from the posterior distributions of the parameters and compute their means:

```
param_names = ['b1', 'b3', 'b5', 'b4', 'b5', 'b6']

means = [trace4[name].mean()
         for name in param_names]
```

We can also compute 94% credible intervals (between the 3rd and 97th percentiles):

```
def credible_interval(sample):
    """Compute 94% credible interval."""
    ci = np.percentile(sample, [3, 97])
    return np.round(ci, 3)

cis = [credible_interval(trace4[name])
       for name in param_names]
```

The following table summarizes the results:

	Posterior mean	94% CI
Logged GDP per capita	0.246	[0.077, 0.417]
Social support	0.224	[0.064, 0.384]
Healthy life expectancy	0.224	[0.064, 0.384]
Freedom to make life choices	0.190	[0.094, 0.291]
Generosity	0.055	[-0.032, 0.139]
Perceptions of corruption	-0.098	[-0.194, -0.002]

It looks like GDP has the strongest association with happiness (or satisfaction), followed by social support, life expectancy, and freedom.

After controlling for those other factors, the parameters of the other factors are substantially smaller, and since the CI for generosity includes 0, it is plausible that generosity is not substantially related to happiness, at least as they were measured in this study.

This example demonstrates the power of MCMC to handle models with more than a few parameters. But it does not really demonstrate the power of Bayesian regression.

If the goal of a regression model is to estimate parameters, there is no great advantage to Bayesian regression compared to conventional least squares regression.

Bayesian methods are more useful if we plan to use the posterior distribution of the parameters as part of a decision analysis process.

Summary

In this chapter we used PyMC3 to implement two models we've seen before: a Poisson model of goal-scoring in soccer and a simple regression model. Then we implemented a multiple regression model that would not have been possible to compute with a grid approximation.

MCMC is more powerful than grid methods, but that power comes with some disadvantages:

- MCMC algorithms are fiddly. The same model might behave well with some priors and less well with others. And the sampling process often produces warnings about tuning steps, divergences, “r-hat statistics”, acceptance rates, and effective samples. It takes some expertise to diagnose and correct these issues.
- I find it easier to develop models incrementally using grid algorithms, checking intermediate results along the way. With PyMC3, it is not as easy to be confident that you have specified a model correctly.

For these reasons, I recommend a model development process that starts with grid algorithms and resorts to MCMC if necessary. As we saw in the previous chapters, you can solve a lot of real-world problems with grid methods. But when you need MCMC, it is useful to have a grid algorithm to compare to (even if it is based on a simpler model).

All of the models in this book can be implemented in PyMC3, but some of them are easier to translate than others. In the exercises, you will have a chance to practice.

Exercises

Exercise 19-1.

As a warm-up, let’s use PyMC3 to solve the Euro Problem. Suppose we spin a coin 250 times and it comes up heads 140 times. What is the posterior distribution of x , the probability of heads?

For the prior, use a beta distribution with parameters $\alpha = 1$ and $\beta = 1$.

See [the PyMC3 documentation](#) for the list of continuous distributions.

Exercise 19-2.

Now let’s use PyMC3 to replicate the solution to the Grizzly Bear Problem in “[The Grizzly Bear Problem](#)” on page 207, which is based on the hypergeometric distribution.

I’ll present the problem with slightly different notation, to make it consistent with PyMC3.

Suppose that during the first session, $k=23$ bears are tagged. During the second session, $n=19$ bears are identified, of which $x=4$ had been tagged.

Estimate the posterior distribution of N , the number of bears in the environment.

For the prior, use a discrete uniform distribution from 50 to 500.

See [the PyMC3 documentation](#) for the list of discrete distributions.

Note: `HyperGeometric` was added to PyMC3 after version 3.8, so you might need to update your installation to do this exercise.

Exercise 19-3.

In “[The Weibull Distribution](#)” on page 191 we generated a sample from a Weibull distribution with $\lambda = 3$ and $k = 0.8$. Then we used the data to compute a grid approximation of the posterior distribution of those parameters.

Now let’s do the same with PyMC3.

For the priors, you can use uniform distributions as we did in [Chapter 14](#), or you could use `HalfNormal` distributions provided by PyMC3.

Note: The `Weibull` class in PyMC3 uses different parameters than SciPy. The parameter `alpha` in PyMC3 corresponds to k , and `beta` corresponds to λ .

```
data = [0.80497283, 2.11577082, 0.43308797, 0.10862644, 5.17334866,
        3.25745053, 3.05555883, 2.47401062, 0.05340806, 1.08386395]
```

Exercise 19-4.

In “[Improving Reading Ability](#)” on page 175 we used data from a reading test to estimate the parameters of a normal distribution.

Make a model that defines uniform prior distributions for `mu` and `sigma` and uses the data to estimate their posterior distributions.

Exercise 19-5.

In “[The Lincoln Index Problem](#)” on page 215 we used a grid algorithm to solve the Lincoln Index Problem as presented by John D. Cook:

Suppose you have a tester who finds 20 bugs in your program. You want to estimate how many bugs are really in the program. You know there are at least 20 bugs, and if you have supreme confidence in your tester, you may suppose there are around 20 bugs. But maybe your tester isn’t very good. Maybe there are hundreds of bugs. How can you have any idea how many bugs there are? There’s no way to know with one tester. But if you have two testers, you can get a good idea, even if you don’t know how skilled the testers are.

Suppose the first tester finds 20 bugs, the second finds 15, and they find 3 in common; use PyMC3 to estimate the number of bugs.

Note: This exercise is more difficult than some of the previous ones. One of the challenges is that the data includes k_{00} , which depends on N :

```
k00 = N - num_seen
```

So we have to construct the data as part of the model. To do that, we can use `pm.math.stack`, which makes an array:

```
data = pm.math.stack((k00, k01, k10, k11))
```

Finally, you might find it helpful to use `pm.Multinomial`.

CHAPTER 20

Approximate Bayesian Computation

This chapter introduces a method of last resort for the most complex problems, Approximate Bayesian Computation (ABC). I say it is a last resort because it usually requires more computation than other methods, so if you can solve a problem any other way, you should. However, for the examples in this chapter, ABC is not just easy to implement; it is also efficient.

The first example is my solution to a problem posed by a patient with a kidney tumor. I use data from a medical journal to model tumor growth, and use simulations to estimate the age of a tumor based on its size.

The second example is a model of cell counting, which has applications in biology, medicine, and zymurgy (beer-making). Given a cell count from a diluted sample, we estimate the concentration of cells.

Finally, as an exercise, you'll have a chance to work on a fun sock-counting problem.

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.

I contacted the author of the message to get 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). So I agree to help him answer his question.

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.

For my analysis I used data from a paper by [Zhang et al](#). They report growth rates in two forms:

- Volumetric doubling time, which is the time it would take for a tumor to double in size.
- Reciprocal doubling time (RDT), which is the number of doublings per year.

The next section shows how we work with these growth rates.

A Simple Growth Model

We'll start with a simple model of tumor growth based on two assumptions:

- Tumors grow with a constant doubling time, and
- They are roughly spherical in shape.

And I'll define two points in time:

- t_1 is when my correspondent retired.
- t_2 is when the tumor was detected.

The time between t_1 and t_2 was about 9.0 years. As an example, let's assume that the diameter of the tumor was 1 cm at t_1 , and estimate its size at t_2 .

I'll use the following function to compute the volume of a sphere with a given diameter:

```
import numpy as np

def calc_volume(diameter):
    """Converts a diameter to a volume."""
    factor = 4 * np.pi / 3
    return factor * (diameter/2.0)**3
```

Assuming that the tumor is spherical, we can compute its volume at t_1 :

```
d1 = 1
v1 = calc_volume(d1)
v1
0.5235987755982988
```

The median volume doubling time reported by Zhang et al. is 811 days, which corresponds to an RDT of 0.45 doublings per year:

```
median_doubling_time = 811
rdt = 365 / median_doubling_time
rdt

0.45006165228113443
```

We can compute the number of doublings that would have happened in the interval between t1 and t2:

```
interval = 9.0
doublings = interval * rdt
doublings

4.05055487053021
```

Given v1 and the number of doublings, we can compute the volume at t2:

```
v2 = v1 * 2**doublings
v2

8.676351488087187
```

The following function computes the diameter of a sphere with the given volume:

```
def calc_diameter(volume):
    """Converts a volume to a diameter."""
    factor = 3 / np.pi / 4
    return 2 * (factor * volume)**(1/3)
```

So we can compute the diameter of the tumor at t2:

```
d2 = calc_diameter(v2)
d2

2.5494480788327483
```

If the diameter of the tumor was 1 cm at t1, and it grew at the median rate, the diameter would be about 2.5 cm at t2.

This example demonstrates the growth model, but it doesn't answer the question my correspondent posed.

A More General Model

Given the size of a tumor at time of diagnosis, we would like to know the distribution of its age. To find it, we'll run simulations of tumor growth to get the distribution of size conditioned on age. Then we'll compute the distribution of age conditioned on size.

The simulation starts with a small tumor and runs these steps:

1. Choose a value from the distribution of growth rates.
2. Compute the size of the tumor at the end of an interval.
3. Repeat until the tumor exceeds the maximum relevant size.

So the first thing we need is the distribution of growth rates.

Using the figures in the paper by Zhange et al., I created an array, `rdt_sample`, that contains estimated values of RDT for the 53 patients in the study.

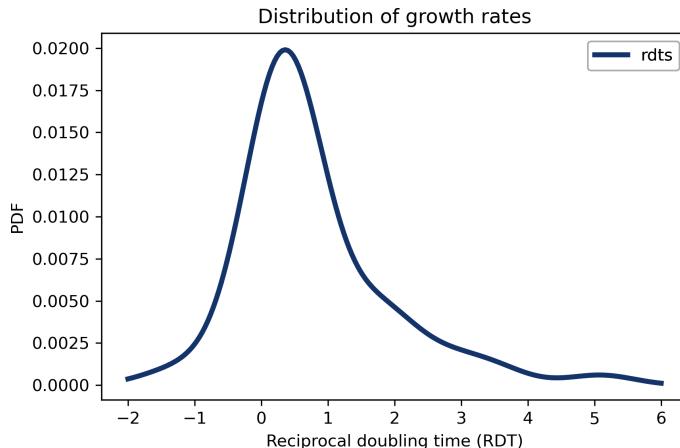
Again, RDT stands for “reciprocal doubling time”, which is in doublings per year. So if `rdt=1`, a tumor would double in volume in one year. If `rdt=2`, it would double twice; that is, the volume would quadruple. And if `rdt=-1`, it would halve in volume.

We can use the sample of RDTs to estimate the PDF of the distribution:

```
from utils import kde_from_sample

qs = np.linspace(-2, 6, num=201)
pmf_rdt = kde_from_sample(rdt_sample, qs)
```

Here's what it looks like:



In the next section we will use this distribution to simulate tumor growth.

Simulation

Now we're ready to run the simulations. Starting with a small tumor, we'll simulate a series of intervals until the tumor reaches a maximum size.

At the beginning of each simulated interval, we'll choose a value from the distribution of growth rates and compute the size of the tumor at the end.

I chose an interval of 245 days (about 8 months) because that is the median time between measurements in the data source.

For the initial diameter 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 [this page on carcinoma](#)). For the maximum diameter I chose 20 cm.

```
interval = 245 / 365      # year
min_diameter = 0.3        # cm
max_diameter = 20         # cm
```

I'll use `calc_volume` to compute the initial and maximum volumes:

```
v0 = calc_volume(min_diameter)
vmax = calc_volume(max_diameter)
v0, vmax
(0.014137166941154066, 4188.790204786391)
```

The following function runs the simulation:

```
import pandas as pd

def simulate_growth(pmf_rdt):
    """Simulate the growth of a tumor."""
    age = 0
    volume = v0
    res = []

    while True:
        res.append((age, volume))
        if volume > vmax:
            break

        rdt = pmf_rdt.choice()
        age += interval
        doublings = rdt * interval
        volume *= 2**doublings

    columns = ['age', 'volume']
    sim = pd.DataFrame(res, columns=columns)
    sim['diameter'] = calc_diameter(sim['volume'])
    return sim
```

`simulate_growth` takes as a parameter a `Pmf` that represents the distribution of RDT. It initializes the age and volume of the tumor, then runs a loop that simulates one interval at a time.

Each time through the loop, it checks the volume of the tumor and exits if it exceeds `vmax`.

Otherwise it chooses a value from `pmf_rdt` and updates `age` and `volume`. Since `rdt` is in doublings per year, we multiply by `interval` to compute the number of doublings during each interval.

At the end of the loop, `simulate_growth` puts the results in a `DataFrame` and computes the diameter that corresponds to each volume.

Here's how we call this function:

```
sim = simulate_growth(pmf_rdt)
```

Here are the results for the first few intervals:

```
sim.head(3)
```

	age	volume	diameter
0	0.000000	0.014137	0.300000
1	0.671233	0.014949	0.305635
2	1.342466	0.019763	0.335441

And the last few intervals:

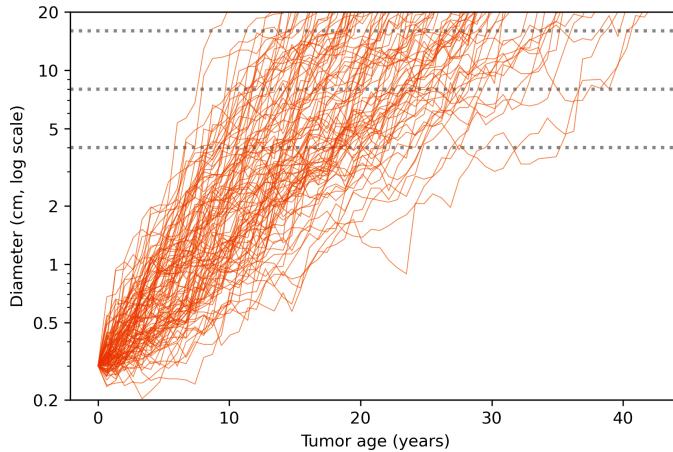
```
sim.tail(3)
```

	age	volume	diameter
43	28.863014	1882.067427	15.318357
44	29.534247	2887.563277	17.667603
45	30.205479	4953.618273	21.149883

To show the results graphically, I'll run 101 simulations:

```
sims = [simulate_growth(pmf_rdt) for _ in range(101)]
```

And plot the results:



In this figure, each thin, solid line shows the simulated growth of a tumor over time, with diameter on a log scale. The dotted lines are at 4, 8, and 16 cm.

By reading across the dotted lines, you can get a sense of the distribution of age at each size. For example, reading across the top line, we see that the age of a 16 cm tumor might be as low 10 years or as high as 40 years, but it is most likely to be between 15 and 30.

To compute this distribution more precisely, we can interpolate the growth curves to see when each one passes through a given size. The following function takes the results of the simulations and returns the age when each tumor reached a given diameter:

```
from scipy.interpolate import interp1d

def interpolate_ages(sims, diameter):
    """Estimate the age when each tumor reached a given size."""
    ages = []
    for sim in sims:
        interp = interp1d(sim['diameter'], sim['age'])
        age = interp(diameter)
        ages.append(float(age))
    return ages
```

We can call this function like this:

```
from empiricaldist import Cdf

ages = interpolate_ages(sims, 15)
cdf = Cdf.from_seq(ages)
print(cdf.median(), cdf.credible_interval(0.9))

22.31854530374061 [13.47056554 34.49632276]
```

For a tumor 15 cm in diameter, the median age is about 22 years, the 90% credible interval is between 13 and 34 years, and the probability that it formed less than 9 years ago is less than 1%:

`1 - cdf(9.0)`

`0.9900990099009901`

But this result is based on two modeling decisions that are potentially problematic:

- In the simulations, growth rate during each interval is independent of previous growth rates. In reality it is plausible that tumors that have grown quickly in the past are likely to grow quickly in the future. In other words, there is probably a serial correlation in growth rate.
- To convert from linear measure to volume, we assume that tumors are approximately spherical.

In additional experiments, I implemented a simulation that chooses growth rates with serial correlation; the effect is that the fast-growing tumors grow faster and the slow-growing tumors grow slower. Nevertheless, with moderate correlation (0.5), the probability that a 15 cm tumor is less than 9 years old is only about 1%.

The assumption that tumors are spherical is probably fine for tumors up to a few centimeters, but not for a tumor with linear dimensions 15.5 x 15 cm. If, as seems likely, a tumor this size is relatively flat, it might have the same volume as a 6 cm sphere. But even with this smaller volume and correlation 0.5, the probability that this tumor is less than 9 years old is about 5%.

So even taking into account modeling errors, it is unlikely that such a large tumor could have formed after my correspondent retired from military service.

Approximate Bayesian Computation

At this point you might wonder why this example is in a book about Bayesian statistics. We never defined a prior distribution or did a Bayesian update. Why not? Because we didn't have to.

Instead, we used simulations to compute ages and sizes for a collection of hypothetical tumors. Then, implicitly, we used the simulation results to form a joint distribution of age and size. If we select a column from the joint distribution, we get a distribution of size conditioned on age. If we select a row, we get a distribution of age conditioned on size.

So this example is like the ones we saw in [Chapter 1](#): if you have all of the data, you don't need Bayes's theorem; you can compute probabilities by counting.

This example is a first step toward Approximate Bayesian Computation (ABC). The next example is a second step.

Counting Cells

This example comes from [this blog post](#), by Cameron Davidson-Pilon. In it, he models the process biologists use to estimate the concentration of cells in a sample of liquid. The example he presents is counting cells in a “yeast slurry”, which is a mixture of yeast and water used in brewing beer.

There are two steps in the process:

- First, the slurry is diluted until the concentration is low enough that it is practical to count cells.
- Then a small sample is put on a hemocytometer, which is a specialized microscope slide that holds a fixed amount of liquid on a rectangular grid.

The cells and the grid are visible in a microscope, making it possible to count the cells accurately.

As an example, suppose we start with a yeast slurry with an unknown concentration of cells. Starting with a 1 mL sample, we dilute it by adding it to a shaker with 9 mL of water and mixing well. Then we dilute it again, and then a third time. Each dilution reduces the concentration by a factor of 10, so three dilutions reduces the concentration by a factor of 1,000.

Then we add the diluted sample to the hemocytometer, which has a capacity of 0.0001 mL spread over a 5 x 5 grid. Although the grid has 25 squares, it is standard practice to inspect only a few of them, say 5, and report the total number of cells in the inspected squares.

This process is simple enough, but at every stage there are sources of error:

- During the dilution process, liquids are measured using pipettes that introduce measurement error.
- The amount of liquid in the hemocytometer might vary from the specification.
- During each step of the sampling process, we might select more or less than the average number of cells, due to random variation.

Davidson-Pilon presents a PyMC model that describes these errors. I'll start by replicating his model; then we'll adapt it for ABC.

Suppose there are 25 squares in the grid, we count 5 of them, and the total number of cells is 49:

```
total_squares = 25
squares_counted = 5
yeast_counted = 49
```

Here's the first part of the model, which defines the prior distribution of yeast_conc, which is the concentration of yeast we're trying to estimate.

shaker1_vol is the actual volume of water in the first shaker, which should be 9 mL, but might be higher or lower, with standard deviation 0.05 mL. shaker2_vol and shaker3_vol are the volumes in the second and third shakers.

```
import pymc3 as pm
billion = 1e9

with pm.Model() as model:
    yeast_conc = pm.Normal("yeast conc",
                           mu=2 * billion, sd=0.4 * billion)

    shaker1_vol = pm.Normal("shaker1 vol",
                           mu=9.0, sd=0.05)
    shaker2_vol = pm.Normal("shaker2 vol",
                           mu=9.0, sd=0.05)
    shaker3_vol = pm.Normal("shaker3 vol",
                           mu=9.0, sd=0.05)
```

Now, the sample drawn from the yeast slurry is supposed to be 1 mL, but might be more or less. And similarly for the sample from the first shaker and from the second shaker. The following variables model these steps:

```
with model:
    yeast_slurry_vol = pm.Normal("yeast slurry vol",
                                  mu=1.0, sd=0.01)
    shaker1_to_shaker2_vol = pm.Normal("shaker1 to shaker2",
                                       mu=1.0, sd=0.01)
    shaker2_to_shaker3_vol = pm.Normal("shaker2 to shaker3",
                                       mu=1.0, sd=0.01)
```

Given the actual volumes in the samples and in the shakers, we can compute the effective dilution, final_dilution, which should be 1,000, but might be higher or lower.

```
with model:
    dilution_shaker1 = (yeast_slurry_vol /
                         (yeast_slurry_vol + shaker1_vol))
    dilution_shaker2 = (shaker1_to_shaker2_vol /
                         (shaker1_to_shaker2_vol + shaker2_vol))
    dilution_shaker3 = (shaker2_to_shaker3_vol /
                         (shaker2_to_shaker3_vol + shaker3_vol))

    final_dilution = (dilution_shaker1 *
                       dilution_shaker2 *
                       dilution_shaker3)
```

The next step is to place a sample from the third shaker in the chamber of the hemocytometer. The capacity of the chamber should be 0.0001 mL, but might vary; to describe this variance, we'll use a gamma distribution, which ensures that we don't generate negative values:

```
with model:  
    chamber_vol = pm.Gamma("chamber_vol",  
                           mu=0.0001, sd=0.0001 / 20)
```

On average, the number of cells in the chamber is the product of the actual concentration, final dilution, and chamber volume. But the actual number might vary; we'll use a Poisson distribution to model this variance:

```
with model:  
    yeast_in_chamber = pm.Poisson("yeast_in_chamber",  
                                   mu=yeast_conc * final_dilution * chamber_vol)
```

Finally, each cell in the chamber will be in one of the squares we count with probability $p=squares_counted/total_squares$. So the actual count follows a binomial distribution:

```
with model:  
    count = pm.Binomial("count",  
                         n=yeast_in_chamber,  
                         p=squares_counted/total_squares,  
                         observed=yeast_counted)
```

With the model specified, we can use `sample` to generate a sample from the posterior distribution:

```
options = dict(return_inferencedata=False)  
  
with model:  
    trace = pm.sample(1000, **options)
```

And we can use the sample to estimate the posterior distribution of `yeast_conc` and compute summary statistics:

```
posterior_sample = trace['yeast conc'] / billion  
cdf_pymc = Cdf.from_seq(posterior_sample)  
print(cdf_pymc.mean(), cdf_pymc.credible_interval(0.9))  
  
2.26789764737366 [1.84164524 2.70290741]
```

The posterior mean is about 2.3 billion cells per mL, with a 90% credible interval from 1.8 and 2.7.

So far we've been following in Davidson-Pilon's footsteps. And for this problem, the solution using MCMC is sufficient. But it also provides an opportunity to demonstrate ABC.

Cell Counting with ABC

The fundamental idea of ABC is that we use the prior distribution to generate a sample of the parameters, and then simulate the system for each set of parameters in the sample.

In this case, since we already have a PyMC model, we can use `sample_prior_predictive` to do the sampling and the simulation:

```
with model:  
    prior_sample = pm.sample_prior_predictive(10000)
```

The result is a dictionary that contains samples from the prior distribution of the parameters and the prior predictive distribution of `count`:

```
count = prior_sample['count']  
print(count.mean())
```

39.9847

Now, to generate a sample from the posterior distribution, we'll select only the elements in the prior sample where the output of the simulation, `count`, matches the observed data, 49:

```
mask = (count == 49)  
mask.sum()  
  
251
```

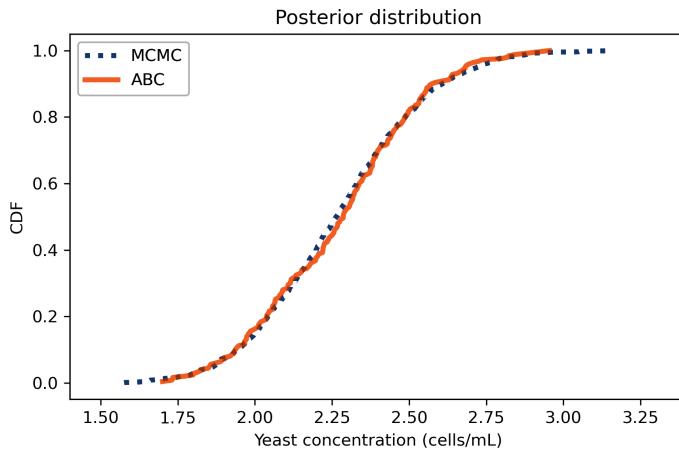
We can use `mask` to select the values of `yeast_conc` for the simulations that yield the observed data:

```
posterior_sample2 = prior_sample['yeast conc'][mask] / billion
```

And we can use the posterior sample to estimate the CDF of the posterior distribution:

```
cdf_abc = Cdf.from_seq(posterior_sample2)  
print(cdf_abc.mean(), cdf_abc.credible_interval(0.9))  
  
2.2635057237709755 [1.85861977 2.68665897]
```

The posterior mean and credible interval are similar to what we got with MCMC. Here's what the distributions look like:



The distributions are similar, but the results from ABC are noisier because the sample size is smaller.

When Do We Get to the Approximate Part?

The examples so far are similar to Approximate Bayesian Computation, but neither of them demonstrates all of the elements of ABC. More generally, ABC is characterized by:

1. A prior distribution of parameters.
2. A simulation of the system that generates the data.
3. A criterion for when we should accept that the output of the simulation matches the data.

The kidney tumor example was atypical because we didn't represent the prior distribution of age explicitly. Because the simulations generate a joint distribution of age and size, we were able to get the marginal posterior distribution of age directly from the results.

The yeast example is more typical because we represented the distribution of the parameters explicitly. But we accepted only simulations where the output matches the data exactly.

The result is approximate in the sense that we have a sample from the posterior distribution rather than the posterior distribution itself. But it is not approximate in the sense of Approximate Bayesian Computation, which typically accepts simulations where the output matches the data only approximately.

To show how that works, I will extend the yeast example with an approximate matching criterion.

In the previous section, we accepted a simulation if the output is precisely 49 and rejected it otherwise. As a result, we got only a few hundred samples out of 10,000 simulations, so that's not very efficient.

We can make better use of the simulations if we give “partial credit” when the output is close to 49. But how close? And how much credit?

One way to answer that is to back up to the second-to-last step of the simulation, where we know the number of cells in the chamber, and we use the binomial distribution to generate the final count.

If there are n cells in the chamber, each has a probability p of being counted, depending on whether it falls in one of the squares in the grid that get counted.

We can extract n from the prior sample, like this:

```
n = prior_sample['yeast in chamber']
n.shape
(10000,)
```

And compute p like this:

```
p = squares_counted/total_squares
p
0.2
```

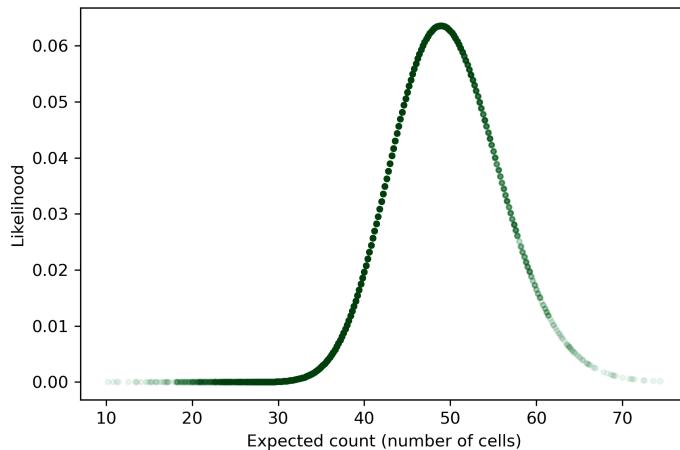
Now here's the idea: we'll use the binomial distribution to compute the likelihood of the data, `yeast_counted`, for each value of n and the fixed value of p :

```
from scipy.stats import binom

likelihood = binom(n, p).pmf(yeast_counted).flatten()
```

When the expected count, $n * p$, is close to the actual count, `likelihood` is relatively high; when it is farther away, `likelihood` is lower.

The following is a scatter plot of these likelihoods versus the expected counts:



We can't use these likelihoods to do a Bayesian update because they are incomplete; that is, each likelihood is the probability of the data given n , which is the result of a single simulation.

But we *can* use them to weight the results of the simulations. Instead of requiring the output of the simulation to match the data exactly, we'll use the likelihoods to give partial credit when the output is close.

Here's how: I'll construct a `Pmf` that contains yeast concentrations as quantities and the likelihoods as unnormalized probabilities.

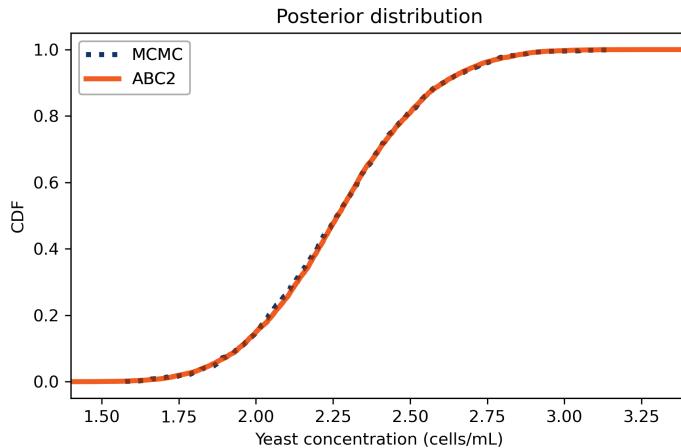
```
qs = prior_sample['yeast conc'] / billion
ps = likelihood
posterior_pmf = Pmf(ps, qs)
```

In this `Pmf`, values of `yeast_conc` that yield outputs close to the data map to higher probabilities. If we sort the quantities and normalize the probabilities, the result is an estimate of the posterior distribution.

```
posterior_pmf.sort_index(inplace=True)
posterior_pmf.normalize()

print(posterior_pmf.mean(), posterior_pmf.credible_interval(0.9))
2.271401984584812 [1.85333758 2.71299385]
```

The posterior mean and credible interval are similar to the values we got from MCMC. And here's what the posterior distributions look like:



The distributions are similar, but the results from MCMC are a little noisier. In this example, ABC is more efficient than MCMC, requiring less computation to generate a better estimate of the posterior distribution. But that's unusual; usually ABC requires a lot of computation. For that reason, it is generally a method of last resort.

Summary

In this chapter we saw two examples of Approximate Bayesian Computation (ABC), based on simulations of tumor growth and cell counting.

The definitive elements of ABC are:

1. A prior distribution of parameters.
2. A simulation of the system that generates the data.
3. A criterion for when we should accept that the output of the simulation matches the data.

ABC is particularly useful when the system is too complex to model with tools like PyMC. For example, it might involve a physical simulation based on differential equations. In that case, each simulation might require substantial computation, and many simulations might be needed to estimate the posterior distribution.

Next, you'll have a chance to practice with one more example.

Exercises

Exercise 20-1.

This exercise is based on [a blog post by Rasmus Bååth](#), which is motivated by a tweet from Karl Broman, who wrote:

That the first 11 socks in the laundry are distinct suggests that there are a lot of socks.

Suppose you pull 11 socks out of the laundry and find that no two of them make a matched pair. Estimate the number of socks in the laundry.

To solve this problem, we'll use the model Bååth suggests, which is based on these assumptions:

- The laundry contains some number of pairs of socks, `n_pairs`, plus some number of odd (unpaired) socks, `n_odds`.
- The pairs of socks are different from each other and different from the unpaired socks; in other words, the number of socks of each type is either 1 or 2, never more.

We'll use the prior distributions Bååth suggests, which are:

- The number of socks follows a negative binomial distribution with mean 30 and standard deviation 15.
- The proportion of socks that are paired follows a beta distribution with parameters `alpha=15` and `beta=2`.

In the notebook for this chapter, I'll define these priors. Then you can simulate the sampling process and use ABC to estimate the posterior distributions.

Index

Symbols

- [] (bracket operator), 6
 - probability mass functions, 30
- | (given), 9
- + (plus) versus Pmf.add_dist(), 93

A

- age of the universe, 53
- Anaconda distribution of Python, xi
- Approximate Bayesian Computation (ABC)
 - about, 287, 299
 - counting cells via ABC, 298-302
 - counting cells via MCMC, 295-297
 - Kidney Tumor Problem
 - ABC aspect, 294
 - about, 287, 299
 - growth model, general, 289
 - growth model, simple, 288
 - simulation of growth, 291-294
- arrays
 - coin tossed twice, 44
 - DataFrame converted to NumPy array, 149
 - meshgrid function
 - comparison operators, 146, 151
 - joint distribution construction, 148
 - likelihood of height of person, 151
 - outer product, 145, 148
 - outer sum, 146
 - 3-dimensional for reading ability, 178
 - normal distribution of height, 147
 - np.repeat function, 200
 - np.where function, 151
 - parentheses versus brackets, 75
 - 3-dimensional to 2-dimensional, 179

- transposing DataFrame matrix, 95
 - triangle-shaped prior, 49
 - weighted mixture of distributions, 95
- ArviZ plot_posterior, 279
- Axtell, Robert, 61

B

- Basu's theorem, 185
- Bayes factor, 72, 131, 134, 234
- Bayes tables
 - Cookie Problem, 20-22
 - Dice Problem, 22
 - Monty Hall Problem, 23-25
- Bayesian Bandit strategy
 - about, 134
 - multiple bandits, 137
 - prior beliefs, 135
 - strategy put together, 140
 - testing, 140
 - update, 136
 - which machine to play, 138
- Bayesian decision analysis
 - bandit strategy (see Bayesian Bandit strategy)
 - instead of hypothesis testing, 134
- Price Is Right Problem
 - about, 113
 - decision analysis, 122
 - distribution of errors, 116-118
 - kernel density estimation, 115, 116
 - maximizing expected gain, 124
 - modeling, 116
 - prior, 114
 - probability of winning, 120

- update, 118
- questions to ask, 134
- Bayesian estimation in Euro Problem, 47-51
- Bayesian hypothesis testing
 - Bayesian Bandit strategy, 140
 - decision analysis instead, 134
 - Euro Problem
 - about, 129
 - about previous solution, 129-131
 - binomial distribution, 46, 130
 - modeling, 131
 - modeling triangle-shaped bias, 133
 - modeling uniform bias, 132
 - statistical versus, 134
 - Bayesian Methods for Hackers (Davidson-Pilon), 114, 226
 - Bayesian regression (see linear regression)
 - Bayesian statistics versus Bayes's theorem, 53
 - Bayesian updates, 20
 - about, 19, 260
 - Bayes tables for, 20-22
 - Bayesian logistic regression, 231
 - dictionary for ease, 48
 - gamma distribution for, 259-261
 - How Tall Is Person A, 152
 - log odds, 224
 - update additive, 226
 - Pmf objects for, 33
 - Bayesian Bandit strategy, 136
 - classification of penguin data, 164-168
 - Cookie Problem, 33
 - Dice Problem, 39
 - World Cup Problem, 103
 - posterior distribution location, 105
 - Price Is Right Problem, 118
 - reading improvement groups, 177-180
 - summary statistics, 186
 - snow amounts, 247
 - wrong classroom, 224
 - update additive, 226
 - Bayes's rule
 - about, 223
 - Bayes's theorem in odds form, 70
 - Cookie Problem, 71, 72
 - Oliver's Blood, 71-73
 - wrong classroom, 224, 225
 - Bayes's theorem, 10
 - Bayesian statistics versus, 53
 - Cookie Problem, 17-19
- derivation of, 8-10
- diachronic Bayes, 19
- example of use, 11
- gluten sensitivity, 76
 - Forward Problem, 77
 - Inverse Problem, 78
- bears (see Grizzly Bear Problem)
- beta distribution, 261
 - Dirichlet distribution marginals as, 266
 - SciPy beta function, 261
- bidding strategy (see Price Is Right Problem)
- binomial distribution, 44
 - beta distribution, 261
 - Dirichlet distribution marginals as, 266
 - SciPy beta function, 261
 - binomial coefficient, 44
 - hypergeometric distribution, 208
 - conjugate prior of, 261
 - Euro Problem tested, 46, 130
 - gluten sensitivity, 76
 - Forward Problem, 77
 - Inverse Problem, 78
 - light bulb dead bulb prediction, 202
 - SciPy binomial function, 44
- binomial likelihood function, 51, 129
 - conjugate priors, 261
- blood type problem, 71-73
- Boolean Series
 - junctions, 5
 - impossible outcomes, 40
 - incomplete data, 195
 - summing, 3
 - probability function, 4
- Box, George, 99
- bracket operator ([]), 6
 - probability mass functions, 30
- bugs in program (see Lincoln Index Problem)

C

- cancer (see Kidney Tumor Problem)
- Cantril ladder question on happiness, 277
- CDF (see cumulative distribution function)
- Cdf objects
 - about, 86, 88
 - classification of penguin data, 162
 - complementary CDF, 89
 - distribution of differences, 182
 - distribution, maximum of six attributes, 88
 - max_dist function, 89

distribution, minimum of six attributes, 89
min_dist function, 90
empiricalist library for Cdf class, 85
Pmf object conversion, 86
reading improvement groups, 176
Weibull distribution, 192-194

censored data, 195
centering data to minimize correlation, 232, 244

Central Limit Theorem, 76

classification of penguin data
about, 161
cumulative distribution functions, 162
data description, 161
data source, 161
joint distributions, 168
scatter plot, 168
scatter plot compared to contours, 169
less naive Bayesian classifier, 172
loading into DataFrame, 161
multivariate normal distribution, 170
normal models, 163, 169
update, 164-168
naive Bayesian classifier, 166

coin fairness (see Euro Problem)

Colab to run Jupyter notebooks, x

collectively exhaustive, 11
gender as, 12
law of total probability, 11, 19

commutative property
conditional probability, 7
conjunctions, 6, 10

company sizes following power law, 61

Conda environment for book code, xi

conditional posteriors, 156

conditional probability
about, 6
commutative property, 7
computing, 6
conjunction to compute, 9
conjunctions and, 8
conjunction as product of probabilities, 10
relationship in math notation, 9

Linda the Banker Problem, 1

probability function, 7

conjugate priors
about, 258
animal preserve with three parameters, 263

Dirichlet distribution, 264

Euro Problem, 261

World Cup Problem
gamma distribution for update, 258-261
Poisson processes solution review, 257

conjunctions, 5
commutative property, 6, 10
conditional probability and, 8
product of probabilities, 10
relationship in math notation, 9
conditional probability computed via, 9

contour plot
joint distribution, 150
Grizzly Bear two-parameter model, 214

Weibull distribution, 193
incomplete data, 197

Cook, John D., 215

Cookie Problem
Bayes tables, 20-22
Bayes's rule, 71
Bayes's theorem, 17-19
likelihood, 19
odds, 72
101 Bowls Problem, 34-37
Euro Problem contrasted, 52

Pmf objects, 29, 32-34
updated data, 33
prior, 19

correlation minimized by centering data, 232, 244

count estimation (see counting cells; estimating counts)

Counter function, 137

counting cells
Approximate Bayesian Computation, 298
MCMC, 295

covariance matrix, 170

credible intervals, 64, 249
credible_interval function, 64

cumulative distribution function (CDF)
about, 83
0 to 1 range, 84

Cdf objects (see Cdf objects)

classification of penguin data, 162

complementary CDF, 89

distribution as mix of distributions, 90-93
general solution, 93-96

distribution of differences, 182

empiricalist for Cdf class, 85

Euro Problem, 83
np.diff function, 86
PMF conversion, 86
reading improvement groups, 176
Weibull distribution, 192-194

D

data
classification of penguin data, 161-172
data source, 161
data in hand, Bayes's theorem not needed, 294
Empirical Bayes method data reused, 237
empirical versus theoretical distributions, 29
evidence in favor of a hypothesis, 72, 131
Bayes factor, 72, 131, 134, 234
evidence of biased coin, 43-52
Bayes's rule for, 132
groupby for data into groups, 176, 230, 242
GSS (General Social Survey) dataset, 2-5, 7, 11
incomplete data, 194-196
called censored, 195
marginal distributions, 197
using, 196-199
informative versus uninformative prior, 65
Price Is Right prices and bids, 114
priors converging on same posterior, 51
summary statistics, 184
swamping the priors, 51
updating probability with new data, 19
(see also Bayesian updates)
weather records, 241

DataFrames (see pandas)

Davidson-Pilon, Cameron, 114, 226, 295

decision analysis in Price Is Right, 122
(see also Bayesian decision analysis)

degree of certainty via odds, 69
(see also odds)

dependence and independence of heights, 157

diachronic Bayes's theorem, 19

Dice Problem
Bayes tables, 22
distribution of sums
three dice, 75
two dice, 73-76
Dungeons & Dragons best three, 86
Pmf to solve, 38-39
updating dice, 39

dictionaries
classification of penguin data, 163
ease of updating, 48

Dirichlet distribution, 264
marginals as beta distributions, 266
discrete distributions and numerical errors, x
distribution objects, 272
distribution of differences, 181
plotting, 182
distribution of errors, 116-118

distributions
about, 29
beta distribution, 261
Dirichlet distribution marginals as, 266
SciPy beta function, 261
Cdf for maximum or minimum, 88-90
coin tossed twice, 44
company sizes following power law, 61
conditional distribution, 156
Cookie Problem
101 bowls of cookies, 34-37
Pmf for, 32-34
Pmf for updated data, 33
cumulative distribution function
0 to 1 range, 84
about, 83
classification of penguin data, 162
complementary CDF, 89
distribution of differences, 182
empiricaldist for Cdf class, 85
(see also Cdf objects)
Euro Problem, 83
np.diff function, 86
PMF conversion, 86
reading improvement groups, 176
Weibull distribution, 192-194

Dirichlet distribution, 264

discrete distributions, x

distribution as mix of distributions, 90-93
general solution, 93-96

distribution of differences, 181
plotting, 182

distribution of errors, 116-118

empirical versus theoretical, 29

exponential distribution, 108
SciPy expon, 109

gluten sensitivity, 76
Forward Problem, 77
Inverse Problem, 78

- hypergeometric distribution, 208
joint distributions, 145, 148
(see also joint distributions)
kernel density estimation, 115
light bulb lifetime distribution, 199-202
dead bulb prediction, 202
marginal distributions, 154
Dirichlet marginals as beta distributions, 266
incomplete data, 197
joint distributions to, 153-156
logistic regression, Bayesian, 232
Pmf marginal function, 219
reading improvement, 180
reading improvement compared, 187
snow amounts, 247
median as 50th percentile, 63
Poisson distribution, 100, 103
gamma distribution as conjugate prior, 258
probability of superiority, 105
posterior distribution, 33, 35
(see also posterior distribution)
predictive distributions
posterior, 106, 202, 235-237, 254
prior, 270
prior distribution, 32, 34
(see also prior distribution)
probability mass functions, 29
Bayesian updates, 260
CDF conversion, 86
coin toss, 30
empiricaldist library for Pmf class, 30
(see also Pmf objects)
outcomes appearing more than once, 30
sequence of possible outcomes, 30
random samples from (see MCMC (Markov chain Monte Carlo))
sampling distribution of the mean, 185
sums of three dice, 75, 86
sums of two dice, 73-76
Weibull distribution, 191-194
light bulb dead bulb prediction, 202
weighted distributions, 94
dog shelter adoption
about, 191
incomplete data, 194-196
called censored, 195
using, 196-199
Weibull distribution, 191-194
Dungeons & Dragons dice rolls, 86
distribution as mix of distributions, 90-93
general solution, 93-96
- ## E
- Empirical Bayes method using data twice, 237
empiricaldist library, 29
Cdf class, 85
installation, xi
Pmf class, 30
errors in document or program (see Lincoln Index Problem)
estimating counts
counting cells via ABC, 298-302
counting cells via MCMC, 295-297
data in hand, Bayes's theorem not needed, 294
German Tank Problem, 64
Train Problem, 57-60
credible intervals, 63
power law prior, 61
sensitivity to the prior, 60
estimating proportions
Euro Problem
about, 43
Bayesian estimation, 47-51
Bayesian statistics versus Bayes's theorem, 53
binomial distribution, 44
binomial likelihood function, 51
modeling, 44, 129
101 Bowls Problem, 34-37, 52
Euro Problem
about, 43
Bayesian estimation, 47-51
binomial distribution, 44
binomial coefficient, 44
conjugate prior of, 261
SciPy binomial function, 44
unbiased coin results tested, 46
binomial likelihood function, 51, 129
conjugate prior of binomial distribution, 261
cumulative distribution function, 83
modeling, 44, 129, 131
101 Bowls Problem contrasted, 52
random versus nonrandom quantities, 52
testing

about, 129
modeling, 131
modeling triangle-shaped bias, 133
modeling uniform bias, 132
solution review, 129-131
exponential distribution, 108
probability density function of, 109
SciPy expon, 109

F

False value summed, 3
Fifty Challenging Problems in Probability with Solutions (Mosteller), 57
floating-point rounding avoided, 22
Forward Problem of gluten sensitivity distribution, 77
fraction of items
 probability function computing, 4
 Series of Boolean values, 3
fractions to avoid floating-point rounding, 22

G

Gallup World Poll on happiness, 277
gamma distribution
 about, 102
 Bayesian updates via, 259-261
 conjugate priors, 258
 goal-scoring rate, 101
 SciPy gamma function, 102
generator expressions, 13
German Tank Problem, 64
given (), 9
gluten sensitivity distribution, 76
 Forward Problem, 77
 Inverse Problem, 78
goal scoring (see World Cup Problem)
Gorman, Kristen, 161
Gould, Stephen J., 2
Grizzly Bear Problem
 about, 207
 estimating total population, 209
 probability of observing a bear, 211-215
hypergeometric distribution, 208
modeling, 208
 three-parameter model, 263
 two-parameter model, 211-215
plotting, 210
 two-parameter model, 214
update, 209

two-parameter model, 213
wild animal preserve, 263
groupby for data into groups, 176, 230, 242
GSS (General Social Survey) dataset, 2-5, 7, 11

H

happiness
 about, 276
 multiple regression via PyMC3 library, 280
 simple regression, 277
 PyMC3 library, 278
 SciPy linregress function, 278
Happiness and Life Satisfaction (Ortiz-Ospina and Roser), 276
How Tall Is Person A
 about, 147
 B height from A, 156
 independence of A and B, 157
 joint distribution construction, 148
 likelihood, 151
 marginal distributions, 153-156
 plotting joint distribution, 149
 prior distribution of height, 147
 update, 152
hypergeometric distribution, 208
 SciPy hypergeom function, 208

hypotheses
 any number of
 Cookie Problem with 101 bowls, 34-37
 law of total probability, 20
 evidence in favor of, 72, 131
 Bayes factor, 72, 131, 134, 234
 hypothesis testing, 134
 (see also testing hypotheses)
 decision analysis instead, 134
 three hypotheses
 Bayes tables, 22, 23-25
 coin tossed twice, 44
 Monty Hall Problem, 23-25
 two hypotheses
 Bayes tables, 20-22
 binomial distribution, 44
 law of total probability, 20
 updating probability with new data, 19
 (see also Bayesian updates)

I

incomplete data, 194-196
 called censored, 195

marginal distributions, 197
using, 196-199
independence and dependence of heights, 157
indus10 industry code, 3
inference
 p-values, 175
 reading ability improvement
 about, 175
 data into DataFrame, 176
 distribution of differences, 181
 groupby for data into groups, 176
 likelihood, 178
 likelihood summary statistics, 184
 marginal distributions, 180
 marginal distributions compared, 187
 prior distribution, 177
 probability of superiority, 181
 update, 177-180
 update with summary statistics, 186
statistical versus Bayesian, 175
Information Theory, Inference, and Learning Algorithms (MacKay), xiii, 43, 71, 129
installing Jupyter, xi
Inverse Problem of gluten sensitivity distribution, 78

J

joint distributions
 about, 145, 148
 constructing, 148
How Tall Is Person A
 about, 147
 B height from A, 156
 independence of A and B, 157
 joint distribution construction, 148
 likelihood, 151
 marginal distributions, 153-156
 plotting joint distribution, 149
 prior distribution of height, 147
 update, 152
marginal distributions from, 153-156
outer operations, 145
 comparison operators, 146, 151
 joint distribution construction, 148
 outer product, 145
 outer sum, 146
plotting, 149
 contour of Pmf Series, 214
 scatter plot of penguin data, 168

reading ability improvement, 177
3-dimensional, 217
Jupyter notebooks
 about running notebooks, x
 installing Jupyter, xi

K

kernel density estimation (KDE), 115, 116
 distribution of differences plotted, 182
 SciPy gaussian_kde function, 115, 182
Kidney Tumor Problem
 about, 287, 299
 Approximate Bayesian Computation, 294
growth model, general, 289
growth model, simple, 288
simulation of growth, 291-294

L

law of total probability, 11
Cookie Problem, 18
Price Is Right decision analysis, 122
total probability of the data, 19
least squares regression
 marathon world record, 251
 snow amounts, 244
light bulb failure time
 about, 191
 dead bulb prediction, 202
 distribution of lifetimes, 199-202
 incomplete data, 194-196
 called censored, 195
 using, 196-199
Weibull distribution, 191-194
likelihood, 19
Bayes tables
 three hypotheses, 22, 23-25
 two hypotheses, 20-22
Bayesian logistic regression, 230
binomial likelihood function, 51, 129
classification of penguin data, 163
computing for entire dataset at once, 51
dictionary to hold, 48
Grizzly Bear with two parameters, 213
How Tall Is Person A, 151
 B height from A, 156
likelihood ratios as Bayes factors, 234
posterior odds, 71
reading ability improvement, 178
snow amounts, 246

- summary statistics for larger datasets, 184, 247
 time between goals, 108
 too small for floating-point arithmetic, 184, 247
Train Problem, 57
 uniform prior, 47, 129
 wrong classroom, 224
Lincoln Index Problem, 215-217
 modeling three parameters, 217
 modeling two testers, 215
Linda the Banker Problem, 1
 linear regression
 about, 242
 least squares regression
 marathon world record, 251
 snow amounts, 244
 marathon world record, 250
 mathematical model, 243
 residuals, 245
 SciPy linregress function, 278
 snow amounts, 241-250
 likelihood, 246
 priors, 245
 residuals, 245, 246
 update, 247
 locomotive count estimation, 57-60
 log odds
 about, 225, 241
 explanatory and dependent variables, 226
 probabilities from, 228, 233
 SciPy expit function, 228, 231, 233
 Space Shuttle Problem, 231, 233
 wrong classroom, 223-226
 logical AND (see conjunctions)
 logistic regression
 about, 223, 241
 Bayesian
 likelihood, 230
 marginal distributions, 232
 prior distribution, 229
 transforming distributions, 233
 update, 231
 Empirical Bayes method, 237
 log odds, 223-226
 predictive distributions, 235-237
 Space Shuttle Problem
 about, 226
 logistic regression, Bayesian, 229-232
 logistic regression, non-Bayesian, 228
 modeling, 237
 modeling, logistic model, 227, 237
 predictions about O-rings, 235-237
 statsmodels for non-Bayesian, 228
- ## M
- MacKay, David, xiii, 43, 71, 129
 MAP as highest posterior probability, 37
 coin tossed twice, 45
 computing, 37
 marathon world record
 about, 250
 least squares regression, 251
 likelihoods, 252
 marginal distributions, 252
 prediction of time-barrier broken, 254
 priors, 252
 marginal distributions, 154
 Dirichlet distribution marginals as beta distributions, 266
 incomplete data, 197
 joint distributions to, 153-156
 logistic regression, Bayesian, 232
 Pmf marginal function, 219
 reading ability improvement, 180
 comparing, 187
 snow amounts, 247
 mark and recapture experiments
 about, 207
 Grizzly Bear Problem
 about, 207
 estimating total population, 209
 modeling, 208
 modeling two parameters, 211-215
 update, 209
 update with two parameters, 213
 hypergeometric distribution, 208
 Lincoln Index Problem, 215-217
 modeling three parameters, 217
 modeling two testers, 215
 Markov chain (see MCMC (Markov chain Monte Carlo))
 mathematical notation for probability, 8
 Bayes's theorem, 10, 70
 conditional probability and conjunctions, 9
 conjunctions as commutative, 10
 law of total probability, 11
 power law, 61

regression model, 243
matplotlib
 installation, xi
 joint distribution plotted, 149
 scatter plot, 168
matrix transposition, 95
maximizing expected gain, 124
McGrayne, Sharon Bertsch, 53
MCMC (Markov chain Monte Carlo)
 about, 269
 happiness
 about, 276
 multiple regression, PyMC3 library, 280
 simple regression, 277
 simple regression, PyMC3 library, 278
 simple regression, SciPy linregress, 278
PyMC3 library, 271
 about, 274
 inference, 274
 sampling the posterior predictive distribution, 275
 sampling the prior, 272
World Cup Problem, 269-276
 gamma distribution prior, 270
 goal-scoring rate possible values, 270
 inference, 274
 Poisson process review, 269
 predicting rematch, 275
 PyMC3 library, 271
 sampling the prior, 272
mean function
 centering data to minimize correlation, 232, 244
 fraction computed via, 3, 9
mean of posterior distribution, 59, 62
 Bayesian updates and, 105
 distribution skew, 210
 joint distributions, 201
 multivariate normal distribution, 170
 sampling distribution of the mean, 185
mean squared error, 60
MECE (mutually exclusive and collectively exhaustive), 12
median of distribution percentile, 63
mesh grids
 comparison operators, 146
 height arrays, 151
likelihood of height of person, 151
outer product, 145
 joint distribution construction, 148
 outer sum, 146
 3-dimensional for reading ability, 178
Model object, 272
modeling
 about modeling errors, x
 all models wrong, 99
 8 parameters via PyMC3, 280
 Euro Problem, 44, 129, 131
 triangle-shaped bias, 133
 uniform bias, 132
 gluten sensitivity distribution, 77
 Grizzly Bear Problem, 208
 two-parameter model, 211-215
 informative versus uninformative prior, 65
 Kidney Tumor Problem
 growth model, general, 289
 growth model, simple, 288
 Lincoln Index Problem
 three parameters, 217
 two testers, 215
 Price Is Right Problem, 116
 Space Shuttle Problem, 237
 logistic model, 227, 237
 3 parameters
 Lincoln Index Problem, 217
 simple regression via PyMC3, 278
 snow amounts, 241
 wild animal preserve, 263
 World Cup Problem, 99, 105
 PyMC3, 271
monster combat (see Dungeons & Dragons)
Monte Carlo (see MCMC (Markov chain Monte Carlo))
Monty Hall Problem via Bayes tables, 23-25
Mosteller, Frederick, 57
MultiIndex
 Bayesian logistic regression, 229
 Pmf objects, 213, 218, 229
 Series in pandas, 203, 213
 3-dimensional joint distribution, 218
multinomial distribution conjugate prior, 264
multinomial function in SciPy, 212, 219
multiple regression via PyMC3 library, 280
multivariate Dirichlet distribution, 264
multivariate normal distribution, 170
mutually exclusive, 11
 law of total probability, 11, 19

mutually exclusive and collectively exhaustive (MECE), 12

N

NaN as not a number, 94

normal distribution

- average height of male adults, 147
- classification of penguin data, 163, 169
- multivariate, 170
- probability density as Pmf normalized, 147
- reading improvement groups, 177
- SciPy norm function, 147, 163
 - probability density function, 147, 164
- snow amounts, 243
- univariate, 170
- update with summary statistics, 188

normalization, 22

joint posterior distribution, 152

normalizing constant, 22

outside of dataset loop, 48

Pmf function, 32

notebooks (Jupyter)

about running notebooks, x

installing Jupyter, xi

np alias for NumPy, 34

null hypothesis significance testing, 175

NumPy

array of values, 44

DataFrame converted to, 149

meshgrid function outer operations, 145

normal distribution of height, 147

repeat function, 200

triangle-shaped prior, 49

weighted mixture of distributions, 95

where function, 151

Cookie Problem with 101 bowls, 34-37

cumsum function, 84, 85

diff function, 86

import as np, 34

installation, xi

mean of posterior distribution, 59

O

O-rings on shuttles (see Space Shuttle Problem)
odds

about, 69

Bayes factor, 72

Bayes's rule, 70

Bayes's theorem in odds form, 70

Cookie Problem, 72

Bayes's rule, 71

log odds

about, 225, 241

explanatory and dependent variables, 226

probabilities from, 228, 233

SciPy expit function, 228, 231, 233

Space Shuttle Problem, 231, 233

wrong classroom, 223-226

odds against an event, 70

odds in favor of an event, 69

Oliver's Blood, 71-73

probability from, 70

Oliver's Blood, 71-73

101 Bowls Problem, 34-37

Euro Problem contrasted, 52

one-armed bandits (see Bayesian Bandit strategy)

Ortiz-Ospina, Esteban, 276

outcomes

Dice Problem, 22

distribution as set of possible, 29

outcomes appearing more than once, 30

probability mass functions, 29

sequence of possible outcomes, 30

impossible outcomes, 40

outer operations, 145

comparison operators, 146

height arrays, 151

outer product, 145

joint distribution construction, 148

outer sum, 146

P

P(A), 8

P(A and B), 9

P(A | B), 9

P(B | A) to P(A | B) via Bayes's theorem, 19

p-values, 134, 175

pandas

Bayes table in DataFrame

three hypotheses, 22

two hypotheses, 20-22

data held by DataFrame, 2

distribution as mix of distributions, 94

gluten sensitivity Inverse Problem, 78

light bulb lifetime data, 199

penguin data, 161, 163

reading ability improvement, 176
summing row of DataFrame, 95
transposing rows and columns, 95
DataFrame converted from Series, 214
DataFrame converted to Series, 202, 213
installation, xi
joint distribution in DataFrame, 149
 converting to Series, 202
 3-dimensional joint distribution, 217
NumPy array from DataFrame, 149
outer product of DataFrame, 146
read .csv file of data
 light bulb lifetime data, 199
 penguin data, 161
 Price Is Right Problem, 114
 reading ability, 176
 snow amounts, 242
Series
 Boolean values, 3
 (see also Boolean Series)
 cumsum results, 84
 DataFrame converted from, 214
 DataFrame converted to, 202, 213
 DataFrame.sum function, 149
 MAP computation, 37
 MultiIndex, 203, 213
 Pmf class, 31
penguin data classification
 about, 161
 cumulative distribution functions, 162
 data description, 161
 data source, 161
 joint distributions, 168
 scatter plot, 168
 scatter plot compared to contours, 169
 less naive Bayesian classifier, 172
 loading into DataFrame, 161
 multivariate normal distribution, 170
 normal models, 163, 169
 update, 164-168
 naive Bayesian classification, 166
percentiles
 marathon world record, 254
 summarizing posterior distribution, 63
 quantiles versus, 64
physical quantities as random, 53
plotting
 distribution of differences as noisy, 182
 joint distribution, 149
contour plot, 150, 214
posterior distribution, 279
 Grizzly Bear, 210
 Grizzly Bear two-parameter model, 214
scatter plot of penguin data, 168
 contours of joint distribution compared, 169
Weibull distribution, 193
 incomplete data, 197
plus (+) versus Pmf.add_dist(), 93
Pmf objects
 about, 29, 31, 33
 add_dist function, 75
 plus (+) operator versus, 93
 binomial likelihood function, 51, 129
 Cdf object conversion, 86
 coin toss, 30
 coin tossed twice, 44
Cookie Problem, 32-34
 101 bowls of cookies, 34-37
 updated data, 33
credible_interval function, 64
Dice Problem, 38-39
 6-sided best three of four rolls, 86
 updating dice, 39
distribution as mix of distributions, 90-93
 general solution, 93-96
distribution of differences, 181
 plotting, 182
distribution of sums of two dice, 73-76
empiricaldist library for Pmf class, 30
joint distribution construction, 148
light bulb lifetimes, 199
loop iterator items(), 64
marginal function, 219
maximum posterior probability, 37
 coin tossed twice, 45
mean of posterior distribution, 59
MultiIndex, 213, 218, 229
normal distribution of penguin data, 169
normalize function, 32, 48
outcomes appearing more than once, 30
percentile rank, 63
Poisson distribution, 100
posterior predictive distribution, 106
probability densities as normal distribution, 147
probability of superiority, 105, 181
probability that threshold exceeded, 46

prob_gt function, 106, 181
sequence of possible outcomes, 30
triangle-shaped prior, 49
uniform prior for reading ability, 177
point estimates from non-Bayesian logistic regression, 228
Poisson distribution, 100, 103
gamma distribution as conjugate prior, 258
Poisson processes
about, 99
exponential distribution, 108
gamma distribution, 101
Poisson distribution, 100, 103
gamma distribution as conjugate prior, 258
poisson object in SciPy, 100
probability of superiority, 105
update, 103
posterior distribution, 33, 35
Bayesian update, 105
Euro Problem
Bayesian estimation, 49, 51, 130
cumulative distribution function, 83
gluten sensitivity Inverse Problem, 79
joint posterior distribution, 152
posterior distributions from, 153
mean of, 59, 62
parameter meanings, 261
percentiles to summarize, 63
plotting, 279
Grizzly Bear, 210
Grizzly Bear two-parameter model, 214
posterior predictive distribution
light bulb lifetime, 202
marathon world record, 254
Space Shuttle O-ring damage, 235-237
World Cup Problem, 106
sensitivity to the prior, 60
slot machine selection, 140
posterior probability, 19
Bayes factor reported instead of, 134
Bayes tables
three hypotheses, 22-25
two hypotheses, 20-22
conditional posteriors, 156
MAP as highest, 37
coin tossed twice, 45
computing, 37
Pmf
Dice Problem, 38
101 hypotheses, 34-37
two hypotheses, 32
two hypotheses, updated data, 33
posterior distribution, 33, 35
(see also posterior distribution)
posterior mean, 59, 62
Bayesian updates and, 105
distribution skew, 210
joint distributions, 201
power law prior, 61
sensitivity to prior, 60
posterior odds, 71, 72
subjective, 52
Train Problem, 59
power law prior, 62
unnormalized, 21, 32
normalization, 22, 32, 152
power law prior, 61
predictive distributions
posterior
light bulb lifetime, 202
marathon world record, 254
Space Shuttle O-ring damage, 235-237
World Cup Problem, 106
prior
World Cup Problem, 270
Price Is Right Problem
about, 113
decision analysis, 122
distribution of errors, 116-118
kernel density estimation, 115, 116
maximizing expected gain, 124
modeling, 116
prior, 114
probability of winning, 120
update, 118
prior distribution, 32, 34
Bayesian logistic regression, 229
classification of penguin data, 163
different lengths for snow amounts, 245
Empirical Bayes method, 237
informative prior, 65
Pmf for Dice Problem, 38
Price Is Right Problem, 114
kernel density estimation, 115
prior predictive distribution, 270
reading ability improvement, 177
uninformative prior, 65

prior probability, 19
Bayes tables
 three hypotheses, 22-25
 two hypotheses, 20-22
Euro versus 101 Bowls Problems, 52
Pmf
 Dice Problem, 38
 101 hypotheses, 34-37
 two hypotheses, 32
posterior odds, 71, 72
power law prior, 61
prior distribution, 32, 34
 (see also prior distribution)
prior odds, 72
sensitivity to the prior, 60
subjective priors, 52
swamping the priors, 51
Train Problem, 57, 61
triangle-shaped prior, 49
 triangle-shaped bias, 133
uniform prior
 Bayesian Bandit strategy, 135
 beta distribution, 262
 Euro Problem, 47, 51, 129
 gluten sensitivity Inverse Problem, 79
 reading ability improvement, 177
 Train Problem, 58

probability
 counting to compute, 2, 294
 dataset size, 184
 defining, 2
 log odds converted to, 228
 mathematical notation for, 8
 Bayes's theorem, 10, 70
 conditional probability and conjunctions, 9
 conjunctions as commutative, 10
 law of total probability, 11
 power law, 61
 regression model, 243
 odds as degree of certainty, 69
 (see also odds)
 probability from, 70, 228
 probability function returning, 4-5
 conditional probability function, 7
 random versus nonrandom quantities, 52
 Bayesian interpretation of random, 53

probability densities, 102, 109, 147
probability density function (PDF)

Bayesian updates, 260
gamma distribution, 102
norm object returning, 164
reading ability improvement, 179
SciPy pdf function, 147
time between goals, 108

probability mass functions (PMF)
 about, 29
 Bayesian updates, 260
 CDF conversion, 86
 coin toss, 30
 outcomes appearing more than once, 30
 Pmf class (see Pmf objects)
 sequence of possible outcomes, 30

probability of superiority
 reading ability improvement, 181
 World Cup Problem, 105

proportion estimation (see estimating proportions)

PyMC3 library
 about, 271, 274
 happiness, 276-282
 importing as pm, 272
 inference, 274
 Model object, 272
 multiple regression, 280
 sampling the prior, 272
 simple regression, 278
 World Cup Problem, 270-276

Python
 about running notebooks, x
 Anaconda distribution, xi
 installation, xi

PyMC3 library
 about, 271, 274
 happiness, 276-282
 importing as pm, 272
 inference, 274
 Model object, 272
 multiple regression, 280
 sampling the prior, 272
 simple regression, 278
 World Cup Problem, 270-276

with statement, 272

Q

quantiles
 Cdf objects to compute, 86
 percentiles versus, 64

R

random distributions (see Poisson processes)
random sampling, 2
from a distribution (see MCMC (Markov chain Monte Carlo))
Thompson sampling, 139
random versus nonrandom quantities, 52
physical quantities as random, 53
ratios of probabilities as odds, 69
reading ability improvement
about, 175
data into DataFrame, 176
distribution of differences, 181
likelihood, 178
summary statistics, 184
marginal distributions, 180
comparing, 187
prior distribution, 177
probability of superiority, 181
update, 177-180
summary statistics, 186
regression, 243
(see also linear regression)
PyMC3 library for multiple regression, 280
PyMC3 library for simple regression, 278
SciPy linregress function for simple regression, 278
residuals of regression, 245, 246, 252
resources
Anaconda distribution of Python, xi
book web page, xiii
URL with links to all notebooks, x
Roser, Max, 276
rounding avoided with fractions, 22

S

sampling distribution of the mean, 185
sampling from a distribution (see MCMC (Markov chain Monte Carlo))
SciPy
beta function for beta distribution, 261
binomial function, 44
binomial likelihood function, 51
expit function, 228, 231
exponential distribution, 109
gamma distribution function, 102
hypergeometric distribution function, 208
installation, xi
kernel density estimation, 115, 182

linregress function for simple regression, 278
multinomial function, 212, 219
multivariate_normal function, 170
norm function for normal distribution, 147, 163
pdf function, 147, 164
poisson object, 100
Weibull distribution, 191
sequence of possible outcomes, 30
6-sided dice
best three of four rolls, 86
box of three dice, 22
Pmf to solve, 38-39
distribution as mix of distributions, 90-93
general solution, 93-96
slot machines (see Bayesian Bandit strategy)
snow amounts
about, 241
fond memories of, 250
least squares regression, 244
likelihood, 246
marginal distributions, 247
normal distribution assumption, 243
priors, 245
regression model, 243
update, 247
soccer goal scoring (see World Cup Problem)
Space Shuttle Problem
about, 226
logistic regression, Bayesian, 229-232
logistic regression, non-Bayesian, 228
modeling, 237
logistic model, 227, 237
predictions about O-ring damage, 235-237
spam filters as classification, 161
stack function
converting DataFrame to Series, 202, 213
Pmf with two levels in index, 229
standard deviation
How Tall Is Person A, 147, 156
normal distribution
classification of penguin data, 163
How Tall Is Person A, 147
multivariate, 170
Price Is Right Problem, 117
snow amounts, 243
univariate, 170

- Pmf approximating normal distribution, 169
reading ability improvement, 179, 184, 186
residuals as estimate of sigma, 245, 252
snow amounts, 243-248
 sigma as nuisance parameter, 248
statistical versus Bayesian hypothesis testing, 134
statistical versus Bayesian inference, 175
statistics (Bayesian) versus Bayes's theorem, 53
statsmodels for non-Bayesian logistic regression, 228
 Empirical Bayes method, 237
 least squares regression
 marathon world record, 251
 snow amounts, 244
Student's t-test, 175
summary statistics
 about, 184
 larger datasets, 184, 247
 likelihood of reading improvement, 184
 sampling distribution of the mean, 185
 update of reading improvement, 186
 normal distribution assumption, 188
summing row of DataFrame, 95
sums as distributions
 probability of superiority, 105
 three dice, 75, 86
 two dice, 73-76
 weighted sum of probabilities, 122
survival analysis
 about, 191
 incomplete data, 194-196
 called censored, 195
 using, 196-199
light bulb dead bulb prediction, 202
light bulb lifetime distribution, 199-202
Weibull distribution, 191-194
swamping the priors, 51
- T**
testers finding bugs in program (see Lincoln Index Problem)
testing hypotheses
 Bayesian Bandit strategy, 140
 Bayesian versus statistical, 134
 decision analysis instead, 134
 Euro Problem
 about, 129
- binomial distribution, 46, 130
modeling, 131
modeling triangle-shaped bias, 133
modeling uniform bias, 132
solution review, 129-131
statistical versus Bayesian, 134
theoretical versus empirical distributions, 29
The Theory That Would Not Die (McGrayne), 53
Thompson sampling, 139
time delta from Timestamps, 251
Timestamp objects, 251
total probability of the data, 19
 normalizing constant, 22
Train Problem, 57-60
 credible intervals, 63
 power law prior, 61
 sensitivity to the prior, 60
transposing a matrix, 95
triangle-shaped prior, 49
True value summed, 3
- U**
uniform prior
 Bayesian Bandit strategy, 135
 beta distribution, 262
 Euro Problem, 47, 129
 101 Bowls Problem contrasted, 52
 gluten sensitivity Inverse Problem, 79
 reading ability improvement, 177
 Train Problem, 58
univariate normal distribution, 170
universe age, 53
unstack function converting Series to Data-Frame, 214, 232
updates (see Bayesian updates)
urn problem, 17
- V**
variances in covariance matrix, 170
visualizing (see plotting)
- W**
weather data, 241
 snow amounts, 241-250
Weibull distribution, 191-194
 light bulb dead bulb prediction, 202
weighted distributions, 94

weighted sum of probabilities, 122

World Cup Problem

conjugate priors

gamma distribution for update, 258-261

Poisson processes solution review, 257

MCMC via PyMC3, 270-276

Poisson processes

goal-scoring rate, 101

number of goals given rate, 100, 103

poisson object in SciPy, 100

Poisson processes, 99

predicting rematch, 106

probability of superiority, 105

time between goals, 108

update, 103

World Happiness Report, 276

Y

yeast cells counted (see counting cells)

About the Author

Allen B. Downey is a Professor of Computer Science at Olin College of Engineering. He has taught computer science at Wellesley College, Colby College and UC Berkeley. He has a PhD in Computer Science from UC Berkeley and master's and bachelor's degrees from MIT. He is the author of *Think Python*, *Think Stats*, *Think DSP*, and a blog, *Probably Overthinking It*.

Colophon

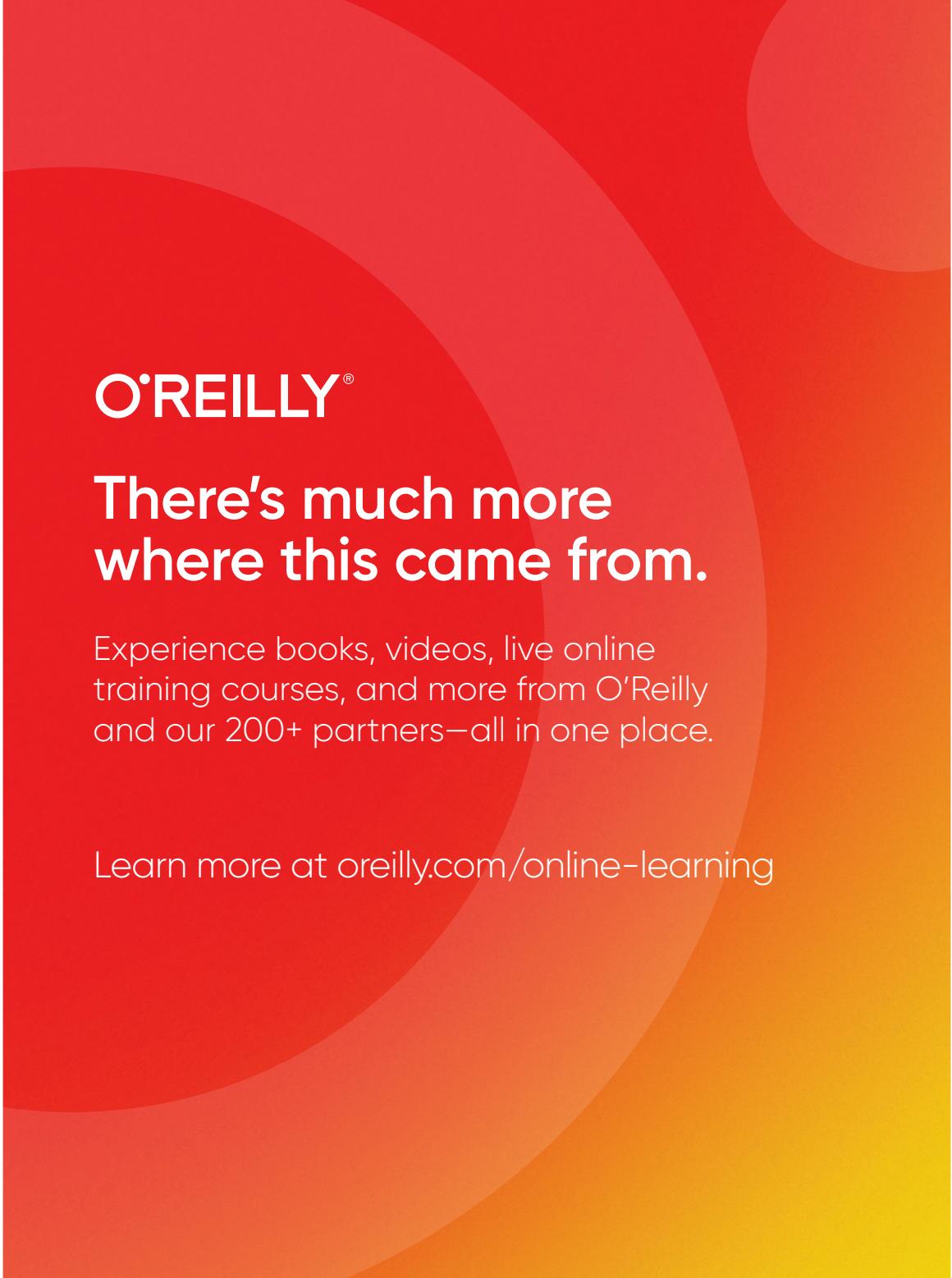
The animal on the cover of *Think Bayes* is a red striped mullet (*Mullus surmuletus*). This species of goatfish can be found in the Mediterranean Sea, east North Atlantic Ocean, and the Black Sea. Known for its distinct striped first dorsal fin, the red striped mullet is a favored delicacy in the Mediterranean—along with a related goatfish, *Mullus barbatus*, which has a first dorsal fin that is not striped. However, the red striped mullet tends to be more prized and is said to taste similar to oysters.

There are stories of ancient Romans rearing the red striped mullet in ponds—attending to, caressing, and even teaching them to feed at the sound of a bell. These fish, generally weighing in under two pounds even when farm-raised, were sometimes sold for their weight in silver.

When left to the wild, red mullets are small bottom-feeding fish with a distinct double beard—known as barbels—on their lower lip, which they use to probe the ocean floor for food. Because the red striped mullet feeds on sandy and rocky bottoms at shallower depths, its barbels are less sensitive than its deep water relative, the *Mullus barbatus*.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *Meyers Kleines Lexicon*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



O'REILLY®

There's much more where this came from.

Experience books, videos, live online training courses, and more from O'Reilly and our 200+ partners—all in one place.

Learn more at oreilly.com/online-learning