O'REILLY®

2nd Edition

Think Stats

EXPLORATORY
DATA ANALYSIS



Allen B. Downey

O'REILLY®

Think Stats

If you know how to program, you have the skills to turn data into knowledge using tools of probability and statistics. This concise introduction shows you how to perform statistical analysis computationally, rather than mathematically, with programs written in Python.

By working with a single case study throughout this thoroughly revised book, you'll learn the entire process of exploratory data analysis—from collecting data and generating statistics to identifying patterns and testing hypotheses. You'll explore distributions, rules of probability, visualization, and many other tools and concepts.

New chapters on regression, time series analysis, survival analysis, and analytic methods will enrich your discoveries.

- Develop an understanding of probability and statistics by writing and testing code
- Run experiments to test statistical behavior, such as generating samples from several distributions
- Use simulations to understand concepts that are hard to grasp mathematically
- Import data from most sources with Python, rather than rely on data that's cleaned and formatted for statistics tools
- Use statistical inference to answer questions about real-world data

"This is the most comprehensive introduction to the Python data analysis stack on the market. Practitioners who want to brush up on their technical skills by learning about the tools available for a modern programming language will also benefit from this book. This is an excellent modern statistics textbook."

—Skipper Seabold
author of *StatsModels*

Allen 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 earned a PhD in Computer Science from UC Berkeley, and master's and bachelor's degrees from MIT.

STATISTICS PROGRAMMING

US $34.99 CAN $36.99
ISBN: 978-1-491-90733-7





Twitter: @oreillymedia
facebook.com/oreilly

SECOND EDITION

Think Stats

Allen B. Downey

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Think Stats, Second Edition

by Allen B. Downey

Copyright © 2015 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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Melanie Yarbrough
Copyeditor: Marta Justak
Proofreader: Amanda Kersey
Indexer: Allen B. Downey
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

October 2014: Second Edition

Revision History for the Second Edition:

2014-10-09: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491907337* for release details.

The O'Reilly logo is a registered trademarks of O'Reilly Media, Inc. *Think Stats*, second edition, the cover image of an archerfish, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

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 Stats is available under the *Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License*. The author maintains an online version at *http://thinkstats2.com*.

ISBN: 978-1-491-90733-7

[LSI]

Table of Contents

Preface ix

1. Exploratory Data Analysis 1
A Statistical Approach 2
The National Survey of Family Growth 2
Importing the Data 3
DataFrames 4
Variables 6
Transformation 7
Validation 8
Interpretation 9
Exercises 11
Glossary 12

2. Distributions 15
Representing Histograms 16
Plotting Histograms 16
NSFG Variables 17
Outliers 19
First Babies 20
Summarizing Distributions 22
Variance 23
Effect Size 23
Reporting Results 24
Exercises 25
Glossary 25

3. Probability Mass Functions 27
Pmfs 27

Plotting PMFs 28
Other Visualizations 30
The Class Size Paradox 30
DataFrame Indexing 34
Exercises 35
Glossary 37

4. Cumulative Distribution Functions. 39
The Limits of PMFs 39
Percentiles 40
CDFs 41
Representing CDFs 42
Comparing CDFs 44
Percentile-Based Statistics 44
Random Numbers 45
Comparing Percentile Ranks 47
Exercises 47
Glossary 48

5. Modeling Distributions. 49
The Exponential Distribution 49
The Normal Distribution 52
Normal Probability Plot 54
The lognormal Distribution 55
The Pareto Distribution 57
Generating Random Numbers 60
Why Model? 61
Exercises 61
Glossary 63

6. Probability Density Functions. 65
PDFs 65
Kernel Density Estimation 67
The Distribution Framework 69
Hist Implementation 69
Pmf Implementation 70
Cdf Implementation 71
Moments 72
Skewness 73
Exercises 75
Glossary 77

7. Relationships Between Variables. 79
Scatter Plots 79
Characterizing Relationships 82
Correlation 83
Covariance 84
Pearson's Correlation 85
Nonlinear Relationships 86
Spearman's Rank Correlation 87
Correlation and Causation 88
Exercises 88
Glossary 89

8. Estimation. 91
The Estimation Game 91
Guess the Variance 93
Sampling Distributions 94
Sampling Bias 97
Exponential Distributions 98
Exercises 99
Glossary 100

9. Hypothesis Testing. 101
Classical Hypothesis Testing 101
HypothesisTest 102
Testing a Difference in Means 104
Other Test Statistics 105
Testing a Correlation 107
Testing Proportions 108
Chi-Squared Tests 109
First Babies Again 110
Errors 111
Power 112
Replication 113
Exercises 114
Glossary 114

10. Linear Least Squares. 117
Least Squares Fit 117
Implementation 118
Residuals 119
Estimation 120
Goodness of Fit 122

Testing a Linear Model 124
Weighted Resampling 126
Exercises 127
Glossary 128

11. Regression. 129
StatsModels 130
Multiple Regression 131
Nonlinear Relationships 133
Data Mining 134
Prediction 135
Logistic Regression 137
Estimating Parameters 139
Implementation 140
Accuracy 141
Exercises 142
Glossary 143

12. Time Series Analysis. 145
Importing and Cleaning 145
Plotting 147
Linear Regression 148
Moving Averages 151
Missing Values 153
Serial Correlation 153
Autocorrelation 155
Prediction 157
Further Reading 161
Exercises 161
Glossary 162

13. Survival Analysis. 165
Survival Curves 165
Hazard Function 167
Estimating Survival Curves 168
Kaplan-Meier Estimation 169
The Marriage Curve 170
Estimating the Survival Function 171
Confidence Intervals 172
Cohort Effects 173
Extrapolation 176
Expected Remaining Lifetime 178

Exercises 180
Glossary 181

14. Analytic Methods. 183
Normal Distributions 183
Sampling Distributions 184
Representing Normal Distributions 185
Central Limit Theorem 186
Testing the CLT 187
Applying the CLT 190
Correlation Test 191
Chi-Squared Test 193
Discussion 194
Exercises 195

Index. 197

Preface

This book is an introduction to the practical tools of exploratory data analysis. The organization of the book follows the process I use when I start working with a dataset:

- Importing and cleaning: Whatever format the data is in, it usually takes some time and effort to read the data, clean and transform it, and check that everything made it through the translation process intact.
- Single variable explorations: I usually start by examining one variable at a time, finding out what the variables mean, looking at distributions of the values, and choosing appropriate summary statistics.
- Pair-wise explorations: To identify possible relationships between variables, I look at tables and scatter plots, and compute correlations and linear fits.
- Multivariate analysis: If there are apparent relationships between variables, I use multiple regression to add control variables and investigate more complex relationships.
- Estimation and hypothesis testing: When reporting statistical results, it is important to answer three questions: How big is the effect? How much variability should we expect if we run the same measurement again? Is it possible that the apparent effect is due to chance?
- Visualization: During exploration, visualization is an important tool for finding possible relationships and effects. Then if an apparent effect holds up to scrutiny, visualization is an effective way to communicate results.

This book takes a computational approach, which has several advantages over mathematical approaches:

- I present most ideas using Python code, rather than mathematical notation. In general, Python code is more readable; also, because it is executable, readers can download it, run it, and modify it.

- Each chapter includes exercises readers can do to develop and solidify their learning. When you write programs, you express your understanding in code; while you are debugging the program, you are also correcting your understanding.
- Some exercises involve experiments to test statistical behavior. For example, you can explore the Central Limit Theorem (CLT) by generating random samples and computing their sums. The resulting visualizations demonstrate why the CLT works and when it doesn't.
- Some ideas that are hard to grasp mathematically are easy to understand by simulation. For example, we approximate p-values by running random simulations, which reinforces the meaning of the p-value.
- Because the book is based on a general-purpose programming language (Python), readers can import data from almost any source. They are not limited to datasets that have been cleaned and formatted for a particular statistics tool.

The book lends itself to a project-based approach. In my class, students work on a semester-long project that requires them to pose a statistical question, find a dataset that can address it, and apply each of the techniques they learn to their own data.

To demonstrate my approach to statistical analysis, the book presents a case study that runs through all of the chapters. It uses data from two sources:

- The National Survey of Family Growth (NSFG), conducted by the U.S. Centers for Disease Control and Prevention (CDC) to gather "information on family life, marriage and divorce, pregnancy, infertility, use of contraception, and men's and women's health." (See *http://cdc.gov/nchs/nsfg.htm*.)
- The Behavioral Risk Factor Surveillance System (BRFSS), conducted by the National Center for Chronic Disease Prevention and Health Promotion to "track health conditions and risk behaviors in the United States." (See *http://cdc.gov/BRFSS/*.)

Other examples use data from the IRS, the U.S. Census, and the Boston Marathon.

This second edition of *Think Stats* includes the chapters from the first edition, many of them substantially revised, and new chapters on regression, time series analysis, survival analysis, and analytic methods. The previous edition did not use pandas, SciPy, or StatsModels, so all of that material is new.

How I Wrote This Book

When people write a new textbook, they usually start by reading a stack of old textbooks. As a result, most books contain the same material in pretty much the same order.

I did not do that. In fact, I used almost no printed material while I was writing this book, for several reasons:

- My goal was to explore a new approach to this material, so I didn't want much exposure to existing approaches.
- Since I am making this book available under a free license, I wanted to make sure that no part of it was encumbered by copyright restrictions.
- Many readers of my books don't have access to libraries of printed material, so I tried to make references to resources that are freely available on the Internet.
- Some proponents of old media think that the exclusive use of electronic resources is lazy and unreliable. They might be right about the first part, but I think they are wrong about the second, so I wanted to test my theory.

The resource I used more than any other is Wikipedia. In general, the articles I read on statistical topics were very good (although I made a few small changes along the way). I include references to Wikipedia pages throughout the book and I encourage you to follow those links; in many cases, the Wikipedia page picks up where my description leaves off. The vocabulary and notation in this book are generally consistent with Wikipedia, unless I had a good reason to deviate. Other resources I found useful were Wolfram MathWorld and the Reddit statistics forum.

Using the Code

The code and data used in this book are available from GitHub. Git is a version control system that allows you to keep track of the files that make up a project. A collection of files under Git's control is called a *repository*. GitHub is a hosting service that provides storage for Git repositories and a convenient web interface.

The GitHub homepage for my repository provides several ways to work with the code:

- You can create a copy of my repository on GitHub by pressing the Fork button. If you don't already have a GitHub account, you'll need to create one. After forking, you'll have your own repository on GitHub that you can use to keep track of code you write while working on this book. Then you can clone the repo, which means that you make a copy of the files on your computer.
- Or you could clone my repository. You don't need a GitHub account to do this, but you won't be able to write your changes back to GitHub.
- If you don't want to use Git at all, you can download the files in a Zip file using the button in the lower-right corner of the GitHub page.

All of the code is written to work in both Python 2 and Python 3 with no translation.

I developed this book using Anaconda from Continuum Analytics, which is a free Python distribution that includes all the packages you'll need to run the code (and lots more). I found Anaconda easy to install. By default it does a user-level installation, not system-level, so you don't need administrative privileges. And it supports both Python 2 and Python 3. You can download Anaconda from Continuum.

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

- pandas for representing and analyzing data
- NumPy for basic numerical computation
- SciPy for scientific computation including statistics
- StatsModels for regression and other statistical analysis
- matplotlib for visualization

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 strongly recommend using Anaconda or one of the other Python distributions that include these packages.

After you clone the repository or unzip the zip file, you should have a file called `ThinkStats2/code/nsfg.py`. If you run it, it should read a data file, run some tests, and print a message like, "All tests passed." If you get import errors, it probably means there are packages you need to install.

Most exercises use Python scripts, but some also use the IPython notebook. If you have not used IPython notebook before, I suggest you start with the documentation.

I wrote this book assuming that the reader is familiar with core Python, including object-oriented features, but not pandas, NumPy, and SciPy. If you are already familiar with these modules, you can skip a few sections.

I assume that the reader knows basic mathematics, including logarithms, for example, and summations. I refer to calculus concepts in a few places, but you don't have to do any calculus.

If you have never studied statistics, I think this book is a good place to start. And if you have taken a traditional statistics class, I hope this book will help repair the damage.

—

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

Contributor List

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

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

- Lisa Downey and June Downey read an early draft and made many corrections and suggestions.
- Steven Zhang found several errors.
- Andy Pethan and Molly Farison helped debug some of the solutions, and Molly spotted several typos.
- Andrew Heine found an error in my error function.
- Dr. Nikolas Akerblom knows how big a Hyracotherium is.
- Alex Morrow clarified one of the code examples.
- Jonathan Street caught an error in the nick of time.
- Gábor Lipták found a typo in the book and the relay race solution.
- Many thanks to Kevin Smith and Tim Arnold for their work on plasTeX, which I used to convert this book to DocBook.
- George Caplan sent several suggestions for improving clarity.
- Julian Ceipek found an error and a number of typos.
- Stijn Debrouwere, Leo Marihart III, Jonathan Hammler, and Kent Johnson found errors in the first print edition.
- Dan Kearney found a typo.
- Jeff Pickhardt found a broken link and a typo.
- Jörg Beyer found typos in the book and made many corrections in the docstrings of the accompanying code.
- Tommie Gannert sent a patch file with a number of corrections.
- Alexander Gryzlov suggested a clarification in an exercise.
- Martin Veillette reported an error in one of the formulas for Pearson's correlation.
- Christoph Lendenmann submitted several errata.
- Haitao Ma noticed a typo and and sent me a note.
- Michael Kearney sent me many excellent suggestions.

- Alex Birch made a number of helpful suggestions.
- Lindsey Vanderlyn, Griffin Tschurwald, and Ben Small read an early version of this book and found many errors.
- John Roth, Carol Willing, and Carol Novitsky performed technical reviews of the book. They found many errors and made many helpful suggestions.
- Rohit Deshpande found a typesetting error.
- David Palmer sent many helpful suggestions and corrections.
- Erik Kulyk found many typos.

Safari® Books Online



Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, and education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

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 *http://bit.ly/think_stats_2e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1
Exploratory Data Analysis

The thesis of this book is that data combined with practical methods can answer questions and guide decisions under uncertainty.

As an example, I present a case study motivated by a question I heard when my wife and I were expecting our first child: do first babies tend to arrive late?

If you Google this question, you will find plenty of discussion. Some people claim it's true, others say it's a myth, and some people say it's the other way around: first babies come early.

In many of these discussions, people provide data to support their claims. I found many examples like these:

> "My two friends that have given birth recently to their first babies, BOTH went almost 2 weeks overdue before going into labour or being induced."
>
> "My first one came 2 weeks late and now I think the second one is going to come out two weeks early!!"
>
> "I don't think that can be true because my sister was my mother's first and she was early, as with many of my cousins."

Reports like these are called **anecdotal evidence** because they are based on data that is unpublished and usually personal. In casual conversation, there is nothing wrong with anecdotes, so I don't mean to pick on the people I quoted.

But we might want evidence that is more persuasive and an answer that is more reliable. By those standards, anecdotal evidence usually fails, because:

Small number of observations
: If pregnancy length is longer for first babies, the difference is probably small compared to natural variation. In that case, we might have to compare a large number of pregnancies to be sure that a difference exists.

Selection bias
: People who join a discussion of this question might be interested because their first babies were late. In that case the process of selecting data would bias the results.

Confirmation bias
: People who believe the claim might be more likely to contribute examples that confirm it. People who doubt the claim are more likely to cite counterexamples.

Inaccuracy
: Anecdotes are often personal stories, and often misremembered, misrepresented, repeated inaccurately, etc.

So how can we do better?

A Statistical Approach

To address the limitations of anecdotes, we will use the tools of statistics, which include:

Data collection
: We will use data from a large national survey that was designed explicitly with the goal of generating statistically valid inferences about the U.S. population.

Descriptive statistics
: We will generate statistics that summarize the data concisely, and evaluate different ways to visualize data.

Exploratory data analysis
: We will look for patterns, differences, and other features that address the questions we are interested in. At the same time we will check for inconsistencies and identify limitations.

Estimation
: We will use data from a sample to estimate characteristics of the general population.

Hypothesis testing
: Where we see apparent effects, like a difference between two groups, we will evaluate whether the effect might have happened by chance.

By performing these steps with care to avoid pitfalls, we can reach conclusions that are more justifiable and more likely to be correct.

The National Survey of Family Growth

Since 1973, the US Centers for Disease Control and Prevention (CDC) have conducted the National Survey of Family Growth (NSFG), which is intended to gather "information on family life, marriage and divorce, pregnancy, infertility, use of contraception, and men's and women's health. The survey results are used…to plan health services and

health education programs, and to do statistical studies of families, fertility, and health."

We will use data collected by this survey to investigate whether first babies tend to come late, as well as answer other questions. In order to use this data effectively, we have to understand the design of the study.

The NSFG is a **cross-sectional** study, which means that it captures a snapshot of a group at a point in time. The most common alternative is a **longitudinal** study, which observes a group repeatedly over a period of time.

The NSFG has been conducted seven times; each deployment is called a **cycle**. We will use data from Cycle 6, which was conducted from January 2002 to March 2003.

The goal of the survey is to draw conclusions about a **population**; the target population of the NSFG is people in the United States aged 15-44. Ideally surveys would collect data from every member of the population, but that's seldom possible. Instead we collect data from a subset of the population called a **sample**. The people who participate in a survey are called **respondents**.

In general, cross-sectional studies are meant to be **representative**, which means that every member of the target population has an equal chance of participating. That ideal is hard to achieve in practice, but people who conduct surveys come as close as they can.

The NSFG is not representative; instead it is deliberately **oversampled**. The designers of the study recruited three groups—Hispanics, African Americans and teenagers—at rates higher than their representation in the U.S. population, in order to make sure that the number of respondents in each of these groups is large enough to draw valid statistical inferences.

Of course, the drawback of oversampling is that it is not as easy to draw conclusions about the general population based on statistics from the survey. We will come back to this point later.

When working with this kind of data, it is important to be familiar with the **codebook**, which documents the design of the study, the survey questions, and the encoding of the responses. The codebook and user's guide for the NSFG data are available from the CDC's website.

Importing the Data

The code and data used in this book are available from GitHub. For information about downloading and working with this code, see "Using the Code" on page xi.

Once you download the code, you should have a folder called `ThinkStats2/code` with a file called `nsfg.py`. If you run it, it should read a data file, run some tests, and print a message like, "All tests passed."

Let's see what it does. Pregnancy data from Cycle 6 of the NSFG is in a file called `2002FemPreg.dat.gz`; it is a gzip-compressed data file in plain text (ASCII), with fixed width columns. Each line in the file is a **record** that contains data about one pregnancy.

The format of the file is documented in `2002FemPreg.dct`, which is a Stata dictionary file. Stata is a statistical software system; a "dictionary" in this context is a list of variable names, types, and indices that identify where in each line to find each variable.

For example, here are a few lines from `2002FemPreg.dct`:

```
infile dictionary {
  _column(1)  str12  caseid    %12s  "RESPONDENT ID NUMBER"
  _column(13) byte   pregordr   %2f  "PREGNANCY ORDER (NUMBER)"
}
```

This dictionary describes two variables: `caseid` is a 12-character string that represents the respondent ID; `pregorder` is a one-byte integer that indicates which pregnancy this record describes for this respondent.

The code you downloaded includes `thinkstats2.py` which is a Python module that contains many classes and functions used in this book, including functions that read the Stata dictionary and the NSFG data file. Here's how they are used in `nsfg.py`:

```
def ReadFemPreg(dct_file='2002FemPreg.dct',
                dat_file='2002FemPreg.dat.gz'):
    dct = thinkstats2.ReadStataDct(dct_file)
    df = dct.ReadFixedWidth(dat_file, compression='gzip')
    CleanFemPreg(df)
    return df
```

`ReadStataDct` takes the name of the dictionary file and returns `dct`, a `FixedWidthVariables` object that contains the information from the dictionary file. `dct` provides `ReadFixedWidth`, which reads the data file.

DataFrames

The result of `ReadFixedWidth` is a DataFrame, which is the fundamental data structure provided by pandas, which is a Python data and statistics package we'll use throughout this book. A DataFrame contains a row for each record, in this case one row per pregnancy, and a column for each variable.

In addition to the data, a DataFrame also contains the variable names and their types, and it provides methods for accessing and modifying the data.

If you print `df` you get a truncated view of the rows and columns, and the shape of the DataFrame, which is 13593 rows/records and 244 columns/variables.

```
>>> import nsfg
>>> df = nsfg.ReadFemPreg()
>>> df
...
[13593 rows x 244 columns]
```

The attribute `columns` returns a sequence of column names as Unicode strings:

```
>>> df.columns
Index([u'caseid', u'pregordr', u'howpreg_n', u'howpreg_p', ... ])
```

The result is an Index, which is another pandas data structure. We'll learn more about Index later, but for now we'll treat it like a list:

```
>>> df.columns[1]
'pregordr'
```

To access a column from a DataFrame, you can use the column name as a key:

```
>>> pregordr = df['pregordr']
>>> type(pregordr)
<class 'pandas.core.series.Series'>
```

The result is a Series, yet another pandas data structure. A Series is like a Python list with some additional features. When you print a Series, you get the indices and the corresponding values:

```
>>> pregordr
0     1
1     2
2     1
3     2
...
13590    3
13591    4
13592    5
Name: pregordr, Length: 13593, dtype: int64
```

In this example the indices are integers from 0 to 13592, but in general they can be any sortable type. The elements are also integers, but they can be any type.

The last line includes the variable name, Series length, and data type; `int64` is one of the types provided by NumPy. If you run this example on a 32-bit machine you might see `int32`.

You can access the elements of a Series using integer indices and slices:

```
>>> pregordr[0]
1
>>> pregordr[2:5]
2    1
3    2
4    3
Name: pregordr, dtype: int64
```

The result of the index operator is an `int64`; the result of the slice is another Series.

You can also access the columns of a DataFrame using dot notation:

```
>>> pregordr = df.pregordr
```

This notation only works if the column name is a valid Python identifier, so it has to begin with a letter, can't contain spaces, etc.

Variables

We have already seen two variables in the NSFG dataset, `caseid` and `pregordr`, and we have seen that there are 244 variables in total. For the explorations in this book, I use the following variables:

- `caseid` is the integer ID of the respondent.
- `prglength` is the integer duration of the pregnancy in weeks.
- `outcome` is an integer code for the outcome of the pregnancy. The code 1 indicates a live birth.
- `pregordr` is a pregnancy serial number; for example, the code for a respondent's first pregnancy is 1, for the second pregnancy is 2, and so on.
- `birthord` is a serial number for live births; the code for a respondent's first child is 1, and so on. For outcomes other than live births, this field is blank.
- `birthwgt_lb` and `birthwgt_oz` contain the pounds and ounces parts of the birth weight of the baby.
- `agepreg` is the mother's age at the end of the pregnancy.
- `finalwgt` is the statistical weight associated with the respondent. It is a floating-point value that indicates the number of people in the U.S. population this respondent represents.

If you read the codebook carefully, you will see that many of the variables are **recodes**, which means that they are not part of the **raw data** collected by the survey; they are calculated using the raw data.

For example, `prglngth` for live births is equal to the raw variable `wksgest` (weeks of gestation) if it is available; otherwise it is estimated using `mosgest * 4.33` (months of gestation times the average number of weeks in a month).

Recodes are often based on logic that checks the consistency and accuracy of the data. In general it is a good idea to use recodes when they are available, unless there is a compelling reason to process the raw data yourself.

Transformation

When you import data like this, you often have to check for errors, deal with special values, convert data into different formats, and perform calculations. These operations are called **data cleaning**.

`nsfg.py` includes `CleanFemPreg`, a function that cleans the variables I am planning to use.

```
def CleanFemPreg(df):
    df.agepreg /= 100.0

    na_vals = [97, 98, 99]
    df.birthwgt_lb.replace(na_vals, np.nan, inplace=True)
    df.birthwgt_oz.replace(na_vals, np.nan, inplace=True)

    df['totalwgt_lb'] = df.birthwgt_lb + df.birthwgt_oz / 16.0
```

`agepreg` contains the mother's age at the end of the pregnancy. In the data file, `agepreg` is encoded as an integer number of centiyears. So the first line divides each element of `agepreg` by 100, yielding a floating-point value in years.

`birthwgt_lb` and `birthwgt_oz` contain the weight of the baby, in pounds and ounces, for pregnancies that end in live births. In addition they use several special codes:

```
97 NOT ASCERTAINED
98 REFUSED
99 DON'T KNOW
```

Special values encoded as numbers are *dangerous* because if they are not handled properly, they can generate bogus results, like a 99-pound baby. The `replace` method replaces these values with `np.nan`, a special floating-point value that represents "not a number." The `inplace` flag tells `replace` to modify the existing Series rather than create a new one.

As part of the IEEE floating-point standard, all mathematical operations return `nan` if either argument is `nan`:

```
>>> import numpy as np
>>> np.nan / 100.0
nan
```

So computations with nan tend to do the right thing, and most pandas functions handle nan appropriately. But dealing with missing data will be a recurring issue.

The last line of CleanFemPreg creates a new column totalwgt_lb that combines pounds and ounces into a single quantity, in pounds.

One important note: when you add a new column to a DataFrame, you must use dictionary syntax, like this:

```
# CORRECT
df['totalwgt_lb'] = df.birthwgt_lb + df.birthwgt_oz / 16.0
```

Not dot notation, like this:

```
# WRONG!
df.totalwgt_lb = df.birthwgt_lb + df.birthwgt_oz / 16.0
```

The version with dot notation adds an attribute to the DataFrame object, but that attribute is not treated as a new column.

Validation

When data is exported from one software environment and imported into another, errors might be introduced. And when you are getting familiar with a new dataset, you might interpret data incorrectly or introduce other misunderstandings. If you take time to validate the data, you can save time later and avoid errors.

One way to validate data is to compute basic statistics and compare them with published results. For example, the NSFG codebook includes tables that summarize each variable. Here is the table for outcome, which encodes the outcome of each pregnancy:

```
value label           Total
1 LIVE BIRTH             9148
2 INDUCED ABORTION       1862
3 STILLBIRTH              120
4 MISCARRIAGE            1921
5 ECTOPIC PREGNANCY       190
6 CURRENT PREGNANCY       352
```

The Series class provides a method, value_counts, that counts the number of times each value appears. If we select the outcome Series from the DataFrame, we can use value_counts to compare with the published data:

```
>>> df.outcome.value_counts().sort_index()
1    9148
2    1862
3     120
4    1921
5     190
6     352
```

The result of `value_counts` is a Series; `sort_index` sorts the Series by index, so the values appear in order.

Comparing the results with the published table, it looks like the values in `outcome` are correct. Similarly, here is the published table for `birthwgt_lb`

```
value label                  Total
. INAPPLICABLE             4449
0-5 UNDER 6 POUNDS           1125
6 6 POUNDS                 2223
7 7 POUNDS                 3049
8 8 POUNDS                 1889
9-95 9 POUNDS OR MORE         799
```

And here are the value counts:

```
>>> df.birthwgt_lb.value_counts(sort=False)
0        8
1       40
2       53
3       98
4      229
5      697
6     2223
7     3049
8     1889
9      623
10     132
11      26
12      10
13       3
14       3
15       1
51       1
```

The counts for 6, 7, and 8 pounds check out, and if you add up the counts for 0-5 and 9-95, they check out, too. But if you look more closely, you will notice one value that has to be an error, a 51 pound baby!

To deal with this error, I added a line to `CleanFemPreg`:

```
df.birthwgt_lb[df.birthwgt_lb > 20] = np.nan
```

This statement replaces invalid values with `np.nan`. The expression in brackets yields a Series of type `bool`, where True indicates that the condition is true. When a Boolean Series is used as an index, it selects only the elements that satisfy the condition.

Interpretation

To work with data effectively, you have to think on two levels at the same time: the level of statistics and the level of context.

As an example, let's look at the sequence of outcomes for a few respondents. Because of the way the data files are organized, we have to do some processing to collect the pregnancy data for each respondent. Here's a function that does that:

```
def MakePregMap(df):
    d = defaultdict(list)
    for index, caseid in df.caseid.iteritems():
        d[caseid].append(index)
    return d
```

`df` is the DataFrame with pregnancy data. The `iteritems` method enumerates the index (row number) and `caseid` for each pregnancy.

`d` is a dictionary that maps from each case ID to a list of indices. If you are not familiar with `defaultdict`, it is in the Python `collections` module. Using `d`, we can look up a respondent and get the indices of that respondent's pregnancies.

This example looks up one respondent and prints a list of outcomes for her pregnancies:

```
>>> caseid = 10229
>>> indices = preg_map[caseid]
>>> df.outcome[indices].values
[4 4 4 4 4 4 1]
```

`indices` is the list of indices for pregnancies corresponding to respondent `10229`.

Using this list as an index into `df.outcome` selects the indicated rows and yields a Series. Instead of printing the whole Series, I selected the `values` attribute, which is a NumPy array.

The outcome code `1` indicates a live birth. Code `4` indicates a miscarriage; that is, a pregnancy that ended spontaneously, usually with no known medical cause.

Statistically this respondent is not unusual. Miscarriages are common and there are other respondents who reported as many or more.

But remembering the context, this data tells the story of a woman who was pregnant six times, each time ending in miscarriage. Her seventh and most recent pregnancy ended in a live birth. If we consider this data with empathy, it is natural to be moved by the story it tells.

Each record in the NSFG dataset represents a person who provided honest answers to many personal and difficult questions. We can use this data to answer statistical questions about family life, reproduction, and health. At the same time, we have an obligation to consider the people represented by the data, and to afford them respect and gratitude.

Exercises

Exercise 1-1.

In the repository you downloaded, you should find a file named `chap01ex.ipynb`, which is an IPython notebook. You can launch IPython notebook from the command line like this:

```
$ ipython notebook &
```

If IPython is installed, it should launch a server that runs in the background and open a browser to view the notebook. If you are not familiar with IPython, I suggest you start at the IPython website.

You can add a command-line option that makes figures appear "inline"; that is, in the notebook rather than a pop-up window:

```
$ ipython notebook --pylab=inline &
```

Open `chap01ex.ipynb`. Some cells are already filled in, and you should execute them. Other cells give you instructions for exercises you should try.

A solution to this exercise is in `chap01soln.ipynb`

Exercise 1-2.

Create a file named `chap01ex.py` and write code that reads the respondent file, `2002FemResp.dat.gz`. You might want to start with a copy of `nsfg.py` and modify it.

The variable `pregnum` is a recode that indicates how many times each respondent has been pregnant. Print the value counts for this variable and compare them to the published results in the NSFG codebook.

You can also cross-validate the respondent and pregnancy files by comparing `pregnum` for each respondent with the number of records in the pregnancy file.

You can use `nsfg.MakePregMap` to make a dictionary that maps from each `caseid` to a list of indices into the pregnancy DataFrame.

A solution to this exercise is in `chap01soln.py`

Exercise 1-3.

The best way to learn about statistics is to work on a project you are interested in. Do you want to investigate a question like "Do first babies arrive late?"

Think about questions you find personally interesting, or items of conventional wisdom, or controversial topics, or questions that have political consequences, and see if you can formulate a question that lends itself to statistical inquiry.

Look for data to help you address the question. Governments are good sources because data from public research is often freely available. Good places to start include *http://www.data.gov/*, and *http://www.science.gov/*, and in the United Kingdom, *http://data.gov.uk/*.

Two of my favorite data sets are the General Social Survey and the European Social Survey.

If it seems like someone has already answered your question, look closely to see whether the answer is justified. There might be flaws in the data or the analysis that make the conclusion unreliable. In that case you could perform a different analysis of the same data, or look for a better source of data.

If you find a published paper that addresses your question, you should be able to get the raw data. Many authors make their data available on the web, but for sensitive data you might have to write to the authors, provide information about how you plan to use the data, or agree to certain terms of use. Be persistent!

Glossary

anecdotal evidence
: Evidence, often personal, that is collected casually rather than by a well-designed study.

population
: A group we are interested in studying. "Population" often refers to a group of people, but the term is used for other subjects, too.

cross-sectional study
: A study that collects data about a population at a particular point in time.

cycle
: In a repeated cross-sectional study, each repetition of the study is called a cycle.

longitudinal study
: A study that follows a population over time, collecting data from the same group repeatedly.

record
: In a dataset, a collection of information about a single person or other subject.

respondent
: A person who responds to a survey.

sample
: The subset of a population used to collect data.

representative
: A sample is representative if every member of the population has the same chance of being in the sample.

oversampling
: The technique of increasing the representation of a subpopulation in order to avoid errors due to small sample sizes.

raw data
: Values collected and recorded with little or no checking, calculation or interpretation.

recode
: A value that is generated by calculation and other logic applied to raw data.

data cleaning
: Processes that include validating data, identifying errors, translating between data types and representations, etc.

CHAPTER 2
Distributions

One of the best ways to describe a variable is to report the values that appear in the dataset and how many times each value appears. This description is called the **distribution** of the variable.

The most common representation of a distribution is a **histogram**, which is a graph that shows the **frequency** of each value. In this context, "frequency" means the number of times the value appears.

In Python, an efficient way to compute frequencies is with a dictionary. Given a sequence of values, `t`:

```
hist = {}
for x in t:
    hist[x] = hist.get(x, 0) + 1
```

The result is a dictionary that maps from values to frequencies. Alternatively, you could use the `Counter` class defined in the `collections` module:

```
from collections import Counter
counter = Counter(t)
```

The result is a `Counter` object, which is a subclass of dictionary.

Another option is to use the pandas method `value_counts`, which we saw in the previous chapter. But for this book I created a class, Hist, that represents histograms and provides the methods that operate on them.

Representing Histograms

The Hist constructor can take a sequence, dictionary, pandas Series, or another Hist. You can instantiate a Hist object like this:

```
>>> import thinkstats2
>>> hist = thinkstats2.Hist([1, 2, 2, 3, 5])
>>> hist
Hist({1: 1, 2: 2, 3: 1, 5: 1})
```

Hist objects provide `Freq`, which takes a value and returns its frequency:

```
>>> hist.Freq(2)
2
```

The bracket operator does the same thing:

```
>>> hist[2]
2
```

If you look up a value that has never appeared, the frequency is 0:

```
>>> hist.Freq(4)
0
```

`Values` returns an unsorted list of the values in the Hist:

```
>>> hist.Values()
[1, 5, 3, 2]
```

To loop through the values in order, you can use the built-in function `sorted`:

```
for val in sorted(hist.Values()):
    print(val, hist.Freq(val))
```

Or you can use `Items` to iterate through value-frequency pairs:

```
for val, freq in hist.Items():
     print(val, freq)
```

Plotting Histograms

For this book I wrote a module called `thinkplot.py` that provides functions for plotting Hists and other objects defined in `thinkstats2.py`. It is based on `pyplot`, which is part of the `matplotlib` package. See "Using the Code" on page xi for information about installing `matplotlib`.

To plot `hist` with `thinkplot`, try this:

```
>>> import thinkplot
>>> thinkplot.Hist(hist)
>>> thinkplot.Show(xlabel='value', ylabel='frequency')
```

You can read the documentation for `thinkplot` at *http://bit.ly/1sgoj7V*.

NSFG Variables

Now let's get back to the data from the NSFG. The code in this chapter is in `first.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

When you start working with a new dataset, I suggest you explore the variables you are planning to use one at a time, and a good way to start is by looking at histograms.

In "Transformation" on page 7 we transformed `agepreg` from centiyears to years, and combined `birthwgt_lb` and `birthwgt_oz` into a single quantity, `totalwgt_lb`. In this section I use these variables to demonstrate some features of histograms.

I'll start by reading the data and selecting records for live births:

```
preg = nsfg.ReadFemPreg()
live = preg[preg.outcome == 1]
```

The expression in brackets is a boolean Series that selects rows from the DataFrame and returns a new DataFrame. Next I generate and plot the histogram of `birthwgt_lb` for live births.

```
hist = thinkstats2.Hist(live.birthwgt_lb, label='birthwgt_lb')
thinkplot.Hist(hist)
thinkplot.Show(xlabel='pounds', ylabel='frequency')
```

When the argument passed to Hist is a pandas Series, any `nan` values are dropped. `label` is a string that appears in the legend when the Hist is plotted.

Figure 2-1 shows the result. The most common value, called the **mode**, is 7 pounds. The distribution is approximately bell-shaped, which is the shape of the **normal** distribution, also called a **Gaussian** distribution. But unlike a true normal distribution, this distribution is asymmetric; it has a **tail** that extends farther to the left than to the right.

Figure 2-2 shows the histogram of `birthwgt_oz`, which is the ounces part of birth weight. In theory we expect this distribution to be **uniform**; that is, all values should have the same frequency. In fact, 0 is more common than the other values, and 1 and 15 are less common, probably because respondents round off birth weights that are close to an integer value.

Figure 2-3 shows the histogram of `agepreg`, the mother's age at the end of pregnancy. The mode is 21 years. The distribution is very roughly bell-shaped, but in this case the tail extends farther to the right than left; most mothers are in their 20s, fewer in their 30s.

Figure 2-4 shows the histogram of `prglngth`, the length of the pregnancy in weeks. By far the most common value is 39 weeks. The left tail is longer than the right; early babies

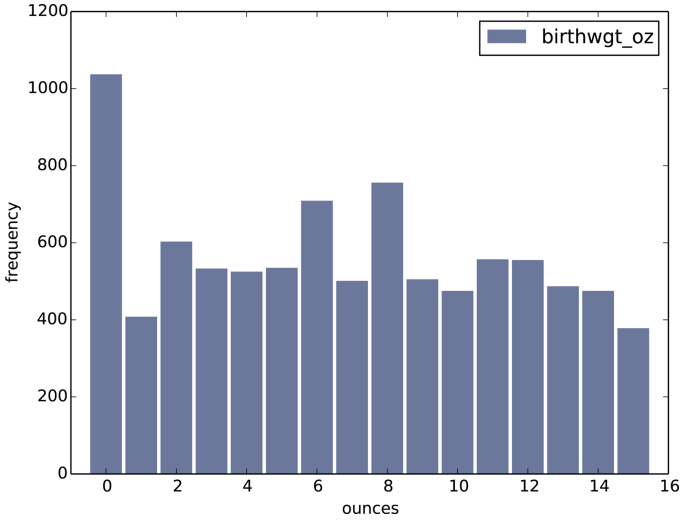


Figure 2-2. Histogram of the ounce part of birth weight

are common, but pregnancies seldom go past 43 weeks, and doctors often intervene if they do.

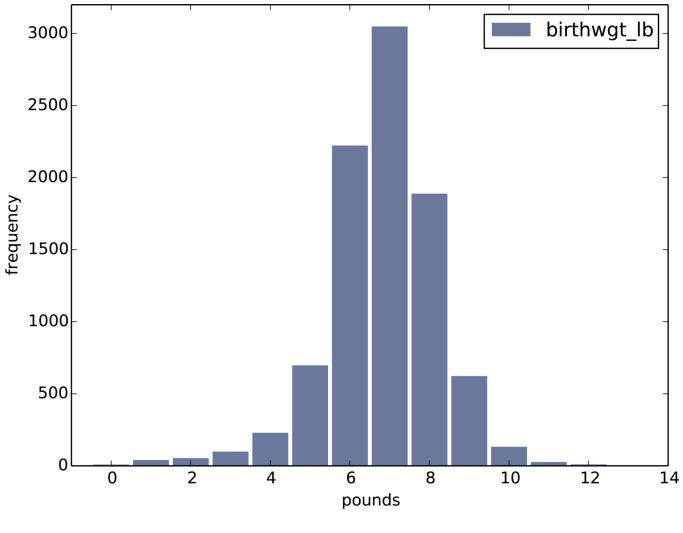


Figure 2-1. Histogram of the pound part of birth weight

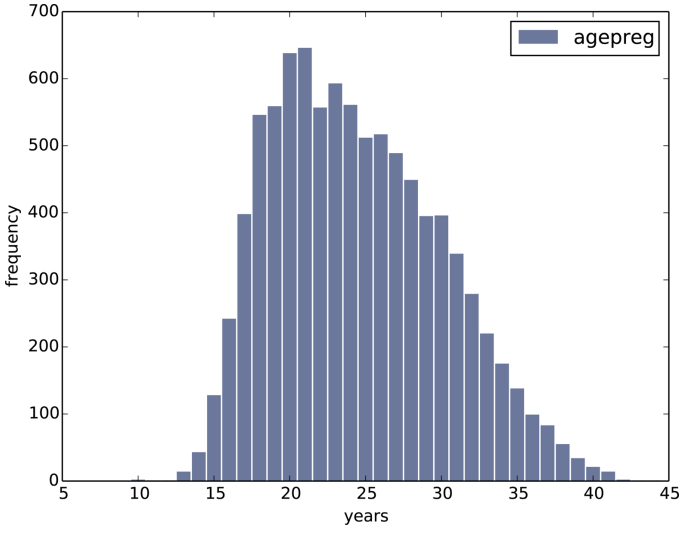


Figure 2-3. Histogram of mother's age at end of pregnancy

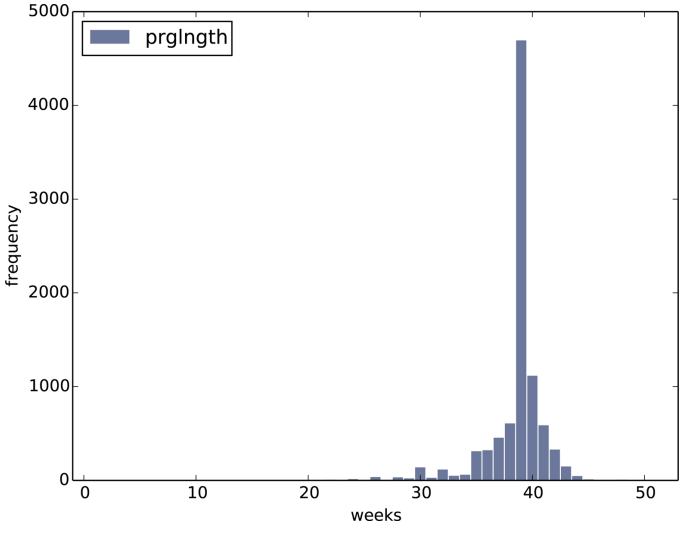


Figure 2-4. Histogram of pregnancy length in weeks

Outliers

Looking at histograms, it is easy to identify the most common values and the shape of the distribution, but rare values are not always visible.

Before going on, it is a good idea to check for **outliers**, which are extreme values that might be errors in measurement and recording, or might be accurate reports of rare events.

Hist provides methods `Largest` and `Smallest`, which take an integer `n` and return the `n` largest or smallest values from the histogram:

```
for weeks, freq in hist.Smallest(10):
    print(weeks, freq)
```

In the list of pregnancy lengths for live births, the 10 lowest values are `[0, 4, 9, 13, 17, 18, 19, 20, 21, 22]`. Values below 10 weeks are certainly errors; the most likely explanation is that the outcome was not coded correctly. Values higher than 30 weeks are probably legitimate. Between 10 and 30 weeks, it is hard to be sure; some values are probably errors, but some represent premature babies.

On the other end of the range, the highest values are:

```
weeks  count
43     148
44     46
45     10
46     1
47     1
48     7
50     2
```

Most doctors recommend induced labor if a pregnancy exceeds 42 weeks, so some of the longer values are surprising. In particular, 50 weeks seems medically unlikely.

The best way to handle outliers depends on "domain knowledge"; that is, information about where the data come from and what they mean. And it depends on what analysis you are planning to perform.

In this example, the motivating question is whether first babies tend to be early (or late). When people ask this question, they are usually interested in full-term pregnancies, so for this analysis I will focus on pregnancies longer than 27 weeks.

First Babies

Now we can compare the distribution of pregnancy lengths for first babies and others. I divided the DataFrame of live births using `birthord`, and computed their histograms:

```
firsts = live[live.birthord == 1]
others = live[live.birthord != 1]

first_hist = thinkstats2.Hist(firsts.prglngth)
other_hist = thinkstats2.Hist(others.prglngth)
```

Then I plotted their histograms on the same axis:

```
width = 0.45
thinkplot.PrePlot(2)
thinkplot.Hist(first_hist, align='right', width=width)
thinkplot.Hist(other_hist, align='left', width=width)
thinkplot.Show(xlabel='weeks', ylabel='frequency')
```

`thinkplot.PrePlot` takes the number of histograms we are planning to plot; it uses this information to choose an appropriate collection of colors.

`thinkplot.Hist` normally uses `align='center'` so that each bar is centered over its value. For this figure, I use `align='right'` and `align='left'` to place corresponding bars on either side of the value.

With `width=0.45`, the total width of the two bars is 0.9, leaving some space between each pair.

Finally, I adjust the axis to show only data between 27 and 46 weeks. Figure 2-5 shows the result.

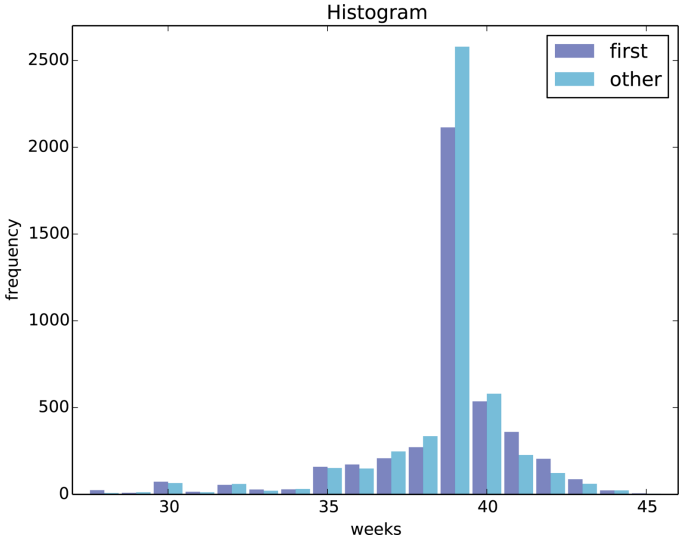


Figure 2-5. Histogram of pregnancy lengths.

Histograms are useful because they make the most frequent values immediately apparent. But they are not the best choice for comparing two distributions. In this example, there are fewer "first babies" than "others," so some of the apparent differences in the histograms are due to sample sizes. In the next chapter we address this problem using probability mass functions.

Summarizing Distributions

A histogram is a complete description of the distribution of a sample; that is, given a histogram, we could reconstruct the values in the sample (although not their order).

If the details of the distribution are important, it might be necessary to present a histogram. But often we want to summarize the distribution with a few descriptive statistics.

Some of the characteristics we might want to report are:

central tendency
: Do the values tend to cluster around a particular point?

modes
: Is there more than one cluster?

spread
: How much variability is there in the values?

tails
: How quickly do the probabilities drop off as we move away from the modes?

outliers
: Are there extreme values far from the modes?

Statistics designed to answer these questions are called **summary statistics**. By far the most common summary statistic is the **mean**, which is meant to describe the **central tendency** of the distribution.

If you have a sample of n values, x_i, the mean, $\bar{x}$, is the sum of the values divided by the number of values:

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

The words *mean* and *average* are sometimes used interchangeably, but I make this distinction:

- The *mean* of a sample is the summary statistic computed with the previous formula.
- An *average* is one of several summary statistics you might choose to describe a central tendency.

Sometimes the mean is a good description of a set of values. For example, apples are all pretty much the same size (at least the ones sold in supermarkets). So if I buy 6 apples and the total weight is 3 pounds, it would be a reasonable summary to say they are about a half pound each.

But pumpkins are more diverse. Suppose I grow several varieties in my garden, and one day I harvest three decorative pumpkins that are 1 pound each, two pie pumpkins that are 3 pounds each, and one Atlantic Giant pumpkin that weighs 591 pounds. The mean of this sample is 100 pounds, but if I told you "The average pumpkin in my garden is 100 pounds," that would be misleading. In this example, there is no meaningful average because there is no typical pumpkin.

Variance

If there is no single number that summarizes pumpkin weights, we can do a little better with two numbers: mean and **variance**.

Variance is a summary statistic intended to describe the variability or **spread** of a distribution. The variance of a set of values is

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

The term $x_i - \bar{x}$ is called the "deviation from the mean," so variance is the mean squared deviation. The square root of variance, S, is the **standard deviation**.

If you have prior experience, you might have seen a formula for variance with $n - 1$ in the denominator, rather than `n`. This statistic is used to estimate the variance in a population using a sample. We will come back to this in Chapter 8.

Pandas data structures provides methods to compute mean, variance and standard deviation:

```
mean = live.prglngth.mean()
var = live.prglngth.var()
std = live.prglngth.std()
```

For all live births, the mean pregnancy length is 38.6 weeks, the standard deviation is 2.7 weeks, which means we should expect deviations of 2-3 weeks to be common.

Variance of pregnancy length is 7.3, which is hard to interpret, especially since the units are weeks^2, or "square weeks." Variance is useful in some calculations, but it is not a good summary statistic.

Effect Size

An **effect size** is a summary statistic intended to describe (wait for it) the size of an effect. For example, to describe the difference between two groups, one obvious choice is the difference in the means.

Mean pregnancy length for first babies is 38.601; for other babies it is 38.523. The difference is 0.078 weeks, which works out to 13 hours. As a fraction of the typical pregnancy length, this difference is about 0.2%.

If we assume this estimate is accurate, such a difference would have no practical consequences. In fact, without observing a large number of pregnancies, it is unlikely that anyone would notice this difference at all.

Another way to convey the size of the effect is to compare the difference between groups to the variability within groups. Cohen's *d* is a statistic intended to do that; it is defined

$$d = \frac{\bar{x}_1 - \bar{x}_2}{s}$$

where $\bar{x}_1$ and $\bar{x}_2$ are the means of the groups and s is the "pooled standard deviation". Here's the Python code that computes Cohen's *d*:

```
def CohenEffectSize(group1, group2):
    diff = group1.mean() - group2.mean()

    var1 = group1.var()
    var2 = group2.var()
    n1, n2 = len(group1), len(group2)

    pooled_var = (n1 * var1 + n2 * var2) / (n1 + n2)
    d = diff / math.sqrt(pooled_var)
    return d
```

In this example, the difference in means is 0.029 standard deviations, which is small. To put that in perspective, the difference in height between men and women is about 1.7 standard deviations (see Wikipedia).

Reporting Results

We have seen several ways to describe the difference in pregnancy length (if there is one) between first babies and others. How should we report these results?

The answer depends on who is asking the question. A scientist might be interested in any (real) effect, no matter how small. A doctor might only care about effects that are **clinically significant**; that is, differences that affect treatment decisions. A pregnant woman might be interested in results that are relevant to her, like the probability of delivering early or late.

How you report results also depends on your goals. If you are trying to demonstrate the importance of an effect, you might choose summary statistics that emphasize differences. If you are trying to reassure a patient, you might choose statistics that put the differences in context.

Of course your decisions should also be guided by professional ethics. It's OK to be persuasive; you *should* design statistical reports and visualizations that tell a story clearly. But you should also do your best to make your reports honest, and to acknowledge uncertainty and limitations.

Exercises

Exercise 2-1.

Based on the results in this chapter, suppose you were asked to summarize what you learned about whether first babies arrive late.

Which summary statistics would you use if you wanted to get a story on the evening news? Which ones would you use if you wanted to reassure an anxious patient?

Finally, imagine that you are Cecil Adams, author of *The Straight Dope*, and your job is to answer the question, "Do first babies arrive late?" Write a paragraph that uses the results in this chapter to answer the question clearly, precisely, and honestly.

Exercise 2-2.

In the repository you downloaded, you should find a file named `chap02ex.ipynb`; open it. Some cells are already filled in, and you should execute them. Other cells give you instructions for exercises. Follow the instructions and fill in the answers.

A solution to this exercise is in `chap02soln.ipynb`

For the following exercises, create a file named `chap02ex.py`. You can find a solution in `chap02soln.py`.

Exercise 2-3.

The mode of a distribution is the most frequent value; see Wikipedia. Write a function called `Mode` that takes a Hist and returns the most frequent value.

As a more challenging exercise, write a function called `AllModes` that returns a list of value-frequency pairs in descending order of frequency.

Exercise 2-4.

Using the variable `totalwgt_lb`, investigate whether first babies are lighter or heavier than others. Compute Cohen's d to quantify the difference between the groups. How does it compare to the difference in pregnancy length?

Glossary

distribution
: The values that appear in a sample and the frequency of each.

histogram
: A mapping from values to frequencies, or a graph that shows this mapping.

frequency
: The number of times a value appears in a sample.

mode
: The most frequent value in a sample, or one of the most frequent values.

normal distribution
: An idealization of a bell-shaped distribution; also known as a Gaussian distribution.

uniform distribution
: A distribution in which all values have the same frequency.

tail
: The part of a distribution at the high and low extremes.

central tendency
: A characteristic of a sample or population; intuitively, it is an average or typical value.

outlier
: A value far from the central tendency.

spread
: A measure of how spread out the values in a distribution are.

summary statistic
: A statistic that quantifies some aspect of a distribution, like central tendency or spread.

variance
: A summary statistic often used to quantify spread.

standard deviation
: The square root of variance, also used as a measure of spread.

effect size
: A summary statistic intended to quantify the size of an effect like a difference between groups.

clinically significant
: A result, like a difference between groups, that is relevant in practice.

CHAPTER 3
Probability Mass Functions

The code for this chapter is in `probability.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

Pmfs

Another way to represent a distribution is a **probability mass function** (PMF), which maps from each value to its probability. A **probability** is a frequency expressed as a fraction of the sample size, `n`. To get from frequencies to probabilities, we divide through by `n`, which is called **normalization**.

Given a Hist, we can make a dictionary that maps from each value to its probability:

```
n = hist.Total()
d = {}
for x, freq in hist.Items():
    d[x] = freq / n
```

Or we can use the Pmf class provided by `thinkstats2`. Like Hist, the Pmf constructor can take a list, pandas Series, dictionary, Hist, or another Pmf object. Here's an example with a simple list:

```
>>> import thinkstats2
>>> pmf = thinkstats2.Pmf([1, 2, 2, 3, 5])
>>> pmf
Pmf({1: 0.2, 2: 0.4, 3: 0.2, 5: 0.2})
```

The Pmf is normalized so total probability is 1.

Pmf and Hist objects are similar in many ways; in fact, they inherit many of their methods from a common parent class. For example, the methods `Values` and `Items` work the same way for both. The biggest difference is that a Hist maps from values to integer counters; a Pmf maps from values to floating-point probabilities.

To look up the probability associated with a value, use `Prob`:

```
>>> pmf.Prob(2)
0.4
```

The bracket operator is equivalent:

```
>>> pmf[2]
0.4
```

You can modify an existing Pmf by incrementing the probability associated with a value:

```
>>> pmf.Incr(2, 0.2)
>>> pmf.Prob(2)
0.6
```

Or you can multiply a probability by a factor:

```
>>> pmf.Mult(2, 0.5)
>>> pmf.Prob(2)
0.3
```

If you modify a Pmf, the result may not be normalized; that is, the probabilities may no longer add up to 1. To check, you can call `Total`, which returns the sum of the probabilities:

```
>>> pmf.Total()
0.9
```

To renormalize, call `Normalize`:

```
>>> pmf.Normalize()
>>> pmf.Total()
1.0
```

Pmf objects provide a `Copy` method so you can make and modify a copy without affecting the original.

My notation in this section might seem inconsistent, but there is a system: I use Pmf for the name of the class, `pmf` for an instance of the class, and PMF for the mathematical concept of a probability mass function.

Plotting PMFs

`thinkplot` provides two ways to plot Pmfs:

- To plot a Pmf as a bar graph, you can use `thinkplot.Hist`. Bar graphs are most useful if the number of values in the Pmf is small.
- To plot a Pmf as a step function, you can use `thinkplot.Pmf`. This option is most useful if there are a large number of values and the Pmf is smooth. This function also works with Hist objects.

In addition, `pyplot` provides a function called `hist` that takes a sequence of values, computes a histogram, and plots it. Since I use Hist objects, I usually don't use `pyplot.hist`.

Figure 3-1 shows PMFs of pregnancy length for first babies and others using bar graphs (left) and step functions (right).

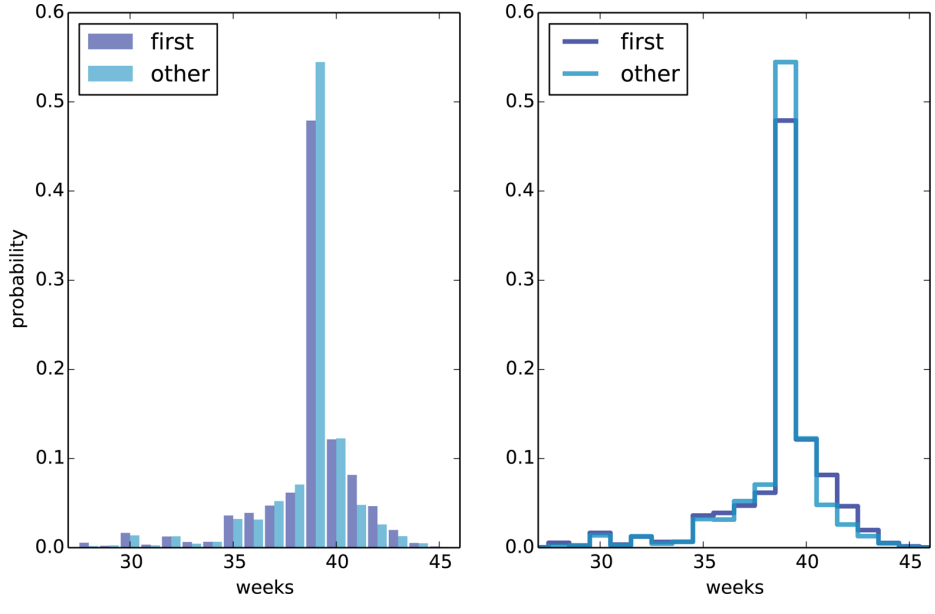


Figure 3-1. PMF of pregnancy lengths for first babies and others, using bar graphs and step functions

Here's the code that generates Figure 3-1:

```
thinkplot.PrePlot(2, cols=2)
thinkplot.Hist(first_pmf, align='right', width=width)
thinkplot.Hist(other_pmf, align='left', width=width)
thinkplot.Config(xlabel='weeks',
                 ylabel='probability',
                 axis=[27, 46, 0, 0.6])

thinkplot.PrePlot(2)
thinkplot.SubPlot(2)
thinkplot.Pmfs([first_pmf, other_pmf])
thinkplot.Show(xlabel='weeks',
               axis=[27, 46, 0, 0.6])
```

By plotting the PMF instead of the histogram, we can compare the two distributions without being misled by the difference in sample size. Based on this figure, first babies

seem to be less likely than others to arrive on time (week 39) and more likely to be late (weeks 41 and 42).

`PrePlot` takes optional parameters `rows` and `cols` to make a grid of figures, in this case one row of two figures. The figure on the left displays the Pmfs using `thinkplot.Hist`, as we have seen before.

The second call to `PrePlot` resets the color generator. Then `SubPlot` switches to the second figure (on the right) and displays the Pmfs using `thinkplot.Pmfs`. I used the `axis` option to ensure that the two figures are on the same axes, which is generally a good idea if you intend to compare two figures.

Other Visualizations

Histograms and PMFs are useful while you are exploring data and trying to identify patterns and relationships. Once you have an idea what is going on, a good next step is to design a visualization that makes the patterns you have identified as clear as possible.

In the NSFG data, the biggest differences in the distributions are near the mode. So it makes sense to zoom in on that part of the graph, and transform the data to emphasize differences:

```
weeks = range(35, 46)
diffs = []
for week in weeks:
    p1 = first_pmf.Prob(week)
    p2 = other_pmf.Prob(week)
    diff = 100 * (p1 - p2)
    diffs.append(diff)

thinkplot.Bar(weeks, diffs)
```

In this code, `weeks` is the range of weeks; `diffs` is the difference between the two PMFs in percentage points. Figure 3-2 shows the result as a bar chart. This figure makes the pattern clearer: first babies are less likely to be born in week 39, and somewhat more likely to be born in weeks 41 and 42.

For now we should hold this conclusion only tentatively. We used the same dataset to identify an apparent difference and then chose a visualization that makes the difference apparent. We can't be sure this effect is real; it might be due to random variation. We'll address this concern later.

The Class Size Paradox

Before we go on, I want to demonstrate one kind of computation you can do with Pmf objects; I call this example the "class size paradox."

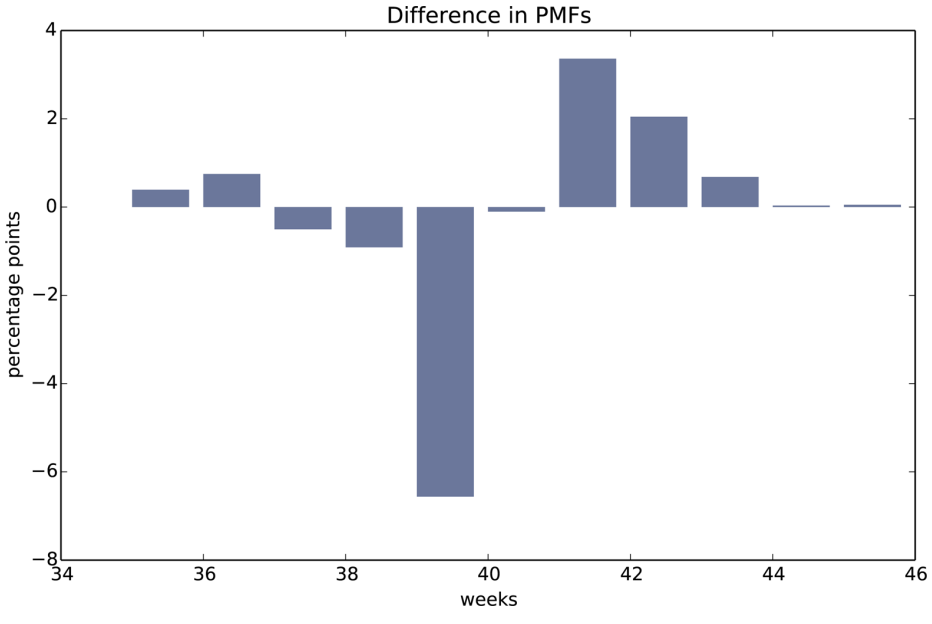


Figure 3-2. Difference, in percentage points, by week

At many American colleges and universities, the student-to-faculty ratio is about 10:1. But students are often surprised to discover that their average class size is bigger than 10. There are two reasons for the discrepancy:

- Students typically take 4–5 classes per semester, but professors often teach 1 or 2.
- The number of students who enjoy a small class is small, but the number of students in a large class is (ahem!) large.

The first effect is obvious, at least once it is pointed out; the second is more subtle. Let's look at an example. Suppose that a college offers 65 classes in a given semester, with the following distribution of sizes:

```
  size      count
   5- 9         8
 10-14          8
 15-19         14
 20-24          4
 25-29          6
 30-34         12
 35-39          8
 40-44          3
 45-49          2
```

If you ask the Dean for the average class size, he would construct a PMF, compute the mean, and report that the average class size is 23.7. Here's the code:

```
d = { 7: 8, 12: 8, 17: 14, 22: 4,
      27: 6, 32: 12, 37: 8, 42: 3, 47: 2 }

pmf = thinkstats2.Pmf(d, label='actual')
print('mean', pmf.Mean())
```

But if you survey a group of students, ask them how many students are in their classes, and compute the mean, you would think the average class size was bigger. Let's see how much bigger.

First, I compute the distribution as observed by students, where the probability associated with each class size is "biased" by the number of students in the class.

```
def BiasPmf(pmf, label):
    new_pmf = pmf.Copy(label=label)

    for x, p in pmf.Items():
        new_pmf.Mult(x, x)

    new_pmf.Normalize()
    return new_pmf
```

For each class size, `x`, we multiply the probability by `x`, the number of students who observe that class size. The result is a new Pmf that represents the biased distribution.

Now we can plot the actual and observed distributions:

```
biased_pmf = BiasPmf(pmf, label='observed')
thinkplot.PrePlot(2)
thinkplot.Pmfs([pmf, biased_pmf])
thinkplot.Show(xlabel='class size', ylabel='PMF')
```

Figure 3-3 shows the result. In the biased distribution there are fewer small classes and more large ones. The mean of the biased distribution is 29.1, almost 25% higher than the actual mean.

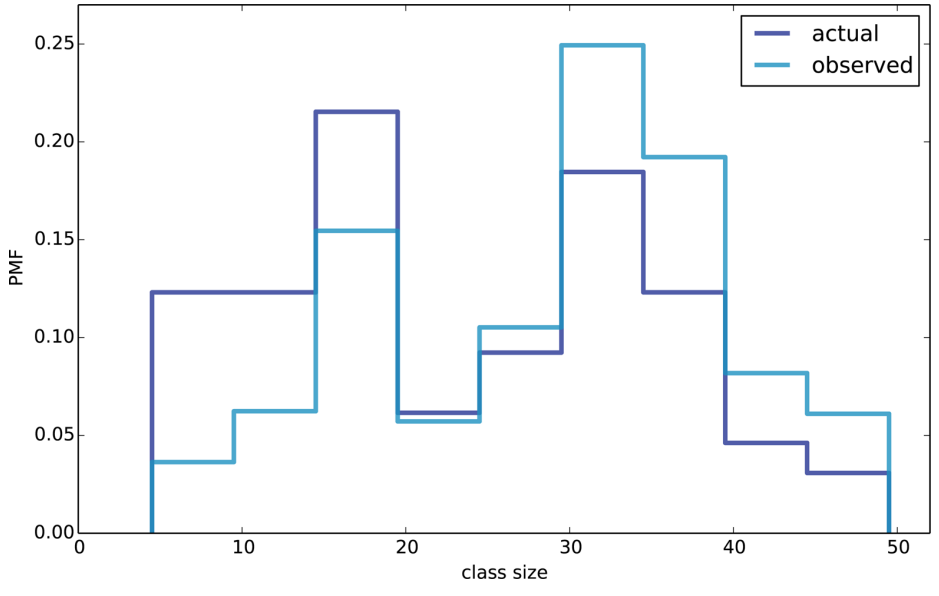


Figure 3-3. Distribution of class sizes, actual and as observed by students

It is also possible to invert this operation. Suppose you want to find the distribution of class sizes at a college, but you can't get reliable data from the Dean. An alternative is to choose a random sample of students and ask how many students are in their classes.

The result would be biased for the reasons we've just seen, but you can use it to estimate the actual distribution. Here's the function that unbiases a Pmf:

```
def UnbiasPmf(pmf, label):
    new_pmf = pmf.Copy(label=label)

    for x, p in pmf.Items():
        new_pmf.Mult(x, 1.0/x)

    new_pmf.Normalize()
    return new_pmf
```

It's similar to `BiasPmf`; the only difference is that it divides each probability by `x` instead of multiplying.

DataFrame Indexing

In "DataFrames" on page 4 we read a pandas DataFrame and used it to select and modify data columns. Now let's look at row selection. To start, I create a NumPy array of random numbers and use it to initialize a DataFrame:

```
>>> import numpy as np
>>> import pandas
>>> array = np.random.randn(4, 2)
>>> df = pandas.DataFrame(array)
>>> df
          0         1
0 -0.143510  0.616050
1 -1.489647  0.300774
2 -0.074350  0.039621
3 -1.369968  0.545897
```

By default, the rows and columns are numbered starting at zero, but you can provide column names:

```
>>> columns = ['A', 'B']
>>> df = pandas.DataFrame(array, columns=columns)
>>> df
          A         B
0 -0.143510  0.616050
1 -1.489647  0.300774
2 -0.074350  0.039621
3 -1.369968  0.545897
```

You can also provide row names. The set of row names is called the **index**; the row names themselves are called **labels**.

```
>>> index = ['a', 'b', 'c', 'd']
>>> df = pandas.DataFrame(array, columns=columns, index=index)
>>> df
          A         B
a -0.143510  0.616050
b -1.489647  0.300774
c -0.074350  0.039621
d -1.369968  0.545897
```

As we saw in the previous chapter, simple indexing selects a column, returning a Series:

```
>>> df['A']
a   -0.143510
b   -1.489647
c   -0.074350
d   -1.369968
Name: A, dtype: float64
```

To select a row by label, you can use the `loc` attribute, which returns a Series:

```
>>> df.loc['a']
A   -0.14351
B    0.61605
Name: a, dtype: float64
```

If you know the integer position of a row, rather than its label, you can use the `iloc` attribute, which also returns a Series.

```
>>> df.iloc[0]
A   -0.14351
B    0.61605
Name: a, dtype: float64
```

`loc` can also take a list of labels; in that case, the result is a DataFrame.

```
>>> indices = ['a', 'c']
>>> df.loc[indices]
         A         B
a -0.14351  0.616050
c -0.07435  0.039621
```

Finally, you can use a slice to select a range of rows by label:

```
>>> df['a':'c']
          A         B
a -0.143510  0.616050
b -1.489647  0.300774
c -0.074350  0.039621
```

Or by integer position:

```
>>> df[0:2]
          A         B
a -0.143510  0.616050
b -1.489647  0.300774
```

The result in either case is a DataFrame, but notice that the first result includes the end of the slice; the second doesn't.

My advice: if your rows have labels that are not simple integers, use the labels consistently and avoid using integer positions.

Exercises

Solutions to these exercises are in `chap03soln.ipynb` and `chap03soln.py`

Exercise 3-1.

Something like the class size paradox appears if you survey children and ask how many children are in their family. Families with many children are more likely to appear in your sample, and families with no children have no chance to be in the sample.

Use the NSFG respondent variable `NUMKDHH` to construct the actual distribution for the number of children under 18 in the household.

Now compute the biased distribution we would see if we surveyed the children and asked them how many children under 18 (including themselves) are in their household.

Plot the actual and biased distributions, and compute their means. As a starting place, you can use `chap03ex.ipynb`.

Exercise 3-2.

In "Summarizing Distributions" on page 22 we computed the mean of a sample by adding up the elements and dividing by n. If you are given a PMF, you can still compute the mean, but the process is slightly different:

$$\bar{x} = \sum_i p_i \ x_i$$

where the x_i are the unique values in the PMF and $p_i = PMF(x_i)$. Similarly, you can compute variance like this:

$$S^2 = \sum_i p_i \ (x_i - \bar{x})^2$$

Write functions called `PmfMean` and `PmfVar` that take a Pmf object and compute the mean and variance. To test these methods, check that they are consistent with the methods `Mean` and `Var` provided by Pmf.

Exercise 3-3.

I started with the question, "Are first babies more likely to be late?" To address it, I computed the difference in means between groups of babies, but I ignored the possibility that there might be a difference between first babies and others *for the same woman*.

To address this version of the question, select respondents who have at least two babies and compute pairwise differences. Does this formulation of the question yield a different result?

Hint: use `nsfg.MakePregMap`.

Exercise 3-4.

In most foot races, everyone starts at the same time. If you are a fast runner, you usually pass a lot of people at the beginning of the race, but after a few miles everyone around you is going at the same speed.

When I ran a long-distance (209 miles) relay race for the first time, I noticed an odd phenomenon: when I overtook another runner, I was usually much faster, and when another runner overtook me, he was usually much faster.

At first I thought that the distribution of speeds might be bimodal; that is, there were many slow runners and many fast runners, but few at my speed.

Then I realized that I was the victim of a bias similar to the effect of class size. The race was unusual in two ways: it used a staggered start, so teams started at different times; also, many teams included runners at different levels of ability.

As a result, runners were spread out along the course with little relationship between speed and location. When I joined the race, the runners near me were (pretty much) a random sample of the runners in the race.

So where does the bias come from? During my time on the course, the chance of overtaking a runner, or being overtaken, is proportional to the difference in our speeds. I am more likely to catch a slow runner, and more likely to be caught by a fast runner. But runners at the same speed are unlikely to see each other.

Write a function called `ObservedPmf` that takes a Pmf representing the actual distribution of runners' speeds, and the speed of a running observer, and returns a new Pmf representing the distribution of runners' speeds as seen by the observer.

To test your function, you can use `relay.py`, which reads the results from the James Joyce Ramble 10K in Dedham MA and converts the pace of each runner to mph.

Compute the distribution of speeds you would observe if you ran a relay race at 7.5 mph with this group of runners. A solution to this exercise is in `relay_soln.py`.

Glossary

Probability mass function (PMF)
: a representation of a distribution as a function that maps from values to probabilities.

probability
: A frequency expressed as a fraction of the sample size.

normalization
: The process of dividing a frequency by a sample size to get a probability.

index
: In a pandas DataFrame, the index is a special column that contains the row labels.

CHAPTER 4

Cumulative Distribution Functions

The code for this chapter is in `cumulative.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

The Limits of PMFs

PMFs work well if the number of values is small. But as the number of values increases, the probability associated with each value gets smaller and the effect of random noise increases.

For example, we might be interested in the distribution of birth weights. In the NSFG data, the variable `totalwgt_lb` records weight at birth in pounds. Figure 4-1 shows the PMF of these values for first babies and others.

Overall, these distributions resemble the bell shape of a normal distribution, with many values near the mean and a few values much higher and lower.

But parts of this figure are hard to interpret. There are many spikes and valleys, and some apparent differences between the distributions. It is hard to tell which of these features are meaningful. Also, it is hard to see overall patterns; for example, which distribution do you think has the higher mean?

These problems can be mitigated by binning the data; that is, dividing the range of values into non-overlapping intervals and counting the number of values in each bin. Binning can be useful, but it is tricky to get the size of the bins right. If they are big enough to smooth out noise, they might also smooth out useful information.

An alternative that avoids these problems is the cumulative distribution function (CDF), which is the subject of this chapter. But before I can explain CDFs, I have to explain percentiles.

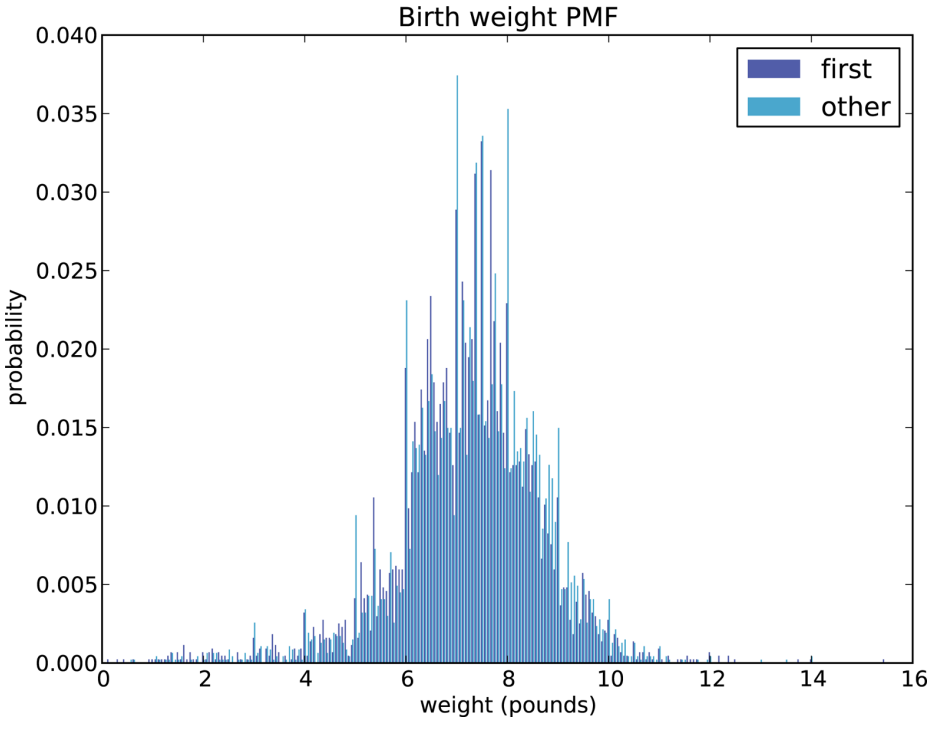


Figure 4-1. PMF of birth weights showing a limitation of PMFs: they are hard to compare visually

Percentiles

If you have taken a standardized test, you probably got your results in the form of a raw score and a **percentile rank**. In this context, the percentile rank is the fraction of people who scored lower than you (or the same). So if you are "in the 90th percentile," you did as well as or better than 90% of the people who took the exam.

Here's how you could compute the percentile rank of a value, `your_score`, relative to the values in the sequence `scores`:

```
def PercentileRank(scores, your_score):
    count = 0
    for score in scores:
        if score <= your_score:
            count += 1

    percentile_rank = 100.0 * count / len(scores)
    return percentile_rank
```

As an example, if the scores in the sequence were 55, 66, 77, 88 and 99, and you got the 88, then your percentile rank would be `100 * 4 / 5` which is 80.

If you are given a value, it is easy to find its percentile rank; going the other way is slightly harder. If you are given a percentile rank and you want to find the corresponding value, one option is to sort the values and search for the one you want:

```
def Percentile(scores, percentile_rank):
    scores.sort()
    for score in scores:
        if PercentileRank(scores, score) >= percentile_rank:
            return score
```

The result of this calculation is a **percentile**. For example, the 50th percentile is the value with percentile rank 50. In the distribution of exam scores, the 50th percentile is 77.

This implementation of `Percentile` is not efficient. A better approach is to use the percentile rank to compute the index of the corresponding percentile:

```
def Percentile2(scores, percentile_rank):
    scores.sort()
    index = percentile_rank * (len(scores)-1) / 100
    return scores[index]
```

The difference between "percentile" and "percentile rank" can be confusing, and people do not always use the terms precisely. To summarize, `PercentileRank` takes a value and computes its percentile rank in a set of values; `Percentile` takes a percentile rank and computes the corresponding value.

CDFs

Now that we understand percentiles and percentile ranks, we are ready to tackle the **cumulative distribution function** (CDF). The CDF is the function that maps from a value to its percentile rank.

The CDF is a function of x, where x is any value that might appear in the distribution. To evaluate CDF(x) for a particular value of x, we compute the fraction of values in the distribution less than or equal to x.

Here's what that looks like as a function that takes a sequence, `t`, and a value, `x`:

```
def EvalCdf(t, x):
    count = 0.0
    for value in t:
        if value <= x:
            count += 1

    prob = count / len(t)
    return prob
```

This function is almost identical to `PercentileRank`, except that the result is a probability in the range 0–1 rather than a percentile rank in the range 0–100.

As an example, suppose we collect a sample with the values `[1, 2, 2, 3, 5]`. Here are some values from its CDF:

$CDF(0) = 0$

$CDF(1) = 0.2$

$CDF(2) = 0.6$

$CDF(3) = 0.8$

$CDF(4) = 0.8$

$CDF(5) = 1$

We can evaluate the CDF for any value of x, not just values that appear in the sample. If x is less than the smallest value in the sample, CDF(x) is 0. If x is greater than the largest value, CDF(x) is 1.

Figure 4-2 is a graphical representation of this CDF. The CDF of a sample is a step function.

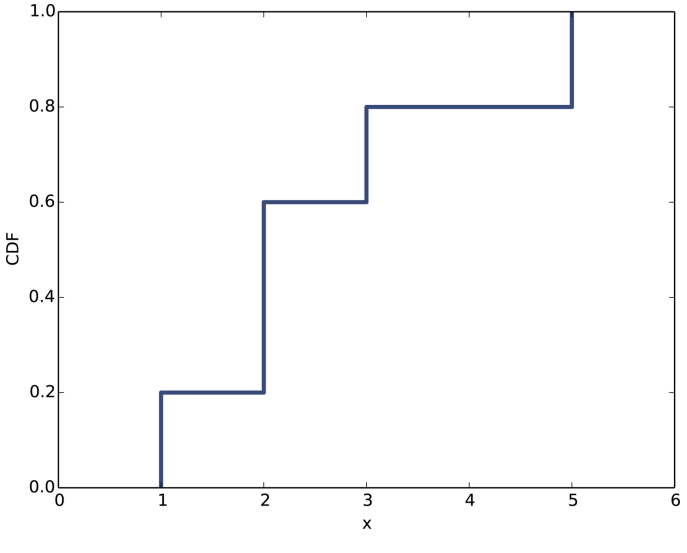


Figure 4-2. Example of a CDF

Representing CDFs

`thinkstats2` provides a class named Cdf that represents CDFs. The fundamental methods Cdf provides are:

`Prob(x)`
: Given a value `x`, computes the probability $p = \text{CDF}(x)$. The bracket operator is equivalent to `Prob`.

`Value(p)`
: Given a probability `p`, computes the corresponding value, `x`; that is, the **inverse CDF** of `p`.

The Cdf constructor can take as an argument a list of values, a pandas Series, a Hist, Pmf, or another Cdf. The following code makes a Cdf for the distribution of pregnancy lengths in the NSFG:

```
live, firsts, others = first.MakeFrames()
cdf = thinkstats2.Cdf(live.prglngth, label='prglngth')
```

`thinkplot` provides a function named `Cdf` that plots Cdfs as lines:

```
thinkplot.Cdf(cdf)
thinkplot.Show(xlabel='weeks', ylabel='CDF')
```

Figure 4-3 shows the result. One way to read a CDF is to look up percentiles. For example, it looks like about 10% of pregnancies are shorter than 36 weeks, and about 90% are shorter than 41 weeks. The CDF also provides a visual representation of the shape of the distribution. Common values appear as steep or vertical sections of the CDF; in this example, the mode at 39 weeks is apparent. There are few values below 30 weeks, so the CDF in this range is flat.

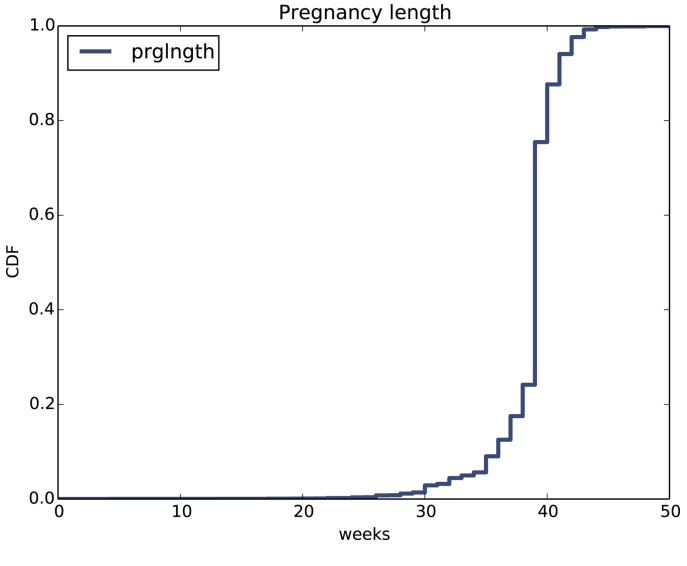


Figure 4-3. CDF of pregnancy length

It takes some time to get used to CDFs, but once you do, I think you will find that they show more information, more clearly, than PMFs.

Comparing CDFs

CDFs are especially useful for comparing distributions. For example, here is the code that plots the CDF of birth weights for first babies and others.

```
first_cdf = thinkstats2.Cdf(firsts.totalwgt_lb, label='first')
other_cdf = thinkstats2.Cdf(others.totalwgt_lb, label='other')

thinkplot.PrePlot(2)
thinkplot.Cdfs([first_cdf, other_cdf])
thinkplot.Show(xlabel='weight (pounds)', ylabel='CDF')
```

Figure 4-4 shows the result. Compared to Figure 4-1, this figure makes the shape of the distributions, and the differences between them, much clearer. We can see that first babies are slightly lighter throughout the distribution, with a larger discrepancy above the mean.

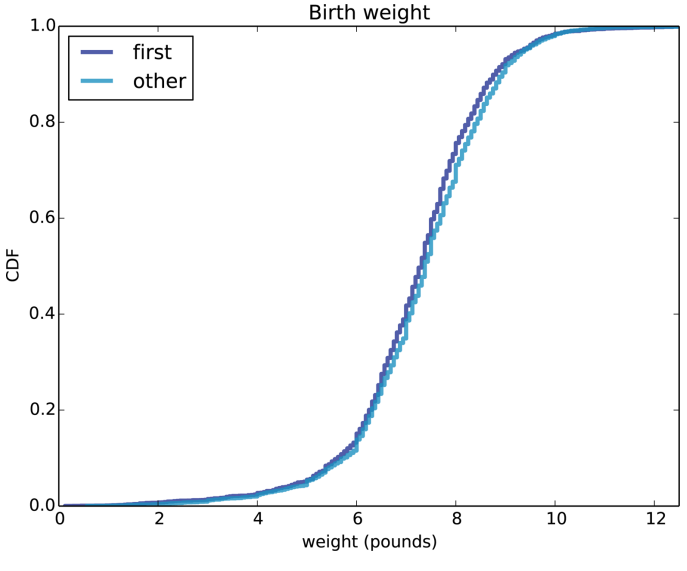


Figure 4-4. CDF of birth weights for first babies and others

Percentile-Based Statistics

Once you have computed a CDF, it is easy to compute percentiles and percentile ranks. The Cdf class provides these two methods:

`PercentileRank(x)`
: Given a value `x`, computes its percentile rank, $100 \cdot \text{CDF}(x)$.

`Percentile(p)`
: Given a percentile rank `rank`, computes the corresponding value, `x`. Equivalent to `Value(p/100)`.

`Percentile` can be used to compute percentile-based summary statistics. For example, the 50th percentile is the value that divides the distribution in half, also known as the **median**. Like the mean, the median is a measure of the central tendency of a distribution.

Actually, there are several definitions of "median," each with different properties. But `Percentile(50)` is simple and efficient to compute.

Another percentile-based statistic is the **interquartile range** (IQR), which is a measure of the spread of a distribution. The IQR is the difference between the 75th and 25th percentiles.

More generally, percentiles are often used to summarize the shape of a distribution. For example, the distribution of income is often reported in "quintiles"; that is, it is split at the 20th, 40th, 60th and 80th percentiles. Other distributions are divided into 10 "deciles". Statistics like these that represent equally spaced points in a CDF are called **quantiles**.

Random Numbers

Suppose we choose a random sample from the population of live births and look up the percentile rank of their birth weights. Now suppose we compute the CDF of the percentile ranks. What do you think the distribution will look like?

Here's how we can compute it. First, we make the Cdf of birth weights:

```
weights = live.totalwgt_lb
cdf = thinkstats2.Cdf(weights, label='totalwgt_lb')
```

Then we generate a sample and compute the percentile rank of each value in the sample.

```
sample = np.random.choice(weights, 100, replace=True)
ranks = [cdf.PercentileRank(x) for x in sample]
```

`sample` is a random sample of 100 birth weights, chosen with **replacement**; that is, the same value could be chosen more than once. `ranks` is a list of percentile ranks.

Finally we make and plot the Cdf of the percentile ranks.

```
rank_cdf = thinkstats2.Cdf(ranks)
thinkplot.Cdf(rank_cdf)
thinkplot.Show(xlabel='percentile rank', ylabel='CDF')
```

Figure 4-5 shows the result. The CDF is approximately a straight line, which means that the distribution is uniform.

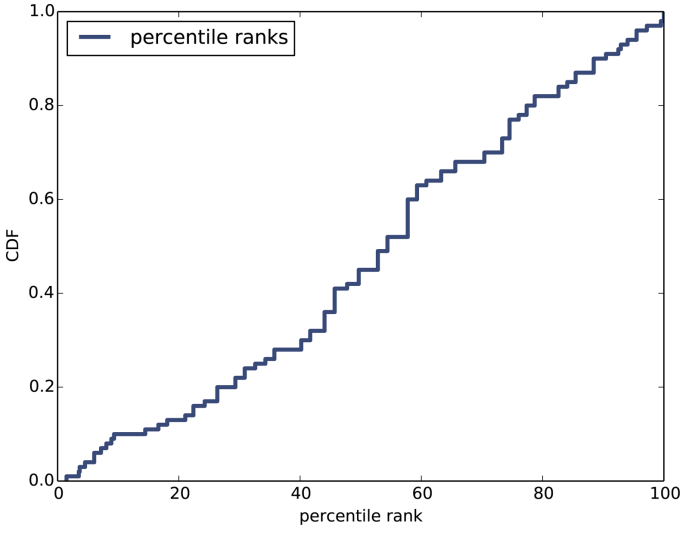


Figure 4-5. CDF of percentile ranks for a random sample of birth weights

That outcome might not be obvious, but it is a consequence of the way the CDF is defined. What this figure shows is that 10% of the sample is below the 10th percentile, 20% is below the 20th percentile, and so on, exactly as we should expect.

So, regardless of the shape of the CDF, the distribution of percentile ranks is uniform. This property is useful, because it is the basis of a simple and efficient algorithm for generating random numbers with a given CDF. Here's how:

- Choose a percentile rank uniformly from the range 0–100.
- Use `Cdf.Percentile` to find the value in the distribution that corresponds to the percentile rank you chose.

Cdf provides an implementation of this algorithm, called `Random`:

```
# class Cdf:
    def Random(self):
        return self.Percentile(random.uniform(0, 100))
```

Cdf also provides `Sample`, which takes an integer, `n`, and returns a list of `n` values chosen at random from the Cdf.

Comparing Percentile Ranks

Percentile ranks are useful for comparing measurements across different groups. For example, people who compete in foot races are usually grouped by age and gender. To compare people in different age groups, you can convert race times to percentile ranks.

A few years ago I ran the James Joyce Ramble 10K in Dedham MA; I finished in 42:44, which was 97th in a field of 1633. I beat or tied 1537 runners out of 1633, so my percentile rank in the field was 94%.

More generally, given position and field size, we can compute percentile rank:

```
def PositionToPercentile(position, field_size):
    beat = field_size - position + 1
    percentile = 100.0 * beat / field_size
    return percentile
```

In my age group, denoted M4049 for "males between 40 and 49 years of age", I came in 26th out of 256. So my percentile rank in my age group was 90%.

If I am still running in 10 years (and I hope I am), I will be in the M5059 division. Assuming that my percentile rank in my division is the same, how much slower should I expect to be?

I can answer that question by converting my percentile rank in M4049 to a position in M5059. Here's the code:

```
def PercentileToPosition(percentile, field_size):
    beat = percentile * field_size / 100.0
    position = field_size - beat + 1
    return position
```

There were 171 people in M5059, so I would have to come in between 17th and 18th place to have the same percentile rank. The finishing time of the 17th runner in M5059 was 46:05, so that's the time I will have to beat to maintain my percentile rank.

Exercises

For the following exercises, you can start with `chap04ex.ipynb`. My solution is in `chap04soln.ipynb`.

Exercise 4-1.

How much did you weigh at birth? If you don't know, call your mother or someone else who knows. Using the NSFG data (all live births), compute the distribution of birth weights and use it to find your percentile rank. If you were a first baby, find your percentile rank in the distribution for first babies. Otherwise use the distribution for others. If you are in the 90th percentile or higher, call your mother back and apologize.

Exercise 4-2.

The numbers generated by `random.random` are supposed to be uniform between 0 and 1; that is, every value in the range should have the same probability.

Generate 1000 numbers from `random.random` and plot their PMF and CDF. Is the distribution uniform?

Glossary

percentile rank
: The percentage of values in a distribution that are less than or equal to a given value.

percentile
: The value associated with a given percentile rank.

cumulative distribution function (CDF)
: A function that maps from values to their cumulative probabilities. CDF(x) is the fraction of the sample less than or equal to x.

inverse CDF
: A function that maps from a cumulative probability, p, to the corresponding value.

median
: The 50th percentile, often used as a measure of central tendency.

interquartile range
: The difference between the 75th and 25th percentiles, used as a measure of spread.

quantile
: A sequence of values that correspond to equally spaced percentile ranks; for example, the quartiles of a distribution are the 25th, 50th and 75th percentiles.

replacement
: A property of a sampling process. "With replacement" means that the same value can be chosen more than once; "without replacement" means that once a value is chosen, it is removed from the population.

CHAPTER 5

Modeling Distributions

The distributions we have used so far are called **empirical distributions** because they are based on empirical observations, which are necessarily finite samples.

The alternative is an **analytic distribution**, which is characterized by a CDF that is a mathematical function. Analytic distributions can be used to model empirical distributions. In this context, a **model** is a simplification that leaves out unneeded details. This chapter presents common analytic distributions and uses them to model data from a variety of sources.

The code for this chapter is in `analytic.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

The Exponential Distribution

I'll start with the **exponential distribution** because it is relatively simple. The CDF of the exponential distribution is

$$\text{CDF}(x) = 1 - e^{-\lambda x}$$

The parameter, λ, determines the shape of the distribution. Figure 5-1 shows what this CDF looks like with $\lambda = 0.5$, 1, and 2.

In the real world, exponential distributions come up when we look at a series of events and measure the times between events, called **interarrival times**. If the events are equally likely to occur at any time, the distribution of interarrival times tends to look like an exponential distribution.

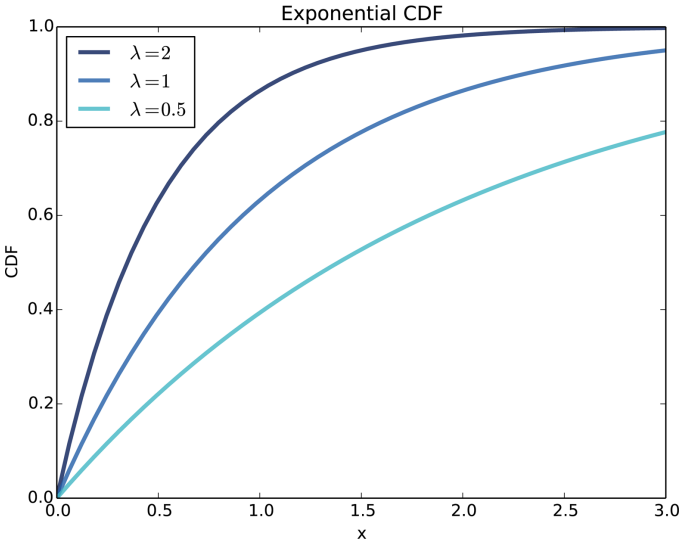


Figure 5-1. CDFs of exponential distributions with various parameters.

As an example, we will look at the interarrival time of births. On December 18, 1997, 44 babies were born in a hospital in Brisbane, Australia.[1] The time of birth for all 44 babies was reported in the local paper; the complete dataset is in a file called `babyboom.dat`, in the `ThinkStats2` repository.

```
df = ReadBabyBoom()
diffs = df.minutes.diff()
cdf = thinkstats2.Cdf(diffs, label='actual')

thinkplot.Cdf(cdf)
thinkplot.Show(xlabel='minutes', ylabel='CDF')
```

`ReadBabyBoom` reads the data file and returns a DataFrame with columns `time`, `sex`, `weight_g`, and `minutes`, where `minutes` is the time of birth converted to minutes since midnight.

`diffs` is the difference between consecutive birth times, and `cdf` is the distribution of these interarrival times. Figure 5-2 (left) shows the CDF. It seems to have the general shape of an exponential distribution, but how can we tell?

1. This example is based on information and data from Dunn, "A Simple Dataset for Demonstrating Common Distributions," Journal of Statistics Education v.7, n.3 (1999).

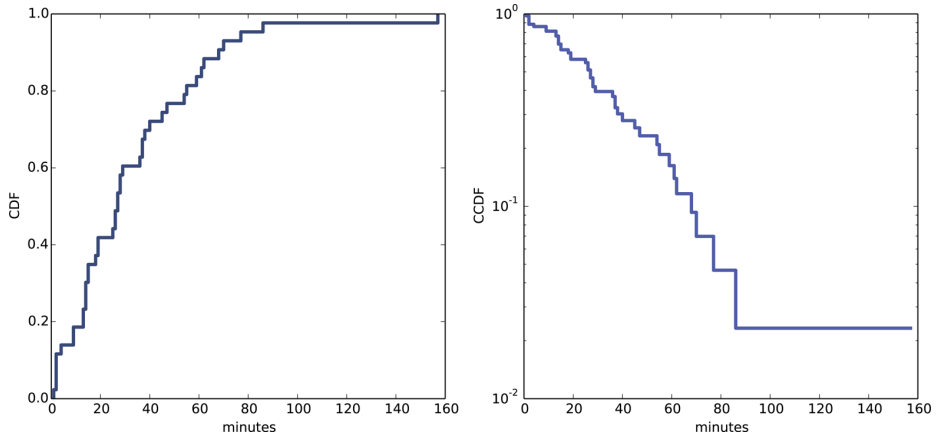


Figure 5-2. CDF of interarrival times (left) and CCDF on a log-y scale (right)

One way is to plot the **complementary CDF**, which is 1 - CDF(x), on a log-y scale. For data from an exponential distribution, the result is a straight line. Let's see why that works.

If you plot the complementary CDF (CCDF) of a dataset that you think is exponential, you expect to see a function like:

$$y \approx e^{-\lambda x}$$

Taking the log of both sides yields:

$$\log y \approx -\lambda x$$

So on a log-y scale the CCDF is a straight line with slope $-\lambda$. Here's how we can generate a plot like that:

```
thinkplot.Cdf(cdf, complement=True)
thinkplot.Show(xlabel='minutes',
               ylabel='CCDF',
               yscale='log')
```

With the argument `complement=True`, `thinkplot.Cdf` computes the complementary CDF before plotting. And with `yscale='log'`, `thinkplot.Show` sets the `y` axis to a logarithmic scale.

Figure 5-2 (right) shows the result. It is not exactly straight, which indicates that the exponential distribution is not a perfect model for this data. Most likely the underlying assumption—that a birth is equally likely at any time of day—is not exactly true. Nevertheless, it might be reasonable to model this dataset with an exponential distribution. With that simplification, we can summarize the distribution with a single parameter.

The parameter, λ, can be interpreted as a rate; that is, the number of events that occur, on average, in a unit of time. In this example, 44 babies are born in 24 hours, so the rate is $\lambda = 0.0306$ births per minute. The mean of an exponential distribution is $1 / \lambda$, so the mean time between births is 32.7 minutes.

The Normal Distribution

The **normal distribution**, also called Gaussian, is commonly used because it describes many phenomena, at least approximately. It turns out that there is a good reason for its ubiquity, which we will get to in "Central Limit Theorem" on page 186.

The normal distribution is characterized by two parameters: the mean, μ, and standard deviation σ. The normal distribution with $\mu = 0$ and $\sigma = 1$ is called the **standard normal distribution**. Its CDF is defined by an integral that does not have a closed form solution, but there are algorithms that evaluate it efficiently. One of them is provided by SciPy: `scipy.stats.norm` is an object that represents a normal distribution; it provides a method, `cdf`, that evaluates the standard normal CDF:

```
>>> import scipy.stats
>>> scipy.stats.norm.cdf(0)
0.5
```

This result is correct: the median of the standard normal distribution is 0 (the same as the mean), and half of the values fall below the median, so CDF(0) is 0.5.

`norm.cdf` takes optional parameters: `loc`, which specifies the mean, and `scale`, which specifies the standard deviation.

`thinkstats2` makes this function a little easier to use by providing `EvalNormalCdf`, which takes parameters `mu` and `sigma` and evaluates the CDF at `x`:

```
def EvalNormalCdf(x, mu=0, sigma=1):
    return scipy.stats.norm.cdf(x, loc=mu, scale=sigma)
```

Figure 5-3 shows CDFs for normal distributions with a range of parameters. The sigmoid shape of these curves is a recognizable characteristic of a normal distribution.

In the previous chapter, we looked at the distribution of birth weights in the NSFG. Figure 5-4 shows the empirical CDF of weights for all live births and the CDF of a normal distribution with the same mean and variance.

The normal distribution is a good model for this dataset, so if we summarize the distribution with the parameters $\mu = 7.28$ and $\sigma = 1.24$, the resulting error (difference between the model and the data) is small.

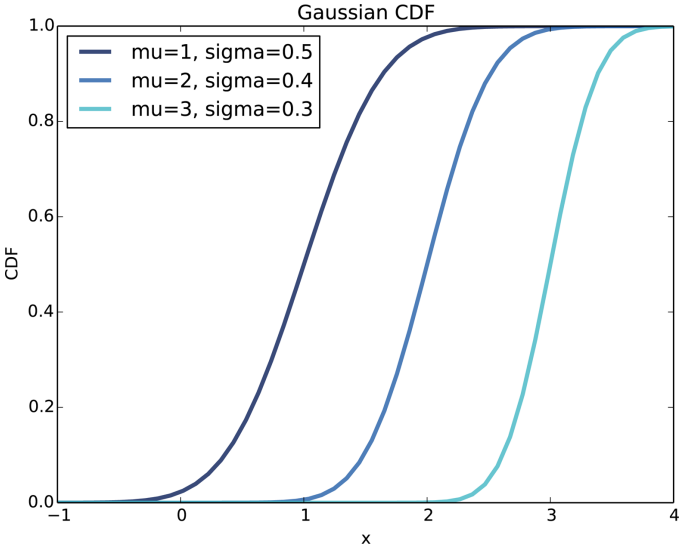


Figure 5-3. CDF of normal distributions with a range of parameters.

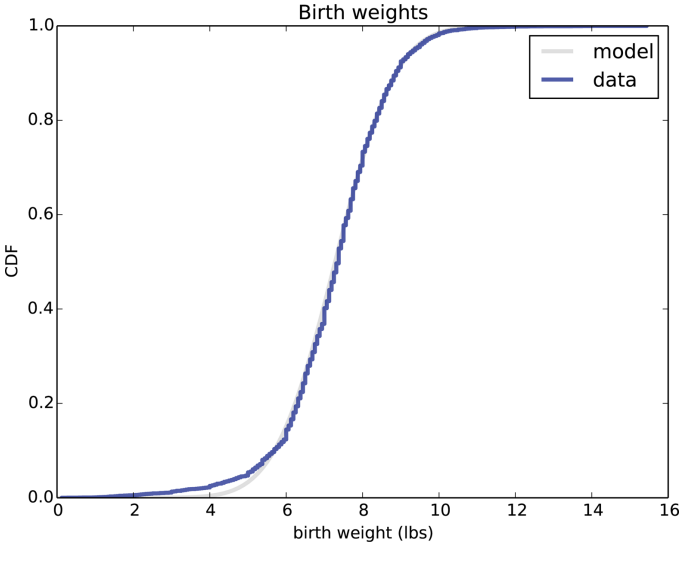


Figure 5-4. CDF of birth weights with a normal model

Below the 10th percentile there is a discrepancy between the data and the model; there are more light babies than we would expect in a normal distribution. If we are specifically

interested in preterm babies, it would be important to get this part of the distribution right, so it might not be appropriate to use the normal model.

Normal Probability Plot

For the exponential distribution, and a few others, there are simple transformations we can use to test whether an analytic distribution is a good model for a dataset.

For the normal distribution there is no such transformation, but there is an alternative called a **normal probability plot**. There are two ways to generate a normal probability plot: the hard way and the easy way. Here's the easy way:

1. Sort the values in the sample.
2. From a standard normal distribution ($\mu = 0$ and $\sigma = 1$), generate a random sample with the same size as the sample, and sort it.
3. Plot the sorted values from the sample versus the random values.

If the distribution of the sample is approximately normal, the result is a straight line with intercept `mu` and slope `sigma`. `thinkstats2` provides `NormalProbability`, which takes a sample and returns two NumPy arrays:

```
xs, ys = thinkstats2.NormalProbability(sample)
```

`ys` contains the sorted values from `sample`; `xs` contains the random values from the standard normal distribution.

To test `NormalProbability` I generated some fake samples that were actually drawn from normal distributions with various parameters. Figure 5-5 shows the results. The lines are approximately straight, with values in the tails deviating more than values near the mean.

Now let's try it with real data. Here's code to generate a normal probability plot for the birth weight data from the previous section. It plots a gray line that represents the model and a blue line that represents the data.

```
def MakeNormalPlot(weights):
    mean = weights.mean()
    std = weights.std()

    xs = [-4, 4]
    fxs, fys = thinkstats2.FitLine(xs, inter=mean, slope=std)
    thinkplot.Plot(fxs, fys, color='gray', label='model')

    xs, ys = thinkstats2.NormalProbability(weights)
    thinkplot.Plot(xs, ys, label='birth weights')
```

`weights` is a pandas Series of birth weights; `mean` and `std` are the mean and standard deviation.

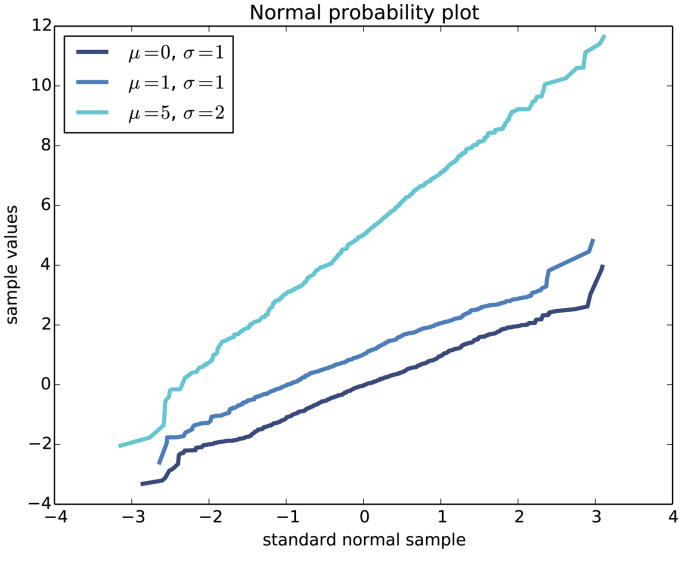


Figure 5-5. Normal probability plot for random samples from normal distributions

`FitLine` takes a sequence of `xs`, an intercept, and a slope; it returns `xs` and `ys` that represent a line with the given parameters, evaluated at the values in `xs`.

`NormalProbability` returns `xs` and `ys` that contain values from the standard normal distribution and values from `weights`. If the distribution of weights is normal, the data should match the model.

Figure 5-6 shows the results for all live births, and also for full term births (pregnancy length greater than 36 weeks). Both curves match the model near the mean and deviate in the tails. The heaviest babies are heavier than what the model expects, and the lightest babies are lighter.

When we select only full term births, we remove some of the lightest weights, which reduces the discrepancy in the lower tail of the distribution.

This plot suggests that the normal model describes the distribution well within a few standard deviations from the mean, but not in the tails. Whether it is good enough for practical purposes depends on the purposes.

The lognormal Distribution

If the logarithms of a set of values have a normal distribution, the values have a **lognormal distribution**. The CDF of the lognormal distribution is the same as the CDF of the normal distribution, with $\log x$ substituted for x.

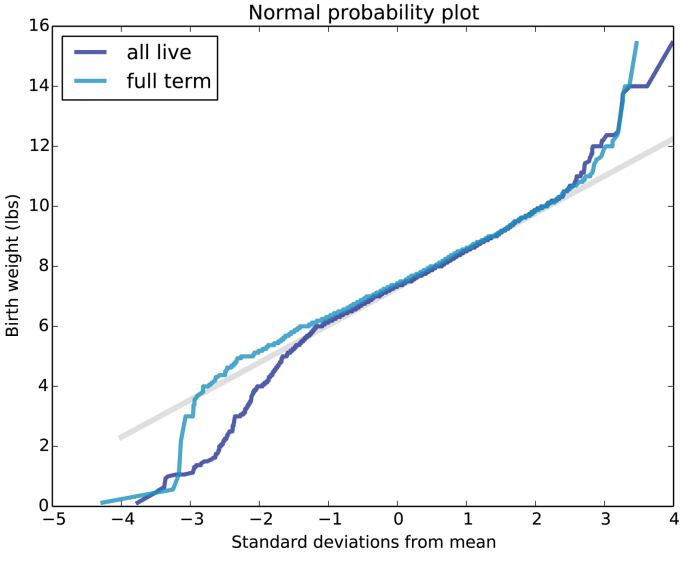


Figure 5-6. Normal probability plot of birth weights

$$CDF_{lognormal}(x) = CDF_{normal}(\log x)$$

The parameters of the lognormal distribution are usually denoted μ and σ. But remember that these parameters are *not* the mean and standard deviation; the mean of a lognormal distribution is $\exp(\mu + \sigma^2/2)$ and the standard deviation is ugly (see Wikipedia).

If a sample is approximately lognormal and you plot its CDF on a log-x scale, it will have the characteristic shape of a normal distribution. To test how well the sample fits a lognormal model, you can make a normal probability plot using the log of the values in the sample.

As an example, let's look at the distribution of adult weights, which is approximately lognormal.[2]

The National Center for Chronic Disease Prevention and Health Promotion conducts an annual survey as part of the Behavioral Risk Factor Surveillance System (BRFSS).[3]

2. I was tipped off to this possibility by a comment (without citation). Subsequently I found a paper that proposes the log transform and suggests a cause: Penman and Johnson, "The Changing Shape of the Body Mass Index Distribution Curve in the Population," Preventing Chronic Disease, 2006 July; 3(3): A74.

3. Centers for Disease Control and Prevention (CDC). Behavioral Risk Factor Surveillance System Survey Data. Atlanta, Georgia: US Department of Health and Human Services, Centers for Disease Control and Prevention, 2008.

In 2008, they interviewed 414,509 respondents and asked about their demographics, health, and health risks. Among the data they collected are the weights in kilograms of 398,484 respondents.

The repository for this book contains `CDBRFS08.ASC.gz`, a fixed-width ASCII file that contains data from the BRFSS, and `brfss.py`, which reads the file and analyzes the data.

Figure 5-7 (left) shows the distribution of adult weights on a linear scale with a normal model. Figure 5-7 (right) shows the same distribution on a log scale with a lognormal model. The lognormal model is a better fit, but this representation of the data does not make the difference particularly dramatic.

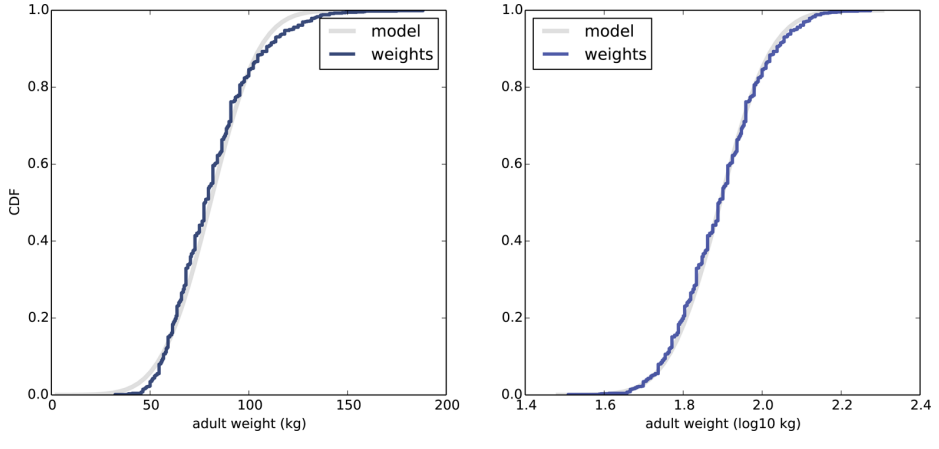


Figure 5-7. CDF of adult weights on a linear scale (left) and log scale (right)

Figure 5-8 shows normal probability plots for adult weights, w, and for their logarithms, $\log_{10} w$. Now it is apparent that the data deviate substantially from the normal model. The lognormal model is a good match for the data within a few standard deviations of the mean, but it deviates in the tails. I conclude that the lognormal distribution is a good model for this data.

The Pareto Distribution

The **Pareto distribution** is named after the economist Vilfredo Pareto, who used it to describe the distribution of wealth. Since then, it has been used to describe phenomena in the natural and social sciences including sizes of cities and towns, sand particles and meteorites, and forest fires and earthquakes.

The CDF of the Pareto distribution is:

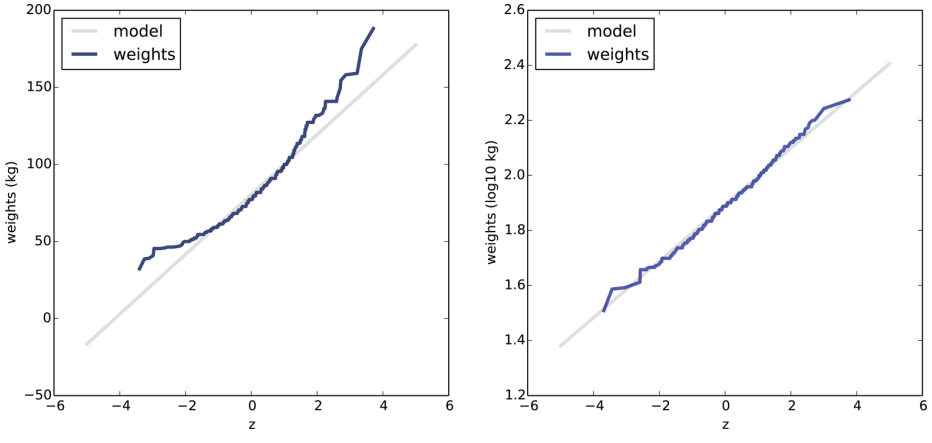


Figure 5-8. Normal probability plots for adult weight on a linear scale (left) and log scale (right).

$$CDF(x) = 1 - \left(\frac{x}{x_m}\right)^{-\alpha}$$

The parameters x_m and α determine the location and shape of the distribution. x_m is the minimum possible value. Figure 5-9 shows CDFs of Pareto distributions with $x_m = 0.5$ and different values of α.

There is a simple visual test that indicates whether an empirical distribution fits a Pareto distribution: on a log-log scale, the CCDF looks like a straight line. Let's see why that works.

If you plot the CCDF of a sample from a Pareto distribution on a linear scale, you expect to see a function like:

$$y \approx \left(\frac{x}{x_m}\right)^{-\alpha}$$

Taking the log of both sides yields:

$$\log y \approx -\alpha(\log x - \log x_m)$$

So if you plot $\log y$ versus $\log x$, it should look like a straight line with slope $-\alpha$ and intercept $\alpha \log x_m$.

As an example, let's look at the sizes of cities and towns. The US Census Bureau publishes the population of every incorporated city and town in the United States.

I downloaded their data; it is in the repository for this book in a file named `PEP_2012_PEPANNRES_with_ann.csv`. The repository also contains `populations.py`, which reads the file and plots the distribution of populations.

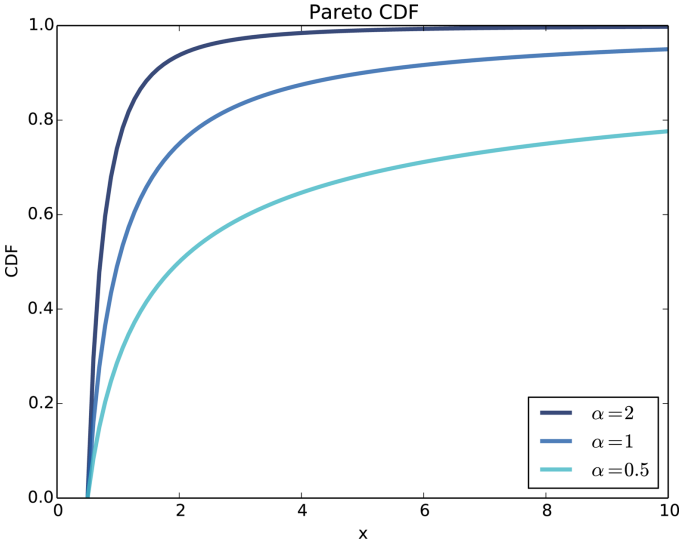


Figure 5-9. CDFs of Pareto distributions with different parameters.

Figure 5-10 shows the CCDF of populations on a log-log scale. The largest 1% of cities and towns, below 10^{-2}, fall along a straight line. So we could conclude, as some researchers have, that the tail of this distribution fits a Pareto model.

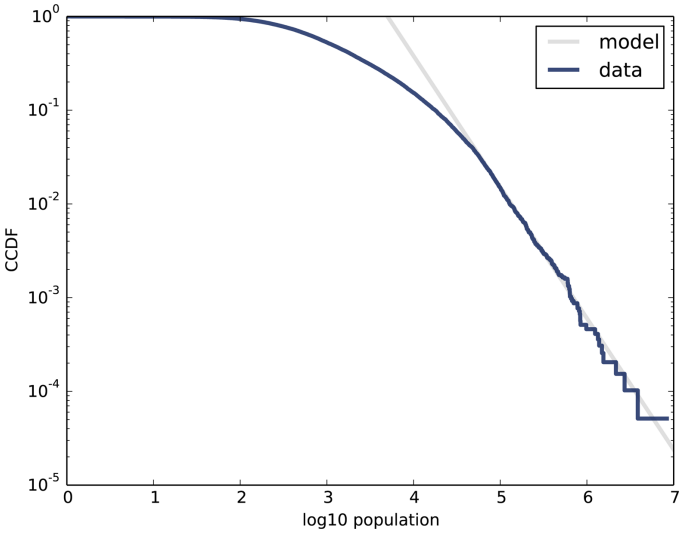


Figure 5-10. CCDFs of city and town populations, on a log-log scale

On the other hand, a lognormal distribution also models the data well. Figure 5-11 shows the CDF of populations and a lognormal model (left), and a normal probability plot (right). Both plots show good agreement between the data and the model.

Neither model is perfect. The Pareto model only applies to the largest 1% of cities, but it is a better fit for that part of the distribution. The lognormal model is a better fit for the other 99%. Which model is appropriate depends on which part of the distribution is relevant.

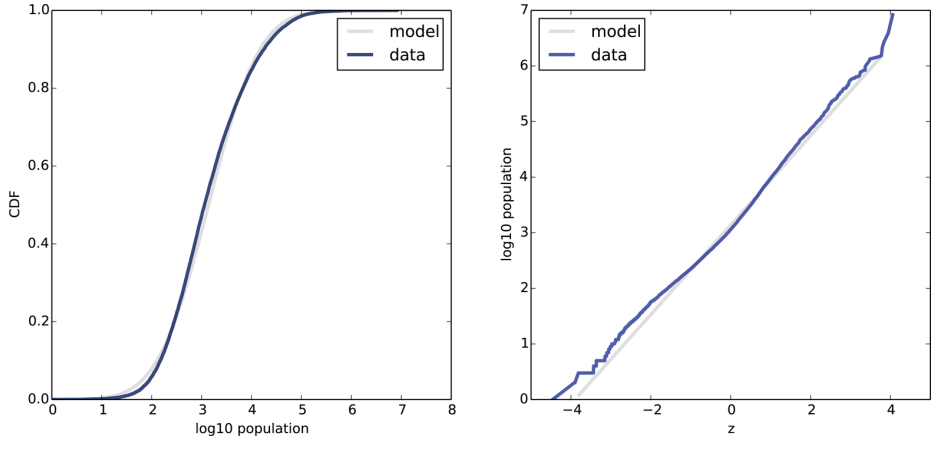


Figure 5-11. CDF of city and town populations on a log-x scale (left), and normal probability plot of log-transformed populations (right)

Generating Random Numbers

Analytic CDFs can be used to generate random numbers with a given distribution function, $p = \mathrm{CDF}(x)$. If there is an efficient way to compute the inverse CDF, we can generate random values with the appropriate distribution by choosing p from a uniform distribution between 0 and 1, and then choosing $x = ICDF(p)$.

For example, the CDF of the exponential distribution is

$$p = 1 - e^{-\lambda x}$$

Solving for x yields:

$$x = -\log(1 - p) / \lambda$$

So in Python we can write

```
def expovariate(lam):
    p = random.random()
    x = -math.log(1-p) / lam
    return x
```

`expovariate` takes `lam` and returns a random value chosen from the exponential distribution with parameter `lam`.

Two notes about this implementation: I called the parameter `lam` because `lambda` is a Python keyword. Also, since $\log 0$ is undefined, we have to be a little careful. The implementation of `random.random` can return 0 but not 1, so $1 - p$ can be 1 but not 0, so `log(1-p)` is always defined.

Why Model?

At the beginning of this chapter, I said that many real world phenomena can be modeled with analytic distributions. "So," you might ask, "what?"

Like all models, analytic distributions are abstractions, which means they leave out details that are considered irrelevant. For example, an observed distribution might have measurement errors or quirks that are specific to the sample; analytic models smooth out these idiosyncrasies.

Analytic models are also a form of data compression. When a model fits a dataset well, a small set of parameters can summarize a large amount of data.

It is sometimes surprising when data from a natural phenomenon fit an analytic distribution, but these observations can provide insight into physical systems. Sometimes we can explain why an observed distribution has a particular form. For example, Pareto distributions are often the result of generative processes with positive feedback (so-called preferential attachment processes).

Also, analytic distributions lend themselves to mathematical analysis, as we will see in Chapter 14.

But it is important to remember that all models are imperfect. Data from the real world never fit an analytic distribution perfectly. People sometimes talk as if data are generated by models; for example, they might say that the distribution of human heights is normal, or the distribution of income is lognormal. Taken literally, these claims cannot be true; there are always differences between the real world and mathematical models.

Models are useful if they capture the relevant aspects of the real world and leave out unneeded details. But what is "relevant" or "unneeded" depends on what you are planning to use the model for.

Exercises

For the following exercises, you can start with `chap05ex.ipynb`. My solution is in `chap05soln.ipynb`.

Exercise 5-1.

In the BRFSS (see "The lognormal Distribution" on page 55), the distribution of heights is roughly normal with parameters $\mu = 178$ cm and $\sigma = 7.7$ cm for men, and $\mu = 163$ cm and $\sigma = 7.3$ cm for women.

In order to join Blue Man Group, you have to be male and between 5′10″ and 6′1″ tall. What percentage of the US male population is in this range? Hint: use `scipy.stats.norm.cdf`.

Exercise 5-2.

To get a feel for the Pareto distribution, let's see how different the world would be if the distribution of human height were Pareto. With the parameters $x_m = 1$ m and $\alpha = 1.7$, we get a distribution with a reasonable minimum, 1 m, and median, 1.5 m.

Plot this distribution. What is the mean human height in Pareto world? What fraction of the population is shorter than the mean? If there are 7 billion people in Pareto world, how many do we expect to be taller than 1 km? How tall do we expect the tallest person to be?

Exercise 5-3.

The Weibull distribution is a generalization of the exponential distribution that comes up in failure analysis. Its CDF is:

$$CDF(x) = 1 - e^{-(x/\lambda)^k}$$

Can you find a transformation that makes a Weibull distribution look like a straight line? What do the slope and intercept of the line indicate?

Use `random.weibullvariate` to generate a sample from a Weibull distribution and use it to test your transformation.

Exercise 5-4.

For small values of n, we don't expect an empirical distribution to fit an analytic distribution exactly. One way to evaluate the quality of fit is to generate a sample from an analytic distribution and see how well it matches the data.

For example, in "The Exponential Distribution" on page 49 we plotted the distribution of time between births and saw that it is approximately exponential. But the distribution is based on only 44 data points. To see whether the data might have come from an exponential distribution, generate 44 values from an exponential distribution with the same mean as the data, about 33 minutes between births.

Plot the distribution of the random values and compare it to the actual distribution. You can use `random.expovariate` to generate the values.

Exercise 5-5.

In the repository for this book, you'll find a set of data files called `mystery0.dat`, `mystery1.dat`, and so on. Each contains a sequence of random numbers generated from an analytic distribution.

You will also find `test_models.py`, a script that reads data from a file and plots the CDF under a variety of transforms. You can run it like this:

```
$ python test_models.py mystery0.dat
```

Based on these plots, you should be able to infer what kind of distribution generated each file. If you are stumped, you can look in `mystery.py`, which contains the code that generated the files.

Exercise 5-6.

The distributions of wealth and income are sometimes modeled using lognormal and Pareto distributions. To see which is better, let's look at some data.

The Current Population Survey (CPS) is a joint effort of the Bureau of Labor Statistics and the Census Bureau to study income and related variables. Data collected in 2013 is available from the Census Bureau. I downloaded `hinc06.xls`, which is an Excel spreadsheet with information about household income, and converted it to `hinc06.csv`, a CSV file you will find in the repository for this book. You will also find `hinc.py`, which reads this file.

Extract the distribution of incomes from this dataset. Are any of the analytic distributions in this chapter a good model of the data? A solution to this exercise is in `hinc_soln.py` .

Glossary

empirical distribution
: The distribution of values in a sample.

analytic distribution
: A distribution whose CDF is an analytic function.

model
: A useful simplification. Analytic distributions are often good models of more complex empirical distributions.

interarrival time
: The elapsed time between two events.

complementary CDF
: A function that maps from a value, x, to the fraction of values that exceed x, which is 1 - CDF(x).

standard normal distribution
: The normal distribution with mean 0 and standard deviation 1.

normal probability plot
: A plot of the values in a sample versus random values from a standard normal distribution.

CHAPTER 6

Probability Density Functions

The code for this chapter is in `density.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

PDFs

The derivative of a CDF is called a **probability density function**, or PDF. For example, the PDF of an exponential distribution is

$$\text{PDF}_{expo}(x) = \lambda e^{-\lambda x}$$

The PDF of a normal distribution is

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

Evaluating a PDF for a particular value of x is usually not useful. The result is not a probability; it is a probability *density*.

In physics, density is mass per unit of volume; in order to get a mass, you have to multiply by volume or, if the density is not constant, you have to integrate over volume.

Similarly, **probability density** measures probability per unit of x. In order to get a probability mass, you have to integrate over x.

`thinkstats2` provides a class called Pdf that represents a probability density function. Every Pdf object provides the following methods:

- `Density`, which takes a value, `x`, and returns the density of the distribution at `x`.
- `Render`, which evaluates the density at a discrete set of values and returns a pair of sequences: the sorted values, `xs`, and their probability densities, `ds`.

- `MakePmf`, which evaluates `Density` at a discrete set of values and returns a normalized Pmf that approximates the Pdf.
- `GetLinspace`, which returns the default set of points used by `Render` and `MakePmf`.

Pdf is an abstract parent class, which means you should not instantiate it; that is, you cannot create a Pdf object. Instead, you should define a child class that inherits from Pdf and provides definitions of `Density` and `GetLinspace`. Pdf provides `Render` and `MakePmf`.

For example, `thinkstats2` provides a class named `NormalPdf` that evaluates the normal density function.

```
class NormalPdf(Pdf):

    def __init__(self, mu=0, sigma=1, label=''):
        self.mu = mu
        self.sigma = sigma
        self.label = label

    def Density(self, xs):
        return scipy.stats.norm.pdf(xs, self.mu, self.sigma)

    def GetLinspace(self):
        low, high = self.mu-3*self.sigma, self.mu+3*self.sigma
        return np.linspace(low, high, 101)
```

The NormalPdf object contains the parameters `mu` and `sigma`. `Density` uses `scipy.stats.norm`, which is an object that represents a normal distribution and provides `cdf` and `pdf`, among other methods (see "The Normal Distribution" on page 52).

The following example creates a NormalPdf with the mean and variance of adult female heights, in cm, from the BRFSS (see "The lognormal Distribution" on page 55). Then it computes the density of the distribution at a location one standard deviation from the mean.

```
>>> mean, var = 163, 52.8
>>> std = math.sqrt(var)
>>> pdf = thinkstats2.NormalPdf(mean, std)
>>> pdf.Density(mean + std)
0.0333001
```

The result is about 0.03, in units of probability mass per cm. Again, a probability density doesn't mean much by itself. But if we plot the Pdf, we can see the shape of the distribution:

```
>>> thinkplot.Pdf(pdf, label='normal')
>>> thinkplot.Show()
```

`thinkplot.Pdf` plots the Pdf as a smooth function, as contrasted with `thinkplot.Pmf`, which renders a Pmf as a step function. Figure 6-1 shows the result, as well as a PDF estimated from a sample, which we'll compute in the next section.

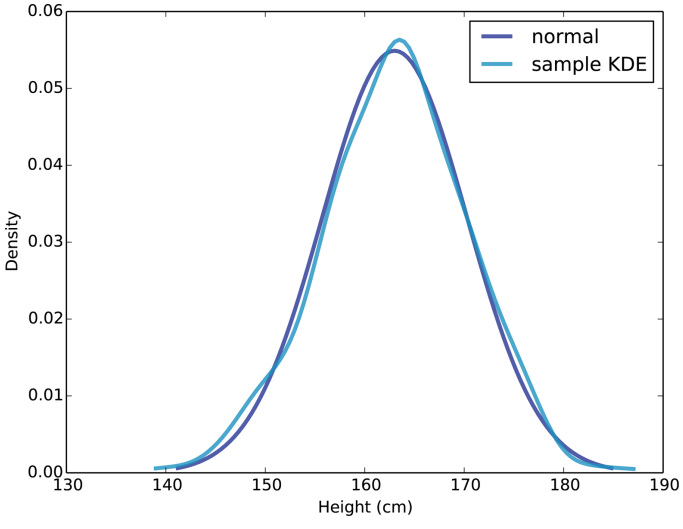


Figure 6-1. A normal PDF that models adult female height in the US, and the kernel density estimate of a sample with $n = 500$.

You can use `MakePmf` to approximate the Pdf:

```
>>> pmf = pdf.MakePmf()
```

By default, the resulting Pmf contains 101 points equally spaced from `mu - 3*sigma` to `mu + 3*sigma`. Optionally, `MakePmf` and `Render` can take keyword arguments `low`, `high`, and `n`.

Kernel Density Estimation

Kernel density estimation (KDE) is an algorithm that takes a sample and finds an appropriately smooth PDF that fits the data. You can read details at Wikipedia.

scipy provides an implementation of KDE and thinkstats2 provides a class called EstimatedPdf that uses it:

```
class EstimatedPdf(Pdf):

    def __init__(self, sample):
        self.kde = scipy.stats.gaussian_kde(sample)

    def Density(self, xs):
        return self.kde.evaluate(xs)
```

__init__ takes a sample and computes a kernel density estimate. The result is a gaussian_kde object that provides an evaluate method.

Density takes a value or sequence, calls gaussian_kde.evaluate, and returns the resulting density. The word "Gaussian" appears in the name because it uses a filter based on a Gaussian distribution to smooth the KDE.

Here's an example that generates a sample from a normal distribution and then makes an EstimatedPdf to fit it:

```
>>> sample = [random.gauss(mean, std) for i in range(500)]
>>> sample_pdf = thinkstats2.EstimatedPdf(sample)
>>> thinkplot.Pdf(pdf, label='sample KDE')
```

sample is a list of 500 random heights. sample_pdf is a Pdf object that contains the estimated KDE of the sample. pmf is a Pmf object that approximates the Pdf by evaluating the density at a sequence of equally spaced values.

Figure 6-1 shows the normal density function and a KDE based on a sample of 500 random heights. The estimate is a good match for the original distribution.

Estimating a density function with KDE is useful for several purposes:

Visualization
: During the exploration phase of a project, CDFs are usually the best visualization of a distribution. After you look at a CDF, you can decide whether an estimated PDF is an appropriate model of the distribution. If so, it can be a better choice for presenting the distribution to an audience that is unfamiliar with CDFs.

Interpolation
: An estimated PDF is a way to get from a sample to a model of the population. If you have reason to believe that the population distribution is smooth, you can use KDE to interpolate the density for values that don't appear in the sample.

Simulation
: Simulations are often based on the distribution of a sample. If the sample size is small, it might be appropriate to smooth the sample distribution using KDE, which allows the simulation to explore more possible outcomes, rather than replicating the observed data.

The Distribution Framework

At this point we have seen PMFs, CDFs and PDFs; let's take a minute to review. Figure 6-2 shows how these functions relate to each other.

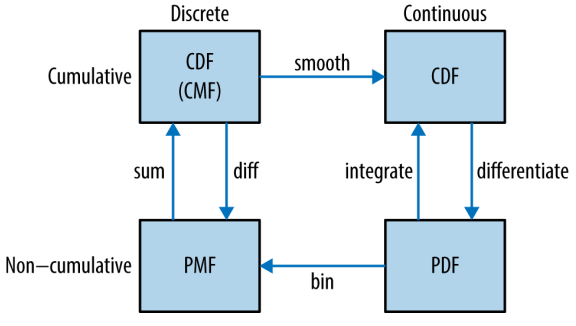


Figure 6-2. A framework that relates representations of distribution functions

We started with PMFs, which represent the probabilities for a discrete set of values. To get from a PMF to a CDF, you add up the probability masses to get cumulative probabilities. To get from a CDF back to a PMF, you compute differences in cumulative probabilities. We'll see the implementation of these operations in the next few sections.

A PDF is the derivative of a continuous CDF; or, equivalently, a CDF is the integral of a PDF. Remember that a PDF maps from values to probability densities; to get a probability, you have to integrate.

To get from a discrete to a continuous distribution, you can perform various kinds of smoothing. One form of smoothing is to assume that the data come from an analytic continuous distribution (like exponential or normal) and to estimate the parameters of that distribution. Another option is kernel density estimation.

The opposite of smoothing is **discretizing**, or quantizing. If you evaluate a PDF at discrete points, you can generate a PMF that is an approximation of the PDF. You can get a better approximation using numerical integration.

To distinguish between continuous and discrete CDFs, it might be better for a discrete CDF to be a "cumulative mass function," but as far as I can tell no one uses that term.

Hist Implementation

At this point you should know how to use the basic types provided by `thinkstats2`: Hist, Pmf, Cdf, and Pdf. The next few sections provide details about how they are implemented. This material might help you use these classes more effectively, but it is not strictly necessary.

Hist and Pmf inherit from a parent class called `_DictWrapper`. The leading underscore indicates that this class is "internal"; that is, it should not be used by code in other modules. The name indicates what it is: a dictionary wrapper. Its primary attribute is `d`, the dictionary that maps from values to their frequencies.

The values can be any hashable type. The frequencies should be integers, but can be any numeric type.

`_DictWrapper` contains methods appropriate for both Hist and Pmf, including `__init__`, `Values`, `Items` and `Render`. It also provides modifier methods `Set`, `Incr`, `Mult`, and `Remove`. These methods are all implemented with dictionary operations. For example:

```
# class _DictWrapper

    def Incr(self, x, term=1):
        self.d[x] = self.d.get(x, 0) + term

    def Mult(self, x, factor):
        self.d[x] = self.d.get(x, 0) * factor

    def Remove(self, x):
        del self.d[x]
```

Hist also provides `Freq`, which looks up the frequency of a given value.

Because Hist operators and methods are based on dictionaries, these methods are constant time operations; that is, their runtime does not increase as the Hist gets bigger.

Pmf Implementation

Pmf and Hist are almost the same thing, except that a Pmf maps values to floating-point probabilities, rather than integer frequencies. If the sum of the probabilities is 1, the Pmf is normalized.

Pmf provides `Normalize`, which computes the sum of the probabilities and divides through by a factor:

```
# class Pmf

    def Normalize(self, fraction=1.0):
        total = self.Total()
        if total == 0.0:
            raise ValueError('Total probability is zero.')

        factor = float(fraction) / total
        for x in self.d:
            self.d[x] *= factor

        return total
```

`fraction` determines the sum of the probabilities after normalizing; the default value is 1. If the total probability is 0, the Pmf cannot be normalized, so `Normalize` raises `ValueError`.

Hist and Pmf have the same constructor. It can take as an argument a `dict`, Hist, Pmf or Cdf, a pandas Series, a list of (value, frequency) pairs, or a sequence of values.

If you instantiate a Pmf, the result is normalized. If you instantiate a Hist, it is not. To construct an unnormalized Pmf, you can create an empty Pmf and modify it. The Pmf modifiers do not renormalize the Pmf.

Cdf Implementation

A CDF maps from values to cumulative probabilities, so I could have implemented Cdf as a `_DictWrapper`. But the values in a CDF are ordered and the values in a `_DictWrapper` are not. Also, it is often useful to compute the inverse CDF; that is, the map from cumulative probability to value. So the implementaion I chose is two sorted lists. That way I can use binary search to do a forward or inverse lookup in logarithmic time.

The Cdf constructor can take as a parameter a sequence of values or a pandas Series, a dictionary that maps from values to probabilities, a sequence of (value, probability) pairs, a Hist, Pmf, or Cdf. Or if it is given two parameters, it treats them as a sorted sequence of values and the sequence of corresponding cumulative probabilities.

Given a sequence, pandas Series, or dictionary, the constructor makes a Hist. Then it uses the Hist to initialize the attributes:

```
        self.xs, freqs = zip(*sorted(dw.Items()))
        self.ps = np.cumsum(freqs, dtype=np.float)
        self.ps /= self.ps[-1]
```

`xs` is the sorted list of values; `freqs` is the list of corresponding frequencies. `np.cumsum` computes the cumulative sum of the frequencies. Dividing through by the total frequency yields cumulative probabilities. For `n` values, the time to construct the Cdf is proportional to $n \log n$.

Here is the implementation of `Prob`, which takes a value and returns its cumulative probability:

```
# class Cdf
    def Prob(self, x):
        if x < self.xs[0]:
            return 0.0
        index = bisect.bisect(self.xs, x)
        p = self.ps[index - 1]
        return p
```

The `bisect` module provides an implementation of binary search. And here is the implementation of `Value`, which takes a cumulative probability and returns the corresponding value:

```
# class Cdf
    def Value(self, p):
        if p < 0 or p > 1:
            raise ValueError('p must be in range [0, 1]')

        index = bisect.bisect_left(self.ps, p)
        return self.xs[index]
```

Given a Cdf, we can compute the Pmf by computing differences between consecutive cumulative probabilities. If you call the Cdf constructor and pass a Pmf, it computes differences by calling `Cdf.Items`:

```
# class Cdf
    def Items(self):
        a = self.ps
        b = np.roll(a, 1)
        b[0] = 0
        return zip(self.xs, a-b)
```

`np.roll` shifts the elements of `a` to the right, and "rolls" the last one back to the beginning. We replace the first element of `b` with 0 and then compute the difference `a-b`. The result is a NumPy array of probabilities.

Cdf provides `Shift` and `Scale`, which modify the values in the Cdf, but the probabilities should be treated as immutable.

Moments

Any time you take a sample and reduce it to a single number, that number is a statistic. The statistics we have seen so far include mean, variance, median, and interquartile range.

A **raw moment** is a kind of statistic. If you have a sample of values, x_i, the kth raw moment is:

$$m_k' = \frac{1}{n}\sum_i x_i^k$$

Or if you prefer Python notation:

```
def RawMoment(xs, k):
    return sum(x**k for x in xs) / len(xs)
```

When $k = 1$ the result is the sample mean, $\bar{x}$. The other raw moments don't mean much by themselves, but they are used in some computations.

The **central moments** are more useful. The kth central moment is:

$$m_k = \frac{1}{n}\sum_i (x_i - \bar{x})^k$$

Or in Python:

```
def CentralMoment(xs, k):
    mean = RawMoment(xs, 1)
    return sum((x - mean)**k for x in xs) / len(xs)
```

When $k = 2$ the result is the second central moment, which you might recognize as variance. The definition of variance gives a hint about why these statistics are called moments. If we attach a weight along a ruler at each location, x_i, and then spin the ruler around the mean, the moment of inertia of the spinning weights is the variance of the values. If you are not familiar with moment of inertia, see Wikipedia..

When you report moment-based statistics, it is important to think about the units. For example, if the values x_i are in cm, the first raw moment is also in cm. But the second moment is in cm^2, the third moment is in cm^3, and so on.

Because of these units, moments are hard to interpret by themselves. That's why, for the second moment, it is common to report standard deviation, which is the square root of variance, so it is in the same units as x_i.

Skewness

Skewness is a property that describes the shape of a distribution. If the distribution is symmetric around its central tendency, it is unskewed. If the values extend farther to the right, it is "right skewed" and if the values extend left, it is "left skewed."

This use of "skewed" does not have the usual connotation of "biased." Skewness only describes the shape of the distribution; it says nothing about whether the sampling process might have been biased.

Several statistics are commonly used to quantify the skewness of a distribution. Given a sequence of values, x_i, the **sample skewness**, g_1, can be computed like this:

```
def StandardizedMoment(xs, k):
    var = CentralMoment(xs, 2)
    std = math.sqrt(var)
    return CentralMoment(xs, k) / std**k

def Skewness(xs):
    return StandardizedMoment(xs, 3)
```

g_1 is the third **standardized moment**, which means that it has been normalized so it has no units.

Negative skewness indicates that a distribution skews left; positive skewness indicates that a distribution skews right. The magnitude of g_1 indicates the strength of the skewness, but by itself it is not easy to interpret.

In practice, computing sample skewness is usually not a good idea. If there are any outliers, they have a disproportionate effect on g_1.

Another way to evaluate the asymmetry of a distribution is to look at the relationship between the mean and the median. Extreme values have more effect on the mean than the median, so in a distribution that skews left, the mean is less than the median. In a distribution that skews right, the mean is greater.

Pearson's median skewness coefficient is a measure of skewness based on the difference between the sample mean and median:

$$g_p = 3(\bar{x} - m) / S$$

Where $\bar{x}$ is the sample mean, m is the median, and S is the standard deviation. Or in Python:

```
def Median(xs):
    cdf = thinkstats2.MakeCdfFromList(xs)
    return cdf.Value(0.5)

def PearsonMedianSkewness(xs):
    median = Median(xs)
    mean = RawMoment(xs, 1)
    var = CentralMoment(xs, 2)
    std = math.sqrt(var)
    gp = 3 * (mean - median) / std
    return gp
```

This statistic is **robust**, which means that it is less vulnerable to the effect of outliers.

As an example, let's look at the skewness of birth weights in the NSFG pregnancy data. Here's the code to estimate and plot the PDF:

```
live, firsts, others = first.MakeFrames()
data = live.totalwgt_lb.dropna()
pdf = thinkstats2.EstimatedPdf(data)
thinkplot.Pdf(pdf, label='birth weight')
```

Figure 6-3 shows the result. The left tail appears longer than the right, so we suspect the distribution is skewed left. The mean, 7.27 lbs, is a bit less than the median, 7.38 lbs, so that is consistent with left skew. And both skewness coefficients are negative: sample skewness is -0.59; Pearson's median skewness is -0.23.

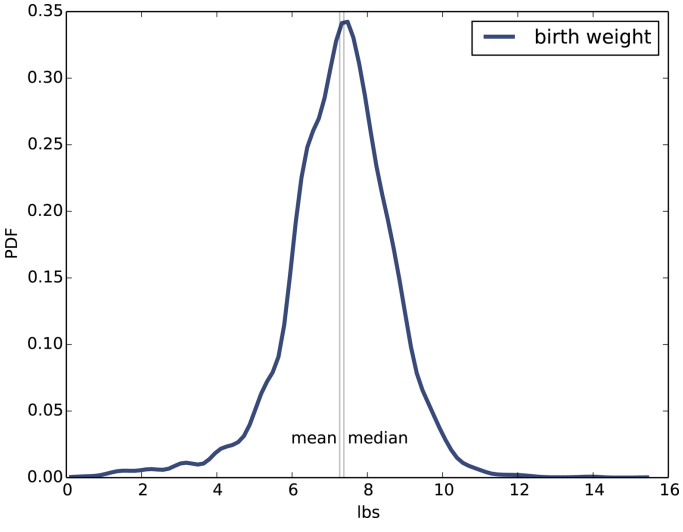


Figure 6-3. Estimated PDF of birthweight data from the NSFG

Now let's compare this distribution to the distribution of adult weight in the BRFSS. Again, here's the code:

```
df = brfss.ReadBrfss(nrows=None)
data = df.wtkg2.dropna()
pdf = thinkstats2.EstimatedPdf(data)
thinkplot.Pdf(pdf, label='adult weight')
```

Figure 6-4 shows the result. The distribution appears skewed to the right. Sure enough, the mean, 79.0, is bigger than the median, 77.3. The sample skewness is 1.1 and Pearson's median skewness is 0.26.

The sign of the skewness coefficient indicates whether the distribution skews left or right, but other than that, they are hard to interpret. Sample skewness is less robust; that is, it is more susceptible to outliers. As a result it is less reliable when applied to skewed distributions, exactly when it would be most relevant.

Pearson's median skewness is based on a computed mean and variance, so it is also susceptible to outliers, but since it does not depend on a third moment, it is somewhat more robust.

Exercises

A solution to this exercise is in `chap06soln.py`.

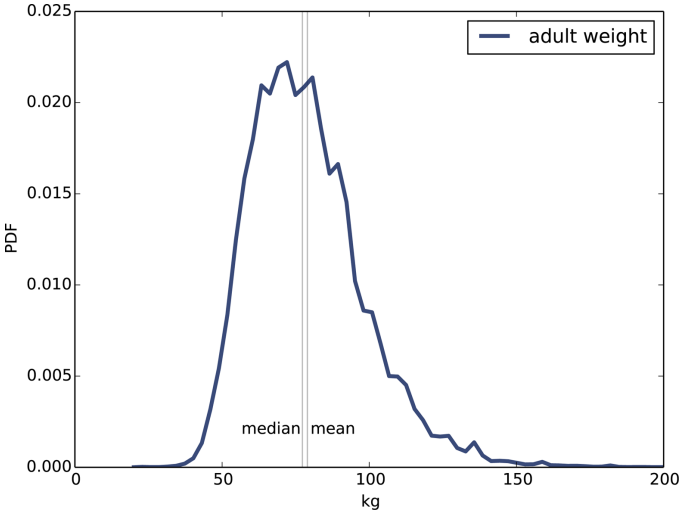


Figure 6-4. Estimated PDF of adult weight data from the BRFSS.

Exercise 6-1.

The distribution of income is famously skewed to the right. In this exercise, we'll measure how strong that skew is.

The Current Population Survey (CPS) is a joint effort of the Bureau of Labor Statistics and the Census Bureau to study income and related variables. Data collected in 2013 is available from the Census Burea's website. I downloaded `hinc06.xls`, which is an Excel spreadsheet with information about household income, and converted it to `hinc06.csv`, a CSV file you will find in the repository for this book. You will also find `hinc2.py`, which reads this file and transforms the data.

The dataset is in the form of a series of income ranges and the number of respondents who fell in each range. The lowest range includes respondents who reported annual household income "Under $5000." The highest range includes respondents who made "$250,000 or more."

To estimate mean and other statistics from these data, we have to make some assumptions about the lower and upper bounds, and how the values are distributed in each range. `hinc2.py` provides `InterpolateSample`, which shows one way to model this data. It takes a DataFrame with a column, `income`, that contains the upper bound of each range, and `freq`, which contains the number of respondents in each frame.

It also takes `log_upper`, which is an assumed upper bound on the highest range, expressed in `log10` dollars. The default value, `log_upper=6.0` represents the assumption that the largest income among the respondents is 10^6, or one million dollars.

`InterpolateSample` generates a pseudo-sample; that is, a sample of household incomes that yields the same number of respondents in each range as the actual data. It assumes that incomes in each range are equally spaced on a log10 scale.

Compute the median, mean, skewness and Pearson's skewness of the resulting sample. What fraction of households reports a taxable income below the mean? How do the results depend on the assumed upper bound?

Glossary

Probability density function (PDF)
: The derivative of a continuous CDF, a function that maps a value to its probability density.

Probability density
: A quantity that can be integrated over a range of values to yield a probability. If the values are in units of cm, for example, probability density is in units of probability per cm.

Kernel density estimation (KDE)
: An algorithm that estimates a PDF based on a sample.

discretize
: To approximate a continuous function or distribution with a discrete function. The opposite of smoothing.

raw moment
: A statistic based on the sum of data raised to a power.

central moment
: A statistic based on deviation from the mean, raised to a power.

standardized moment
: A ratio of moments that has no units.

skewness
: A measure of how asymmetric a distribution is.

sample skewness
: A moment-based statistic intended to quantify the skewness of a distribution.

Pearson's median skewness coefficient
: A statistic intended to quantify the skewness of a distribution based on the median, mean, and standard deviation.

robust
: A statistic is robust if it is relatively immune to the effect of outliers.

CHAPTER 7

Relationships Between Variables

So far we have only looked at one variable at a time. In this chapter we look at relationships between variables. Two variables are related if knowing one gives you information about the other. For example, height and weight are related; people who are taller tend to be heavier. Of course, it is not a perfect relationship: there are short heavy people and tall light ones. But if you are trying to guess someone's weight, you will be more accurate if you know their height than if you don't.

The code for this chapter is in `scatter.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

Scatter Plots

The simplest way to check for a relationship between two variables is a **scatter plot**, but making a good scatter plot is not always easy. As an example, I'll plot weight versus height for the respondents in the BRFSS (see "The lognormal Distribution" on page 55).

Here's the code that reads the data file and extracts height and weight:

```
df = brfss.ReadBrfss(nrows=None)
sample = thinkstats2.SampleRows(df, 5000)
heights, weights = sample.htm3, sample.wtkg2
```

`SampleRows` chooses a random subset of the data:

```
def SampleRows(df, nrows, replace=False):
    indices = np.random.choice(df.index, nrows, replace=replace)
    sample = df.loc[indices]
    return sample
```

`df` is the DataFrame, `nrows` is the number of rows to choose, and `replace` is a boolean indicating whether sampling should be done with replacement; in other words, whether the same row could be chosen more than once.

`thinkplot` provides `Scatter`, which makes scatter plots:

```
thinkplot.Scatter(heights, weights)
thinkplot.Show(xlabel='Height (cm)',
               ylabel='Weight (kg)',
               axis=[140, 210, 20, 200])
```

The result, in Figure 7-1 (left), shows the shape of the relationship. As we expected, taller people tend to be heavier.

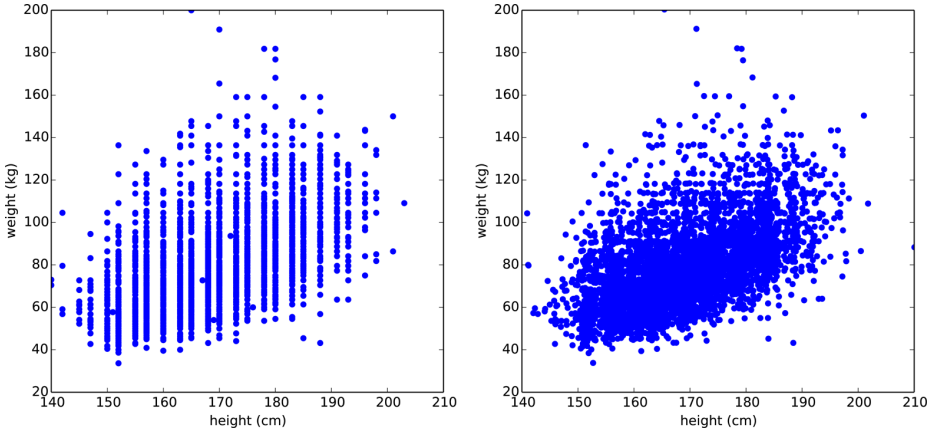


Figure 7-1. Scatter plots of weight versus height for the respondents in the BRFSS, unjittered (left), jittered (right).

But this is not the best representation of the data, because the data are packed into columns. The problem is that the heights are rounded to the nearest inch, converted to centimeters, and then rounded again. Some information is lost in translation.

We can't get that information back, but we can minimize the effect on the scatter plot by **jittering** the data, which means adding random noise to reverse the effect of rounding off. Since these measurements were rounded to the nearest inch, they might be off by up to 0.5 inches or 1.3 cm. Similarly, the weights might be off by 0.5 kg.

```
heights = thinkstats2.Jitter(heights, 1.3)
weights = thinkstats2.Jitter(weights, 0.5)
```

Here's the implementation of `Jitter`:

```
def Jitter(values, jitter=0.5):
    n = len(values)
    return np.random.uniform(-jitter, +jitter, n) + values
```

The values can be any sequence; the result is a NumPy array.

Figure 7-1 (right) shows the result. Jittering reduces the visual effect of rounding and makes the shape of the relationship clearer. But in general you should only jitter data for purposes of visualization and avoid using jittered data for analysis.

Even with jittering, this is not the best way to represent the data. There are many overlapping points, which hides data in the dense parts of the figure and gives disproportionate emphasis to outliers. This effect is called **saturation**.

We can solve this problem with the `alpha` parameter, which makes the points partly transparent:

```
thinkplot.Scatter(heights, weights, alpha=0.2)
```

Figure 7-2 (left) shows the result. Overlapping data points look darker, so darkness is proportional to density. In this version of the plot we can see two details that were not apparent before: vertical clusters at several heights and a horizontal line near 90 kg or 200 pounds. Since this data is based on self-reports in pounds, the most likely explanation is that some respondents reported rounded values.

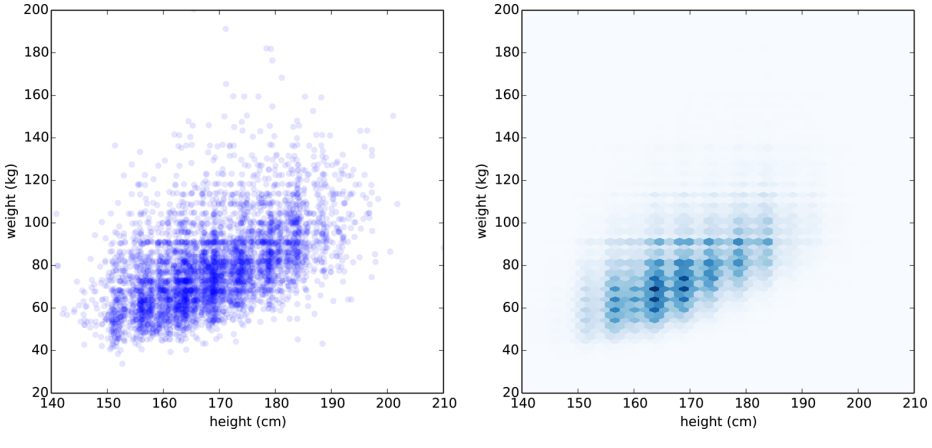


Figure 7-2. Scatter plot with jittering and transparency (left), hexbin plot (right).

Using transparency works well for moderate-sized datasets, but this figure only shows the first 5000 records in the BRFSS, out of a total of 414,509.

To handle larger datasets, another option is a hexbin plot, which divides the graph into hexagonal bins and colors each bin according to how many data points fall in it. `thinkplot` provides `HexBin`:

```
thinkplot.HexBin(heights, weights)
```

Figure 7-2 (right) shows the result. An advantage of a hexbin is that it shows the shape of the relationship well, and it is efficient for large datasets, both in time and in the size of the file it generates. A drawback is that it makes the outliers invisible.

The point of this example is that it is not easy to make a scatter plot that shows relationships clearly without introducing misleading artifacts.

Characterizing Relationships

Scatter plots provide a general impression of the relationship between variables, but there are other visualizations that provide more insight into the nature of the relationship. One option is to bin one variable and plot percentiles of the other.

NumPy and pandas provide functions for binning data:

```
df = df.dropna(subset=['htm3', 'wtkg2'])
bins = np.arange(135, 210, 5)
indices = np.digitize(df.htm3, bins)
groups = df.groupby(indices)
```

`dropna` drops rows with `nan` in any of the listed columns. `arange` makes a NumPy array of bins from 135 to, but not including, 210, in increments of 5.

`digitize` computes the index of the bin that contains each value in `df.htm3`. The result is a NumPy array of integer indices. Values that fall below the lowest bin are mapped to index 0. Values above the highest bin are mapped to `len(bins)`.

`groupby` is a DataFrame method that returns a GroupBy object; used in a `for` loop, `groups` iterates the names of the groups and the DataFrames that represent them. So, for example, we can print the number of rows in each group like this:

```
for i, group in groups:
    print(i, len(group))
```

Now for each group we can compute the mean height and the CDF of weight:

```
heights = [group.htm3.mean() for i, group in groups]
cdfs = [thinkstats2.Cdf(group.wtkg2) for i, group in groups]
```

Finally, we can plot percentiles of weight versus height:

```
for percent in [75, 50, 25]:
    weights = [cdf.Percentile(percent) for cdf in cdfs]
    label = '%dth' % percent
    thinkplot.Plot(heights, weights, label=label)
```

Figure 7-3 shows the result. Between 140 and 200 cm the relationship between these variables is roughly linear. This range includes more than 99% of the data, so we don't have to worry too much about the extremes.

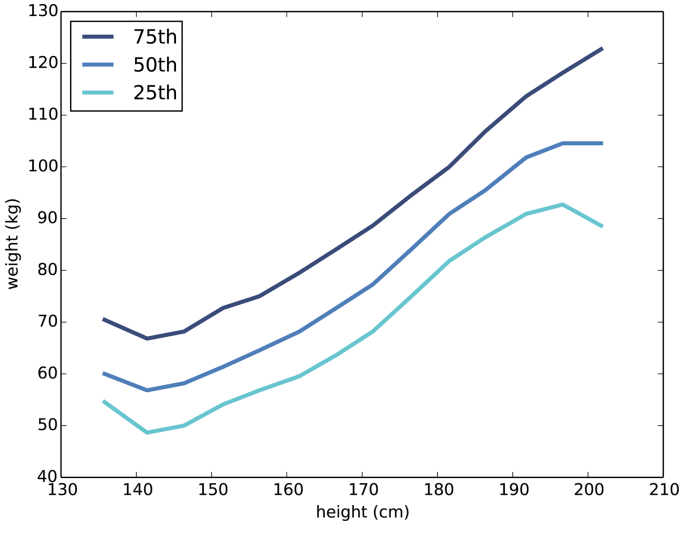


Figure 7-3. Percentiles of weight for a range of height bins.

Correlation

A **correlation** is a statistic intended to quantify the strength of the relationship between two variables.

A challenge in measuring correlation is that the variables we want to compare are often not expressed in the same units. And even if they are in the same units, they come from different distributions.

There are two common solutions to these problems:

- Transform each value to a **standard scores**, which is the number of standard deviations from the mean. This transform leads to the "Pearson product-moment correlation coefficient."
- Transform each value to its **rank**, which is its index in the sorted list of values. This transform leads to the "Spearman rank correlation coefficient."

If X is a series of n values, x_i, we can convert to standard scores by subtracting the mean and dividing by the standard deviation: $z_i = (x_i - \mu) / \sigma$.

The numerator is a deviation: the distance from the mean. Dividing by σ **standardizes** the deviation, so the values of Z are dimensionless (no units) and their distribution has mean 0 and variance 1.

If X is normally distributed, so is Z. But if X is skewed or has outliers, so does Z; in those cases, it is more robust to use percentile ranks. If we compute a new variable, R, so that

r_i is the rank of x_i, the distribution of R is uniform from 1 to n, regardless of the distribution of X.

Covariance

Covariance is a measure of the tendency of two variables to vary together. If we have two series, X and Y, their deviations from the mean are

$$dx_i = x_i - \bar{x}$$

$$dy_i = y_i - \bar{y}$$

where $\bar{x}$ is the sample mean of X and $\bar{y}$ is the sample mean of Y. If X and Y vary together, their deviations tend to have the same sign.

If we multiply them together, the product is positive when the deviations have the same sign and negative when they have the opposite sign. So adding up the products gives a measure of the tendency to vary together.

Covariance is the mean of these products:

$$Cov(X, Y) = \frac{1}{n}\sum dx_i \, dy_i$$

where n is the length of the two series (they have to be the same length).

If you have studied linear algebra, you might recognize that `Cov` is the dot product of the deviations, divided by their length. So the covariance is maximized if the two vectors are identical, 0 if they are orthogonal, and negative if they point in opposite directions. `thinkstats2` uses `np.dot` to implement `Cov` efficiently:

```
def Cov(xs, ys, meanx=None, meany=None):
    xs = np.asarray(xs)
    ys = np.asarray(ys)

    if meanx is None:
        meanx = np.mean(xs)
    if meany is None:
        meany = np.mean(ys)

    cov = np.dot(xs-meanx, ys-meany) / len(xs)
    return cov
```

By default `Cov` computes deviations from the sample means, or you can provide known means. If `xs` and `ys` are Python sequences, `np.asarray` converts them to NumPy arrays. If they are already NumPy arrays, `np.asarray` does nothing.

This implementation of covariance is meant to be simple for purposes of explanation. NumPy and pandas also provide implementations of covariance, but both of them apply

a correction for small sample sizes that we have not covered yet, and `np.cov` returns a covariance matrix, which is more than we need for now.

Pearson's Correlation

Covariance is useful in some computations, but it is seldom reported as a summary statistic because it is hard to interpret. Among other problems, its units are the product of the units of X and Y. For example, the covariance of weight and height in the BRFSS dataset is 113 kilogram-centimeters, whatever that means.

One solution to this problem is to divide the deviations by the standard deviation, which yields standard scores, and compute the product of standard scores:

$$p_i = \frac{(x_i - \bar{x})}{S_X} \frac{(y_i - \bar{y})}{S_Y}$$

Where S_X and S_Y are the standard deviations of X and Y. The mean of these products is

$$\rho = \frac{1}{n} \sum p_i$$

Or we can rewrite ρ by factoring out S_X and S_Y:

$$\rho = \frac{Cov(X, Y)}{S_X S_Y}$$

This value is called **Pearson's correlation** after Karl Pearson, an influential early statistician. It is easy to compute and easy to interpret. Because standard scores are dimensionless, so is ρ.

Here is the implementation in `thinkstats2`:

```
def Corr(xs, ys):
    xs = np.asarray(xs)
    ys = np.asarray(ys)

    meanx, varx = MeanVar(xs)
    meany, vary = MeanVar(ys)

    corr = Cov(xs, ys, meanx, meany) / math.sqrt(varx * vary)
    return corr
```

`MeanVar` computes mean and variance slightly more efficiently than separate calls to `np.mean` and `np.var`.

Pearson's correlation is always between -1 and +1 (including both). If ρ is positive, we say that the correlation is positive, which means that when one variable is high, the other tends to be high. If ρ is negative, the correlation is negative, so when one variable is high, the other is low.

The magnitude of ρ indicates the strength of the correlation. If ρ is 1 or -1, the variables are perfectly correlated, which means that if you know one, you can make a perfect prediction about the other.

Most correlation in the real world is not perfect, but it is still useful. The correlation of height and weight is 0.51, which is a strong correlation compared to similar human-related variables.

Nonlinear Relationships

If Pearson's correlation is near 0, it is tempting to conclude that there is no relationship between the variables, but that conclusion is not valid. Pearson's correlation only measures *linear* relationships. If there's a nonlinear relationship, ρ understates its strength.

Figure 7-4 is from the Wikipedia page for correlation and dependence. It shows scatter plots and correlation coefficients for several carefully constructed datasets.

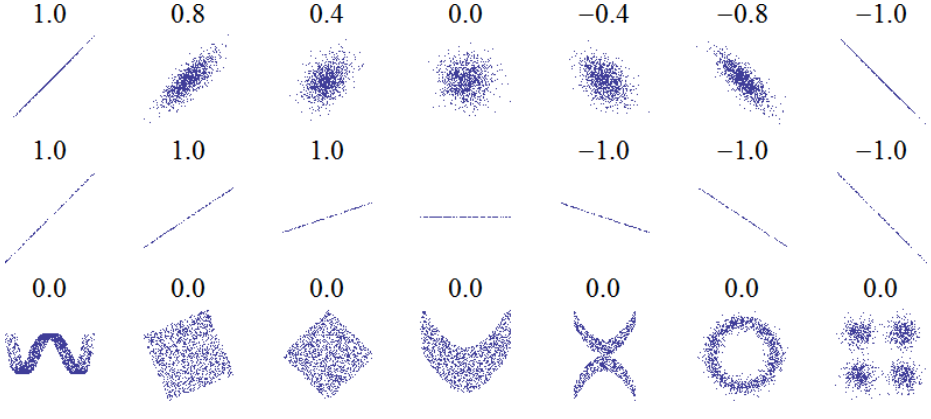


Figure 7-4. Examples of datasets with a range of correlations.

The top row shows linear relationships with a range of correlations; you can use this row to get a sense of what different values of ρ look like. The second row shows perfect correlations with a range of slopes, which demonstrates that correlation is unrelated to slope (we'll talk about estimating slope soon). The third row shows variables that are clearly related, but because the relationship is non-linear, the correlation coefficient is 0.

The moral of this story is that you should always look at a scatter plot of your data before blindly computing a correlation coefficient.

Spearman's Rank Correlation

Pearson's correlation works well if the relationship between variables is linear and if the variables are roughly normal. But it is not robust in the presence of outliers. Spearman's rank correlation is an alternative that mitigates the effect of outliers and skewed distributions. To compute Spearman's correlation, we have to compute the rank of each value, which is its index in the sorted sample. For example, in the sample `[1, 2, 5, 7]` the rank of the value 5 is 3, because it appears third in the sorted list. Then we compute Pearson's correlation for the ranks.

`thinkstats2` provides a function that computes Spearman's rank correlation:

```
def SpearmanCorr(xs, ys):
    xranks = pandas.Series(xs).rank()
    yranks = pandas.Series(ys).rank()
    return Corr(xranks, yranks)
```

I convert the arguments to pandas Series objects so I can use `rank`, which computes the rank for each value and returns a Series. Then I use `Corr` to compute the correlation of the ranks.

I could also use `Series.corr` directly and specify Spearman's method:

```
def SpearmanCorr(xs, ys):
    xranks = pandas.Series(xs)
    yranks = pandas.Series(ys)
    return xs.corr(ys, method='spearman')
```

The Spearman rank correlation for the BRFSS data is 0.54, which is a little higher than the Pearson correlation, 0.51. There are several possible reasons for the difference, including the following:

- If the relationship is nonlinear, Pearson's correlation tends to underestimate the strength of the relationship.
- Pearson's correlation can be affected (in either direction) if one of the distributions is skewed or contains outliers. Spearman's rank correlation is more robust.

In the BRFSS example, we know that the distribution of weights is roughly lognormal; under a log transform, it approximates a normal distribution, so it has no skew. So another way to eliminate the effect of skewness is to compute Pearson's correlation with log-weight and height:

```
thinkstats2.Corr(df.htm3, np.log(df.wtkg2)))
```

The result is 0.53, close to the rank correlation, 0.54. So that suggests that skewness in the distribution of weight explains most of the difference between Pearson's and Spearman's correlation.

Correlation and Causation

If variables A and B are correlated, there are three possible explanations: A causes B, or B causes A, or some other set of factors causes both A and B. These explanations are called "causal relationships".

Correlation alone does not distinguish between these explanations, so it does not tell you which ones are true. This rule is often summarized with the phrase "Correlation does not imply causation," which is so pithy, it has its own Wikipedia page.

So what can you do to provide evidence of causation?

- Use time. If A comes before B, then A can cause B but not the other way around (at least according to our common understanding of causation). The order of events can help us infer the direction of causation, but it does not preclude the possibility that something else causes both A and B.
- Use randomness. If you divide a large sample into two groups at random and compute the means of almost any variable, you expect the difference to be small. If the groups are nearly identical in all variables but one, you can eliminate spurious relationships.

  This works even if you don't know what the relevant variables are, but it works even better if you do, because you can check that the groups are identical.

These ideas are the motivation for the **randomized controlled trial**, in which subjects are assigned randomly to two (or more) groups: a **treatment group** that receives some kind of intervention, like a new medicine, and a **control group** that receives no intervention, or another treatment whose effects are known.

A randomized controlled trial is the most reliable way to demonstrate a causal relationship, and it is the foundation of science-based medicine.

Unfortunately, controlled trials are only possible in the laboratory sciences, medicine, and a few other disciplines. In the social sciences, controlled experiments are rare, usually because they are impossible or unethical.

An alternative is to look for a **natural experiment**, where different "treatments" are applied to groups that are otherwise similar. One danger of natural experiments is that the groups might differ in ways that are not apparent.

In some cases it is possible to infer causal relationships using **regression analysis**, which is the topic of Chapter 11.

Exercises

A solution to this exercise is in `chap07soln.py`.

Exercise 7-1.

Using data from the NSFG, make a scatter plot of birth weight versus mother's age. Plot percentiles of birth weight versus mother's age. Compute Pearson's and Spearman's correlations. How would you characterize the relationship between these variables?

Glossary

scatter plot
: A visualization of the relationship between two variables, showing one point for each row of data.

jitter
: Random noise added to data for purposes of visualization.

saturation
: Loss of information when multiple points are plotted on top of each other.

correlation
: A statistic that measures the strength of the relationship between two variables.

standardize
: To transform a set of values so that their mean is 0 and their variance is 1.

standard score
: A value that has been standardized so that it is expressed in standard deviations from the mean.

covariance
: A measure of the tendency of two variables to vary together.

rank
: The index where an element appears in a sorted list.

randomized controlled trial
: An experimental design in which subjects are divided into groups at random, and different groups are given different treatments.

treatment group
: A group in a controlled trial that receives some kind of intervention.

control group
: A group in a controlled trial that receives no treatment, or a treatment whose effect is known.

natural experiment
: An experimental design that takes advantage of a natural division of subjects into groups in ways that are at least approximately random.

CHAPTER 8
Estimation

The code for this chapter is in `estimation.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

The Estimation Game

Let's play a game. I think of a distribution, and you have to guess what it is. I'll give you two hints: it's a normal distribution, and here's a random sample drawn from it:

```
[-0.441, 1.774, -0.101, -1.138, 2.975, -2.138]
```

What do you think is the mean parameter, μ, of this distribution?

One choice is to use the sample mean, $\bar{x}$, as an estimate of μ. In this example, $\bar{x}$ is 0.155, so it would be reasonable to guess $\mu = 0.155$. This process is called **estimation**, and the statistic we used (the sample mean) is called an **estimator**.

Using the sample mean to estimate μ is so obvious that it is hard to imagine a reasonable alternative. But suppose we change the game by introducing outliers.

I'm thinking of a distribution. It's a normal distribution, and here's a sample that was collected by an unreliable surveyor who occasionally puts the decimal point in the wrong place.

```
[-0.441, 1.774, -0.101, -1.138, 2.975, -213.8]
```

Now what's your estimate of μ? If you use the sample mean, your guess is -35.12. Is that the best choice? What are the alternatives?

One option is to identify and discard outliers, and then compute the sample mean of the rest. Another option is to use the median as an estimator.

Which estimator is best depends on the circumstances (for example, whether there are outliers) and on what the goal is. Are you trying to minimize errors, or maximize your chance of getting the right answer?

If there are no outliers, the sample mean minimizes the **mean squared error** (MSE). That is, if we play the game many times, and each time compute the error $\bar{x} - \mu$, the sample mean minimizes

$$MSE = \frac{1}{m}\sum (\bar{x} - \mu)^2$$

Where m is the number of times you play the estimation game, not to be confused with n, which is the size of the sample used to compute $\bar{x}$.

Here is a function that simulates the estimation game and computes the root mean squared error (RMSE), which is the square root of MSE:

```
def Estimate1(n=7, m=1000):
    mu = 0
    sigma = 1

    means = []
    medians = []
    for _ in range(m):
        xs = [random.gauss(mu, sigma) for i in range(n)]
        xbar = np.mean(xs)
        median = np.median(xs)
        means.append(xbar)
        medians.append(median)

    print('rmse xbar', RMSE(means, mu))
    print('rmse median', RMSE(medians, mu))
```

Again, `n` is the size of the sample, and `m` is the number of times we play the game. `means` is the list of estimates based on $\bar{x}$. `medians` is the list of medians.

Here's the function that computes RMSE:

```
def RMSE(estimates, actual):
    e2 = [(estimate-actual)**2 for estimate in estimates]
    mse = np.mean(e2)
    return math.sqrt(mse)
```

`estimates` is a list of estimates; `actual` is the actual value being estimated. In practice, of course, we don't know `actual`; if we did, we wouldn't have to estimate it. The purpose of this experiment is to compare the performance of the two estimators.

When I ran this code, the RMSE of the sample mean was 0.41, which means that if we use $\bar{x}$ to estimate the mean of this distribution, based on a sample with $n = 7$, we should expect to be off by 0.41 on average. Using the median to estimate the mean yields RMSE 0.53, which confirms that $\bar{x}$ yields lower RMSE, at least for this example.

Minimizing MSE is a nice property, but it's not always the best strategy. For example, suppose we are estimating the distribution of wind speeds at a building site. If the estimate is too high, we might overbuild the structure, increasing its cost. But if it's too low, the building might collapse. Because cost as a function of error is not symmetric, minimizing MSE is not the best strategy.

As another example, suppose I roll three six-sided dice and ask you to predict the total. If you get it exactly right, you get a prize; otherwise you get nothing. In this case the value that minimizes MSE is 10.5, but that would be a bad guess, because the total of three dice is never 10.5. For this game, you want an estimator that has the highest chance of being right, which is a **maximum likelihood estimator** (MLE). If you pick 10 or 11, your chance of winning is 1 in 8, and that's the best you can do.

Guess the Variance

I'm thinking of a distribution. It's a normal distribution, and here's a (familiar) sample:

```
[-0.441, 1.774, -0.101, -1.138, 2.975, -2.138]
```

What do you think is the variance, σ^2, of my distribution? Again, the obvious choice is to use the sample variance, S^2, as an estimator.

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

For large samples, S^2 is an adequate estimator, but for small samples it tends to be too low. Because of this unfortunate property, it is called a **biased** estimator. An estimator is **unbiased** if the expected total (or mean) error, after many iterations of the estimation game, is 0.

Fortunately, there is another simple statistic that is an unbiased estimator of σ^2:

$$S_{n-1}^2 = \frac{1}{n-1}\sum (x_i - \bar{x})^2$$

For an explanation of why S^2 is biased, and a proof that S_{n-1}^2 is unbiased, see this Wikipedia page.

The biggest problem with this estimator is that its name and symbol are used inconsistently. The name "sample variance" can refer to either S^2 or S_{n-1}^2, and the symbol S^2 is used for either or both.

Here is a function that simulates the estimation game and tests the performance of S^2 and S_{n-1}^2:

```
def Estimate2(n=7, m=1000):
    mu = 0
    sigma = 1
```

```
estimates1 = []
estimates2 = []
for _ in range(m):
    xs = [random.gauss(mu, sigma) for i in range(n)]
    biased = np.var(xs)
    unbiased = np.var(xs, ddof=1)
    estimates1.append(biased)
    estimates2.append(unbiased)

print('mean error biased', MeanError(estimates1, sigma**2))
print('mean error unbiased', MeanError(estimates2, sigma**2))
```

Again, `n` is the sample size and `m` is the number of times we play the game. `np.var` computes S^2 by default and S_{n-1}^2 if you provide the argument `ddof=1`, which stands for "delta degrees of freedom." I won't explain that term, but you can read about it at Wikipedia.

`MeanError` computes the mean difference between the estimates and the actual value:

```
def MeanError(estimates, actual):
    errors = [estimate-actual for estimate in estimates]
    return np.mean(errors)
```

When I ran this code, the mean error for S^2 was -0.13. As expected, this biased estimator tends to be too low. For S_{n-1}^2, the mean error was 0.014, about 10 times smaller. As `m` increases, we expect the mean error for S_{n-1}^2 to approach 0.

Properties like MSE and bias are long-term expectations based on many iterations of the estimation game. By running simulations like the ones in this chapter, we can compare estimators and check whether they have desired properties.

But when you apply an estimator to real data, you just get one estimate. It would not be meaningful to say that the estimate is unbiased; being unbiased is a property of the estimator, not the estimate.

After you choose an estimator with appropriate properties, and use it to generate an estimate, the next step is to characterize the uncertainty of the estimate, which is the topic of the next section.

Sampling Distributions

Suppose you are a scientist studying gorillas in a wildlife preserve. You want to know the average weight of the adult female gorillas in the preserve. To weigh them, you have to tranquilize them, which is dangerous, expensive, and possibly harmful to the gorillas. But if it is important to obtain this information, it might be acceptable to weigh a sample of 9 gorillas. Let's assume that the population of the preserve is well known, so we can choose a representative sample of adult females. We could use the sample mean, $\bar{x}$, to estimate the unknown population mean, μ.

Having weighed 9 female gorillas, you might find $\bar{x} = 90$ kg and sample standard deviation, $S = 7.5$ kg. The sample mean is an unbiased estimator of μ, and in the long run it minimizes MSE. So if you report a single estimate that summarizes the results, you would report 90 kg.

But how confident should you be in this estimate? If you only weigh $n = 9$ gorillas out of a much larger population, you might be unlucky and choose the 9 heaviest gorillas (or the 9 lightest ones) just by chance. Variation in the estimate caused by random selection is called **sampling error**.

To quantify sampling error, we can simulate the sampling process with hypothetical values of μ and σ, and see how much $\bar{x}$ varies.

Since we don't know the actual values of μ and σ in the population, we'll use the estimates $\bar{x}$ and S. So the question we answer is: "If the actual values of μ and σ were 90 kg and 7.5 kg, and we ran the same experiment many times, how much would the estimated mean, $\bar{x}$, vary?"

The following function answers that question:

```
def SimulateSample(mu=90, sigma=7.5, n=9, m=1000):
    means = []
    for j in range(m):
        xs = np.random.normal(mu, sigma, n)
        xbar = np.mean(xs)
        means.append(xbar)

    cdf = thinkstats2.MakeCdfFromList(means)
    ci = cdf.Percentile(5), cdf.Percentile(95)
    stderr = RMSE(means, mu)
```

`mu` and `sigma` are the *hypothetical* values of the parameters. `n` is the sample size, the number of gorillas we measured. `m` is the number of times we run the simulation.

In each iteration, we choose `n` values from a normal distribution with the given parameters, and compute the sample mean, `xbar`. We run 1000 simulations and then compute the distribution, `cdf`, of the estimates. The result is shown in Figure 8-1. This distribution is called the **sampling distribution** of the estimator. It shows how much the estimates would vary if we ran the experiment over and over.

The mean of the sampling distribution is pretty close to the hypothetical value of μ, which means that the experiment yields the right answer, on average. After 1,000 tries, the lowest result is 82 kg, and the highest is 98 kg. This range suggests that the estimate might be off by as much as 8 kg.

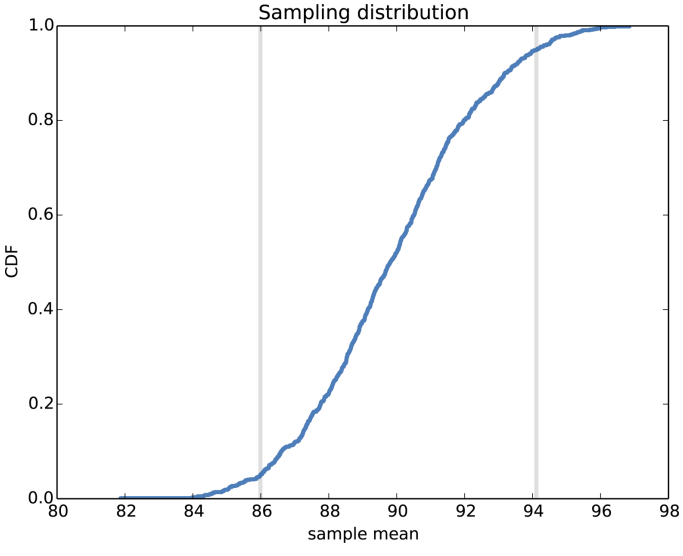


Figure 8-1. Sampling distribution of $\bar{x}$, with confidence interval

There are two common ways to summarize the sampling distribution:

Standard error *(SE)*
: A measure of how far we expect the estimate to be off, on average. For each simulated experiment, we compute the error, $\bar{x} - \mu$, and then compute the **root mean squared error** (RMSE). In this example, it is roughly 2.5 kg.

Confidence interval *(CI)*
: A range that includes a given fraction of the sampling distribution. For example, the 90% confidence interval is the range from the 5th to the 95th percentile. In this example, the 90% CI is (86, 94) kg.

Standard errors and confidence intervals are the source of much confusion:

- People often confuse standard error and standard deviation. Remember that standard deviation describes variability in a measured quantity; in this example, the standard deviation of gorilla weight is 7.5 kg. Standard error describes variability in an estimate. In this example, the standard error of the mean, based on a sample of 9 measurements, is 2.5 kg.

  One way to remember the difference is that, as sample size increases, standard error gets smaller; standard deviation does not.

- People often think that there is a 90% probability that the actual parameter, μ, falls in the 90% confidence interval. Sadly, that is not true. If you want to make a claim like that, you have to use Bayesian methods (see my book, *Think Bayes*).

> The sampling distribution answers a different question: it gives you a sense of how reliable an estimate is by telling you how much it would vary if you ran the experiment again.

It is important to remember that confidence intervals and standard errors only quantify sampling error; that is, error due to measuring only part of the population. The sampling distribution does not account for other sources of error, notably sampling bias and measurement error, which are the topics of the next section.

Sampling Bias

Suppose that instead of the weight of gorillas in a nature preserve, you want to know the average weight of women in the city where you live. It is unlikely that you would be allowed to choose a representative sample of women and weigh them.

A simple alternative would be "telephone sampling;" that is, you could choose random numbers from the phone book, call and ask to speak to an adult woman, and ask how much she weighs.

Telephone sampling has obvious limitations. For example, the sample is limited to people whose telephone numbers are listed, so it eliminates people without phones (who might be poorer than average) and people with unlisted numbers (who might be richer). Also, if you call home telephones during the day, you are less likely to sample people with jobs. And if you only sample the person who answers the phone, you are less likely to sample people who share a phone line.

If factors like income, employment, and household size are related to weight—and it is plausible that they are—the results of your survey would be affected one way or another. This problem is called **sampling bias** because it is a property of the sampling process.

This sampling process is also vulnerable to self-selection, which is a kind of sampling bias. Some people will refuse to answer the question, and if the tendency to refuse is related to weight, that would affect the results.

Finally, if you ask people how much they weigh, rather than weighing them, the results might not be accurate. Even helpful respondents might round up or down if they are uncomfortable with their actual weight. And not all respondents are helpful. These inaccuracies are examples of **measurement error**.

When you report an estimated quantity, it is useful to report standard error, or a confidence interval, or both, in order to quantify sampling error. But it is also important to remember that sampling error is only one source of error, and often it is not the biggest.

Exponential Distributions

Let's play one more round of the estimation game. *I'm thinking of a distribution.* It's an exponential distribution, and here's a sample:

```
[5.384, 4.493, 19.198, 2.790, 6.122, 12.844]
```

What do you think is the parameter, λ, of this distribution?

In general, the mean of an exponential distribution is $1 / \lambda$, so working backwards, we might choose

$$L = 1 / \bar{x}$$

L is an estimator of λ. And not just any estimator; it is also the maximum likelihood estimator. So if you want to maximize your chance of guessing λ exactly, L is the way to go.

But we know that $\bar{x}$ is not robust in the presence of outliers, so we expect L to have the same problem.

We can choose an alternative based on the sample median. The median of an exponential distribution is $\ln(2) / \lambda$, so working backwards again, we can define an estimator

$$L_m = \ln(2) / m$$

where m is the sample median.

To test the performance of these estimators, we can simulate the sampling process:

```
def Estimate3(n=7, m=1000):
    lam = 2

    means = []
    medians = []
    for _ in range(m):
        xs = np.random.exponential(1.0/lam, n)
        L = 1 / np.mean(xs)
        Lm = math.log(2) / thinkstats2.Median(xs)
        means.append(L)
        medians.append(Lm)

    print('rmse L', RMSE(means, lam))
    print('rmse Lm', RMSE(medians, lam))
    print('mean error L', MeanError(means, lam))
    print('mean error Lm', MeanError(medians, lam))
```

When I run this experiment with $\lambda = 2$, the RMSE of L is 1.1. For the median-based estimator L_m, RMSE is 1.8. We can't tell from this experiment whether L minimizes MSE, but at least it seems better than L_m.

Sadly, it seems that both estimators are biased. For L the mean error is 0.33, and for L_m it is 0.45. And neither converges to 0 as `m` increases.

It turns out that $\bar{x}$ is an unbiased estimator of the mean of the distribution, $1 / \lambda$, but L is not an unbiased estimator of λ.

Exercises

For the following exercises, you might want to start with a copy of `estimation.py`. Solutions are in `chap08soln.py`

Exercise 8-1.

In this chapter we used $\bar{x}$ and median to estimate μ, and found that $\bar{x}$ yields lower MSE. Also, we used S^2 and S_{n-1}^2 to estimate σ, and found that S^2 is biased and S_{n-1}^2 is unbiased.

Run similar experiments to see if $\bar{x}$ and median are biased estimates of μ. Also check whether S^2 or S_{n-1}^2 yields a lower MSE.

Exercise 8-2.

Suppose that you draw a sample with size $n = 10$ from an exponential distribution with $\lambda = 2$. Simulate this experiment 1000 times and plot the sampling distribution of the estimate L. Compute the standard error of the estimate and the 90% confidence interval.

Repeat the experiment with a few different values of n and make a plot of standard error versus n.

Exercise 8-3.

In games like hockey and soccer, the time between goals is roughly exponential. So you could estimate a team's goal-scoring rate by observing the number of goals they score in a game. This estimation process is a little different from sampling the time between goals, so let's see how it works.

Write a function that takes a goal-scoring rate, `lam`, in goals per game, and simulates a game by generating the time between goals until the total time exceeds one game, then returns the number of goals scored.

Write another function that simulates many games, stores the estimates of `lam`, then computes their mean error and RMSE.

Is this way of making an estimate biased? Plot the sampling distribution of the estimates and the 90% confidence interval. What is the standard error? What happens to sampling error for increasing values of `lam`?

Glossary

estimation
: The process of inferring the parameters of a distribution from a sample.

estimator
: A statistic used to estimate a parameter.

mean squared error (MSE)
: A measure of estimation error.

root mean squared error (RMSE)
: The square root of MSE, a more meaningful representation of typical error magnitude.

maximum likelihood estimator (MLE)
: An estimator that computes the point estimate most likely to be correct.

bias (of an estimator)
: The tendency of an estimator to be above or below the actual value of the parameter, when averaged over repeated experiments.

sampling error
: Error in an estimate due to the limited size of the sample and variation due to chance.

sampling bias
: Error in an estimate due to a sampling process that is not representative of the population.

measurement error
: Error in an estimate due to inaccuracy collecting or recording data.

sampling distribution
: The distribution of a statistic if an experiment is repeated many times.

standard error
: The RMSE of an estimate, which quantifies variability due to sampling error (but not other sources of error).

confidence interval
: An interval that represents the expected range of an estimator if an experiment is repeated many times.

CHAPTER 9
Hypothesis Testing

The code for this chapter is in `hypothesis.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

Classical Hypothesis Testing

Exploring the data from the NSFG, we saw several "apparent effects," including differences between first babies and others. So far we have taken these effects at face value; in this chapter, we put them to the test.

The fundamental question we want to address is whether the effects we see in a sample are likely to appear in the larger population. For example, in the NSFG sample we see a difference in mean pregnancy length for first babies and others. We would like to know if that effect reflects a real difference for women in the U.S., or if it might appear in the sample by chance.

There are several ways we could formulate this question, including Fisher null hypothesis testing, Neyman-Pearson decision theory, and Bayesian inference.[1] What I present here is a subset of all three that makes up most of what people use in practice, which I will call **classical hypothesis testing**.

The goal of classical hypothesis testing is to answer the question, "Given a sample and an apparent effect, what is the probability of seeing such an effect by chance?" Here's how we answer that question:

- The first step is to quantify the size of the apparent effect by choosing a **test statistic**. In the NSFG example, the apparent effect is a difference in pregnancy length

1. For more about Bayesian inference, see the sequel to this book, *Think Bayes*.

between first babies and others, so a natural choice for the test statistic is the difference in means between the two groups.

- The second step is to define a **null hypothesis**, which is a model of the system based on the assumption that the apparent effect is not real. In the NSFG example the null hypothesis is that there is no difference between first babies and others; that is, that pregnancy lengths for both groups have the same distribution.
- The third step is to compute a **p-value**, which is the probability of seeing the apparent effect if the null hypothesis is true. In the NSFG example, we would compute the actual difference in means, then compute the probability of seeing a difference as big, or bigger, under the null hypothesis.
- The last step is to interpret the result. If the p-value is low, the effect is said to be **statistically significant**, which means that it is unlikely to have occurred by chance. In that case we infer that the effect is more likely to appear in the larger population.

The logic of this process is similar to a proof by contradiction. To prove a mathematical statement, A, you assume temporarily that A is false. If that assumption leads to a contradiction, you conclude that A must actually be true.

Similarly, to test a hypothesis like, "This effect is real," we assume, temporarily, that it is not. That's the null hypothesis. Based on that assumption, we compute the probability of the apparent effect. That's the p-value. If the p-value is low, we conclude that the null hypothesis is unlikely to be true.

HypothesisTest

`thinkstats2` provides `HypothesisTest`, a class that represents the structure of a classical hypothesis test. Here is the definition:

```
class HypothesisTest(object):

    def __init__(self, data):
        self.data = data
        self.MakeModel()
        self.actual = self.TestStatistic(data)

    def PValue(self, iters=1000):
        self.test_stats = [self.TestStatistic(self.RunModel())
                           for _ in range(iters)]

        count = sum(1 for x in self.test_stats if x >= self.actual)
        return count / iters

    def TestStatistic(self, data):
        raise UnimplementedMethodException()
```

```
    def MakeModel(self):
        pass

    def RunModel(self):
        raise UnimplementedMethodException()
```

`HypothesisTest` is an abstract parent class that provides complete definitions for some methods and place-keepers for others. Child classes based on `HypothesisTest` inherit `__init__` and `PValue` and provide `TestStatistic`, `RunModel`, and optionally `MakeModel`.

`__init__` takes the data in whatever form is appropriate. It calls `MakeModel`, which builds a representation of the null hypothesis, then passes the data to `TestStatistic`, which computes the size of the effect in the sample.

`PValue` computes the probability of the apparent effect under the null hypothesis. It takes as a parameter `iters`, which is the number of simulations to run. The first line generates simulated data, computes test statistics, and stores them in `test_stats`. The result is the fraction of elements in `test_stats` that exceed or equal the observed test statistic, `self.actual`.

As a simple example,[2] suppose we toss a coin 250 times and see 140 heads and 110 tails. Based on this result, we might suspect that the coin is biased; that is, more likely to land heads. To test this hypothesis, we compute the probability of seeing such a difference if the coin is actually fair:

```
class CoinTest(thinkstats2.HypothesisTest):

    def TestStatistic(self, data):
        heads, tails = data
        test_stat = abs(heads - tails)
        return test_stat

    def RunModel(self):
        heads, tails = self.data
        n = heads + tails
        sample = [random.choice('HT') for _ in range(n)]
        hist = thinkstats2.Hist(sample)
        data = hist['H'], hist['T']
        return data
```

The parameter, `data`, is a pair of integers: the number of heads and tails. The test statistic is the absolute difference between them, so `self.actual` is 30.

2. Adapted from MacKay, *Information Theory, Inference, and Learning Algorithms*, 2003.

`RunModel` simulates coin tosses assuming that the coin is actually fair. It generates a sample of 250 tosses, uses Hist to count the number of heads and tails, and returns a pair of integers.

Now all we have to do is instantiate `CoinTest` and call `PValue`:

```
ct = CoinTest((140, 110))
pvalue = ct.PValue()
```

The result is about 0.07, which means that if the coin is fair, we expect to see a difference as big as 30 about 7% of the time.

How should we interpret this result? By convention, 5% is the threshold of statistical significance. If the p-value is less than 5%, the effect is considered significant; otherwise it is not.

But the choice of 5% is arbitrary, and (as we will see later) the p-value depends on the choice of the test statistics and the model of the null hypothesis. So p-values should not be considered precise measurements.

I recommend interpreting p-values according to their order of magnitude: if the p-value is less than 1%, the effect is unlikely to be due to chance; if it is greater than 10%, the effect can plausibly be explained by chance. P-values between 1% and 10% should be considered borderline. So in this example I conclude that the data do not provide strong evidence that the coin is biased or not.

Testing a Difference in Means

One of the most common effects to test is a difference in mean between two groups. In the NSFG data, we saw that the mean pregnancy length for first babies is slightly longer, and the mean birth weight is slightly smaller. Now we will see if those effects are statistically significant.

For these examples, the null hypothesis is that the distributions for the two groups are the same. One way to model the null hypothesis is by **permutation**; that is, we can take values for first babies and others and shuffle them, treating the two groups as one big group:

```
class DiffMeansPermute(thinkstats2.HypothesisTest):

    def TestStatistic(self, data):
        group1, group2 = data
        test_stat = abs(group1.mean() - group2.mean())
        return test_stat

    def MakeModel(self):
        group1, group2 = self.data
        self.n, self.m = len(group1), len(group2)
        self.pool = np.hstack((group1, group2))
```

```
def RunModel(self):
    np.random.shuffle(self.pool)
    data = self.pool[:self.n], self.pool[self.n:]
    return data
```

`data` is a pair of sequences, one for each group. The test statistic is the absolute difference in the means.

`MakeModel` records the sizes of the groups, `n` and `m`, and combines the groups into one NumPy array, `self.pool`.

`RunModel` simulates the null hypothesis by shuffling the pooled values and splitting them into two groups with sizes `n` and `m`. As always, the return value from `RunModel` has the same format as the observed data.

To test the difference in pregnancy length, we run:

```
live, firsts, others = first.MakeFrames()
data = firsts.prglngth.values, others.prglngth.values
ht = DiffMeansPermute(data)
pvalue = ht.PValue()
```

`MakeFrames` reads the NSFG data and returns DataFrames representing all live births, first babies, and others. We extract pregnancy lengths as NumPy arrays, pass them as data to `DiffMeansPermute`, and compute the p-value. The result is about 0.17, which means that we expect to see a difference as big as the observed effect about 17% of the time. So this effect is not statistically significant.

`HypothesisTest` provides `PlotCdf`, which plots the distribution of the test statistic and a gray line indicating the observed effect size:

```
ht.PlotCdf()
thinkplot.Show(xlabel='test statistic',
               ylabel='CDF')
```

Figure 9-1 shows the result. The CDF intersects the observed difference at 0.83, which is the complement of the p-value, 0.17.

If we run the same analysis with birth weight, the computed p-value is 0; after 1000 attempts, the simulation never yields an effect as big as the observed difference, 0.12 lbs. So we would report $p < 0.001$, and conclude that the difference in birth weight is statistically significant.

Other Test Statistics

Choosing the best test statistic depends on what question you are trying to address. For example, if the relevant question is whether pregnancy lengths are different for first babies, then it makes sense to test the absolute difference in means, as we did in the previous section.

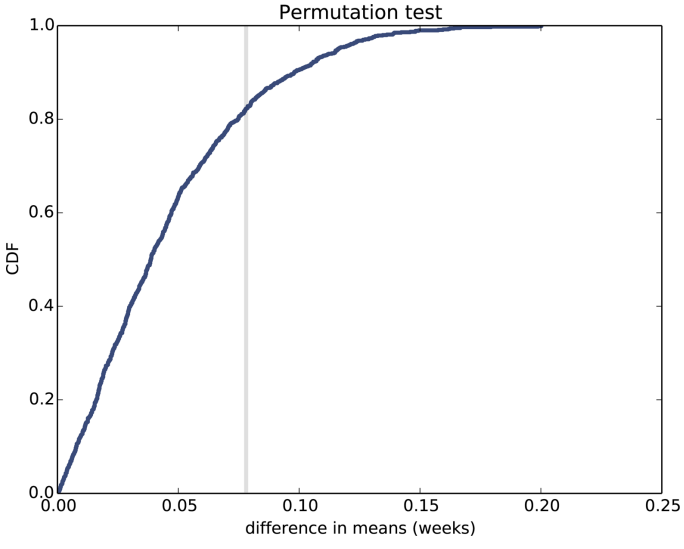


Figure 9-1. CDF of difference in mean pregnancy length under the null hypothesis

If we had some reason to think that first babies are likely to be late, then we would not take the absolute value of the difference; instead, we would use this test statistic:

```
class DiffMeansOneSided(DiffMeansPermute):

    def TestStatistic(self, data):
        group1, group2 = data
        test_stat = group1.mean() - group2.mean()
        return test_stat
```

`DiffMeansOneSided` inherits `MakeModel` and `RunModel` from `DiffMeansPermute`; the only difference is that `TestStatistic` does not take the absolute value of the difference. This kind of test is called **one-sided** because it only counts one side of the distribution of differences. The previous test, using both sides, is **two-sided**.

For this version of the test, the p-value is 0.09. In general, the p-value for a one-sided test is about half the p-value for a two-sided test, depending on the shape of the distribution.

The one-sided hypothesis, that first babies are born late, is more specific than the two-sided hypothesis, so the p-value is smaller. But even for the stronger hypothesis, the difference is not statistically significant.

We can use the same framework to test for a difference in standard deviation. In "Other Visualizations" on page 30, we saw some evidence that first babies are more likely to be

early or late, and less likely to be on time. So we might hypothesize that the standard deviation is higher. Here's how we can test that:

```
class DiffStdPermute(DiffMeansPermute):

    def TestStatistic(self, data):
        group1, group2 = data
        test_stat = group1.std() - group2.std()
        return test_stat
```

This is a one-sided test because the hypothesis is that the standard deviation for first babies is higher, not just different. The p-value is 0.09, which is not statistically significant.

Testing a Correlation

This framework can also test correlations. For example, in the NSFG data set, the correlation between birth weight and mother's age is about 0.07. It seems like older mothers have heavier babies. But could this effect be due to chance?

For the test statistic, I use Pearson's correlation, but Spearman's would work as well. If we had reason to expect positive correlation, we would do a one-sided test. But since we have no such reason, I'll do a two-sided test using the absolute value of correlation.

The null hypothesis is that there is no correlation between mother's age and birth weight. By shuffling the observed values, we can simulate a world where the distributions of age and birth weight are the same, but where the variables are unrelated:

```
class CorrelationPermute(thinkstats2.HypothesisTest):

    def TestStatistic(self, data):
        xs, ys = data
        test_stat = abs(thinkstats2.Corr(xs, ys))
        return test_stat

    def RunModel(self):
        xs, ys = self.data
        xs = np.random.permutation(xs)
        return xs, ys
```

`data` is a pair of sequences. `TestStatistic` computes the absolute value of Pearson's correlation. `RunModel` shuffles the `xs` and returns simulated data.

Here's the code that reads the data and runs the test:

```
    live, firsts, others = first.MakeFrames()
    live = live.dropna(subset=['agepreg', 'totalwgt_lb'])
    data = live.agepreg.values, live.totalwgt_lb.values
    ht = CorrelationPermute(data)
    pvalue = ht.PValue()
```

I use `dropna` with the `subset` argument to drop rows that are missing either of the variables we need.

The actual correlation is 0.07. The computed p-value is 0; after 1000 iterations the largest simulated correlation is 0.04. So although the observed correlation is small, it is statistically significant.

This example is a reminder that "statistically significant" does not always mean that an effect is important, or significant in practice. It only means that it is unlikely to have occurred by chance.

Testing Proportions

Suppose you run a casino and you suspect that a customer is using a crooked die; that is, one that has been modified to make one of the faces more likely than the others. You apprehend the alleged cheater and confiscate the die, but now you have to prove that it is crooked. You roll the die 60 times and get the following results:

Value	1	2	3	4	5	6
Frequency	8	9	19	5	8	11

On average you expect each value to appear 10 times. In this dataset, the value 3 appears more often than expected, and the value 4 appears less often. But are these differences statistically significant?

To test this hypothesis, we can compute the expected frequency for each value, the difference between the expected and observed frequencies, and the total absolute difference. In this example, we expect each side to come up 10 times out of 60; the deviations from this expectation are -2, -1, 9, -5, -2, and 1; so the total absolute difference is 20. How often would we see such a difference by chance?

Here's a version of `HypothesisTest` that answers that question:

```
class DiceTest(thinkstats2.HypothesisTest):

    def TestStatistic(self, data):
        observed = data
        n = sum(observed)
        expected = np.ones(6) * n / 6
        test_stat = sum(abs(observed - expected))
        return test_stat

    def RunModel(self):
        n = sum(self.data)
        values = [1, 2, 3, 4, 5, 6]
        rolls = np.random.choice(values, n, replace=True)
        hist = thinkstats2.Hist(rolls)
```

```
        freqs = hist.Freqs(values)
        return freqs
```

The data are represented as a list of frequencies: the observed values are `[8, 9, 19, 5, 8, 11]`; the expected frequencies are all 10. The test statistic is the sum of the absolute differences.

The null hypothesis is that the die is fair, so we simulate that by drawing random samples from `values`. `RunModel` uses `Hist` to compute and return the list of frequencies.

The p-value for this data is 0.13, which means that if the die is fair we expect to see the observed total deviation, or more, about 13% of the time. So the apparent effect is not statistically significant.

Chi-Squared Tests

In the previous section we used total deviation as the test statistic. But for testing proportions it is more common to use the chi-squared statistic:

$$\chi^2 = \sum_i \frac{(O_i - E_i)^2}{E_i}$$

Where O_i are the observed frequencies and E_i are the expected frequencies. Here's the Python code:

```
class DiceChiTest(DiceTest):

    def TestStatistic(self, data):
        observed = data
        n = sum(observed)
        expected = np.ones(6) * n / 6
        test_stat = sum((observed - expected)**2 / expected)
        return test_stat
```

Squaring the deviations (rather than taking absolute values) gives more weight to large deviations. Dividing through by `expected` standardizes the deviations, although in this case it has no effect because the expected frequencies are all equal.

The p-value using the chi-squared statistic is 0.04, substantially smaller than what we got using total deviation, 0.13. If we take the 5% threshold seriously, we would consider this effect statistically significant. But considering the two tests together, I would say that the results are borderline. I would not rule out the possibility that the die is crooked, but I would not convict the accused cheater.

This example demonstrates an important point: the p-value depends on the choice of test statistic and the model of the null hypothesis, and sometimes these choices determine whether an effect is statistically significant or not.

First Babies Again

Earlier in this chapter we looked at pregnancy lengths for first babies and others, and concluded that the apparent differences in mean and standard deviation are not statistically significant. But in "Other Visualizations" on page 30, we saw several apparent differences in the distribution of pregnancy length, especially in the range from 35 to 43 weeks. To see whether those differences are statistically significant, we can use a test based on a chi-squared statistic.

The code combines elements from previous examples:

```
class PregLengthTest(thinkstats2.HypothesisTest):

    def MakeModel(self):
        firsts, others = self.data
        self.n = len(firsts)
        self.pool = np.hstack((firsts, others))

        pmf = thinkstats2.Pmf(self.pool)
        self.values = range(35, 44)
        self.expected_probs = np.array(pmf.Probs(self.values))

    def RunModel(self):
        np.random.shuffle(self.pool)
        data = self.pool[:self.n], self.pool[self.n:]
        return data
```

The data are represented as two lists of pregnancy lengths. The null hypothesis is that both samples are drawn from the same distribution. `MakeModel` models that distribution by pooling the two samples using `hstack`. Then `RunModel` generates simulated data by shuffling the pooled sample and splitting it into two parts.

`MakeModel` also defines `values`, which is the range of weeks we'll use, and `expected_probs`, which is the probability of each value in the pooled distribution.

Here's the code that computes the test statistic:

```
# class PregLengthTest:

    def TestStatistic(self, data):
        firsts, others = data
        stat = self.ChiSquared(firsts) + self.ChiSquared(others)
        return stat

    def ChiSquared(self, lengths):
        hist = thinkstats2.Hist(lengths)
        observed = np.array(hist.Freqs(self.values))
        expected = self.expected_probs * len(lengths)
        stat = sum((observed - expected)**2 / expected)
        return stat
```

`TestStatistic` computes the chi-squared statistic for first babies and others, and adds them.

`ChiSquared` takes a sequence of pregnancy lengths, computes its histogram, and computes `observed`, which is a list of frequencies corresponding to `self.values`. To compute the list of expected frequencies, it multiplies the precomputed probabilities, `expected_probs`, by the sample size. It returns the chi-squared statistic, `stat`.

For the NSFG data the total chi-squared statistic is 102, which doesn't mean much by itself. But after 1,000 iterations, the largest test statistic generated under the null hypothesis is 32. We conclude that the observed chi-squared statistic is unlikely under the null hypothesis, so the apparent effect is statistically significant.

This example demonstrates a limitation of **chi-squared tests**: they indicate that there is a difference between the two groups, but they don't say anything specific about what the difference is.

Errors

In classical hypothesis testing, an effect is considered statistically significant if the p-value is below some threshold, commonly 5%. This procedure raises two questions:

- If the effect is actually due to chance, what is the probability that we will wrongly consider it significant? This probability is the **false positive rate**.
- If the effect is real, what is the chance that the hypothesis test will fail? This probability is the **false negative rate**.

The false positive rate is relatively easy to compute: if the threshold is 5%, the false positive rate is 5%. Here's why:

- If there is no real effect, the null hypothesis is true, so we can compute the distribution of the test statistic by simulating the null hypothesis. Call this distribution CDF_T.
- Each time we run an experiment, we get a test statistic, t, which is drawn from CDF_T. Then we compute a p-value, which is the probability that a random value from CDF_T exceeds `t`, so that's 1 - $CDF_T(t)$.
- The p-value is less than 5% if $CDF_T(t)$ is greater than 95%; that is, if t exceeds the 95th percentile. And how often does a value chosen from CDF_T exceed the 95th percentile? 5% of the time.

So if you perform one hypothesis test with a 5% threshold, you expect a false positive 1 time in 20.

Power

The false negative rate is harder to compute because it depends on the actual effect size, and normally we don't know that. One option is to compute a rate conditioned on a hypothetical effect size.

For example, if we assume that the observed difference between groups is accurate, we can use the observed samples as a model of the population and run hypothesis tests with simulated data:

```
def FalseNegRate(data, num_runs=100):
    group1, group2 = data
    count = 0

    for i in range(num_runs):
        sample1 = thinkstats2.Resample(group1)
        sample2 = thinkstats2.Resample(group2)

        ht = DiffMeansPermute((sample1, sample2))
        pvalue = ht.PValue(iters=101)
        if pvalue > 0.05:
            count += 1

    return count / num_runs
```

`FalseNegRate` takes data in the form of two sequences, one for each group. Each time through the loop, it simulates an experiment by drawing a random sample from each group and running a hypothesis test. Then it checks the result and counts the number of false negatives.

`Resample` takes a sequence and draws a sample with the same length, with replacement:

```
def Resample(xs):
    return np.random.choice(xs, len(xs), replace=True)
```

Here's the code that tests pregnancy lengths:

```
    live, firsts, others = first.MakeFrames()
    data = firsts.prglngth.values, others.prglngth.values
    neg_rate = FalseNegRate(data)
```

The result is about 70%, which means that if the actual difference in mean pregnancy length is 0.78 weeks, we expect an experiment with this sample size to yield a negative test 70% of the time.

This result is often presented the other way around: if the actual difference is 0.78 weeks, we should expect a positive test only 30% of the time. This "correct positive rate" is called the **power** of the test, or sometimes "sensitivity." It reflects the ability of the test to detect an effect of a given size.

In this example, the test had only a 30% chance of yielding a positive result (again, assuming that the difference is 0.78 weeks). As a rule of thumb, a power of 80% is considered acceptable, so we would say that this test was "underpowered."

In general a negative hypothesis test does not imply that there is no difference between the groups; instead it suggests that if there is a difference, it is too small to detect with this sample size.

Replication

The hypothesis testing process I demonstrated in this chapter is not, strictly speaking, good practice.

First, I performed multiple tests. If you run one hypothesis test, the chance of a false positive is about 1 in 20, which might be acceptable. But if you run 20 tests, you should expect at least one false positive, most of the time.

Second, I used the same dataset for exploration and testing. If you explore a large dataset, find a surprising effect, and then test whether it is significant, you have a good chance of generating a false positive.

To compensate for multiple tests, you can adjust the p-value threshold (see this Wikipedia page). Or you can address both problems by partitioning the data, using one set for exploration and the other for testing.

In some fields these practices are required or at least encouraged. But it is also common to address these problems implicitly by replicating published results. Typically the first paper to report a new result is considered exploratory. Subsequent papers that replicate the result with new data are considered confirmatory.

As it happens, we have an opportunity to replicate the results in this chapter. The first edition of this book is based on Cycle 6 of the NSFG, which was released in 2002. In October 2011, the CDC released additional data based on interviews conducted from 2006–2010. `nsfg2.py` contains code to read and clean this data. In the new dataset:

- The difference in mean pregnancy length is 0.16 weeks and statistically significant with $p < 0.001$ (compared to 0.078 weeks in the original dataset).
- The difference in birth weight is 0.17 pounds with $p < 0.001$ (compared to 0.12 lbs in the original dataset).
- The correlation between birth weight and mother's age is 0.08 with $p < 0.001$ (compared to 0.07).
- The chi-squared test is statistically significant with $p < 0.001$ (as it was in the original).

In summary, all of the effects that were statistically significant in the original dataset were replicated in the new dataset, and the difference in pregnancy length, which was not significant in the original, is bigger in the new dataset and significant.

Exercises

A solution to these exercises is in `chap09soln.py`.

Exercise 9-1.

As sample size increases, the power of a hypothesis test increases, which means it is more likely to be positive if the effect is real. Conversely, as sample size decreases, the test is less likely to be positive even if the effect is real.

To investigate this behavior, run the tests in this chapter with different subsets of the NSFG data. You can use `thinkstats2.SampleRows` to select a random subset of the rows in a DataFrame.

What happens to the p-values of these tests as the sample size decreases? What is the smallest sample size that yields a positive test?

Exercise 9-2.

In "Testing a Difference in Means" on page 104, we simulated the null hypothesis by permutation; that is, we treated the observed values as if they represented the entire population, and randomly assigned the members of the population to the two groups.

An alternative is to use the sample to estimate the distribution for the population, then draw a random sample from that distribution. This process is called **resampling**. There are several ways to implement resampling, but one of the simplest is to draw a sample with replacement from the observed values, as in "Power" on page 112.

Write a class named `DiffMeansResample` that inherits from `DiffMeansPermute` and overrides `RunModel` to implement resampling, rather than permutation.

Use this model to test the differences in pregnancy length and birth weight. How much does the model affect the results?

Glossary

hypothesis testing
: The process of determining whether an apparent effect is statistically significant.

test statistic
: A statistic used to quantify an effect size.

null hypothesis
: A model of a system based on the assumption that an apparent effect is due to chance.

p-value
: The probability that an effect could occur by chance.

statistically significant
: An effect is statistically significant if it is unlikely to occur by chance.

permutation test
: A way to compute p-values by generating permutations of an observed dataset.

resampling test
: A way to compute p-values by generating samples, with replacement, from an observed dataset.

two-sided test
: A test that asks, "What is the chance of an effect as big as the observed effect, positive or negative?"

one-sided test
: A test that asks, "What is the chance of an effect as big as the observed effect, and with the same sign?"

chi-squared test
: A test that uses the chi-squared statistic as the test statistic.

false positive
: The conclusion that an effect is real when it is not.

false negative
: The conclusion that an effect is due to chance when it is not.

power
: The probability of a positive test if the null hypothesis is false.

CHAPTER 10
Linear Least Squares

The code for this chapter is in `linear.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

Least Squares Fit

Correlation coefficients measure the strength and sign of a relationship, but not the slope. There are several ways to estimate the slope; the most common is a **linear least squares fit**. A "linear fit" is a line intended to model the relationship between variables. A "least squares" fit is one that minimizes the mean squared error (MSE) between the line and the data.

Suppose we have a sequence of points, `ys`, that we want to express as a function of another sequence `xs`. If there is a linear relationship between `xs` and `ys` with intercept `inter` and slope `slope`, we expect each `y[i]` to be `inter + slope * x[i]`.

But unless the correlation is perfect, this prediction is only approximate. The vertical deviation from the line, or **residual**, is

```
res = ys - (inter + slope * xs)
```

The residuals might be due to random factors like measurement error, or nonrandom factors that are unknown. For example, if we are trying to predict weight as a function of height, unknown factors might include diet, exercise, and body type.

If we get the parameters `inter` and `slope` wrong, the residuals get bigger, so it makes intuitive sense that the parameters we want are the ones that minimize the residuals.

We might try to minimize the absolute value of the residuals, or their squares, or their cubes; but the most common choice is to minimize the sum of squared residuals, `sum(res**2)`.

Why? There are three good reasons and one less important one:

- Squaring has the feature of treating positive and negative residuals the same, which is usually what we want.
- Squaring gives more weight to large residuals, but not so much weight that the largest residual always dominates.
- If the residuals are uncorrelated and normally distributed with mean 0 and constant (but unknown) variance, then the least squares fit is also the maximum likelihood estimator of `inter` and `slope`. See Wikipedia.
- The values of `inter` and `slope` that minimize the squared residuals can be computed efficiently.

The last reason made sense when computational efficiency was more important than choosing the method most appropriate to the problem at hand. That's no longer the case, so it is worth considering whether squared residuals are the right thing to minimize.

For example, if you are using `xs` to predict values of `ys`, guessing too high might be better (or worse) than guessing too low. In that case you might want to compute some cost function for each residual, and minimize total cost, `sum(cost(res))`. However, computing a least squares fit is quick, easy and often good enough.

Implementation

`thinkstats2` provides simple functions that demonstrate linear least squares:

```
def LeastSquares(xs, ys):
    meanx, varx = MeanVar(xs)
    meany = Mean(ys)

    slope = Cov(xs, ys, meanx, meany) / varx
    inter = meany - slope * meanx

    return inter, slope
```

`LeastSquares` takes sequences `xs` and `ys` and returns the estimated parameters `inter` and `slope`. For details on how it works, see Wikipedia.

`thinkstats2` also provides `FitLine`, which takes `inter` and `slope` and returns the fitted line for a sequence of `xs`.

```
def FitLine(xs, inter, slope):
    fit_xs = np.sort(xs)
    fit_ys = inter + slope * fit_xs
    return fit_xs, fit_ys
```

We can use these functions to compute the least squares fit for birth weight as a function of mother's age.

```
live, firsts, others = first.MakeFrames()
live = live.dropna(subset=['agepreg', 'totalwgt_lb'])
ages = live.agepreg
weights = live.totalwgt_lb

inter, slope = thinkstats2.LeastSquares(ages, weights)
fit_xs, fit_ys = thinkstats2.FitLine(ages, inter, slope)
```

The estimated intercept and slope are 6.8 lbs and 0.017 lbs per year. These values are hard to interpret in this form: the intercept is the expected weight of a baby whose mother is 0 years old, which doesn't make sense in context, and the slope is too small to grasp easily.

Instead of presenting the intercept at $x = 0$, it is often helpful to present the intercept at the mean of x. In this case the mean age is about 25 years and the mean baby weight for a 25-year-old mother is 7.3 pounds. The slope is 0.27 ounces per year, or 0.17 pounds per decade.

Figure 10-1 shows a scatter plot of birth weight and age along with the fitted line. It's a good idea to look at a figure like this to assess whether the relationship is linear and whether the fitted line seems like a good model of the relationship.

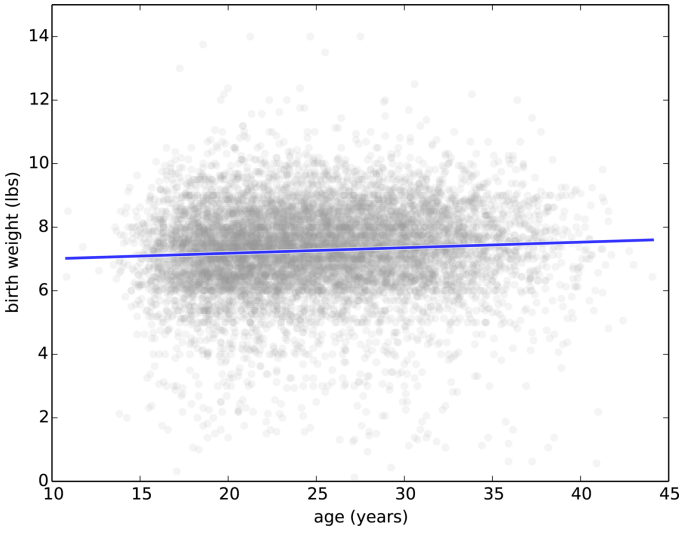


Figure 10-1. Scatter plot of birth weight and mother's age with a linear fit

Residuals

Another useful test is to plot the residuals. `thinkstats2` provides a function that computes residuals:

```
def Residuals(xs, ys, inter, slope):
    xs = np.asarray(xs)
    ys = np.asarray(ys)
    res = ys - (inter + slope * xs)
    return res
```

`Residuals` takes sequences `xs` and `ys` and estimated parameters `inter` and `slope`. It returns the differences between the actual values and the fitted line.

To visualize the residuals, I group respondents by age and compute percentiles in each group, as we saw in "Characterizing Relationships" on page 82. Figure 10-2 shows the 25th, 50th and 75th percentiles of the residuals for each age group. The median is near zero, as expected, and the interquartile range is about 2 pounds. So if we know the mother's age, we can guess the baby's weight within a pound, about 50% of the time.

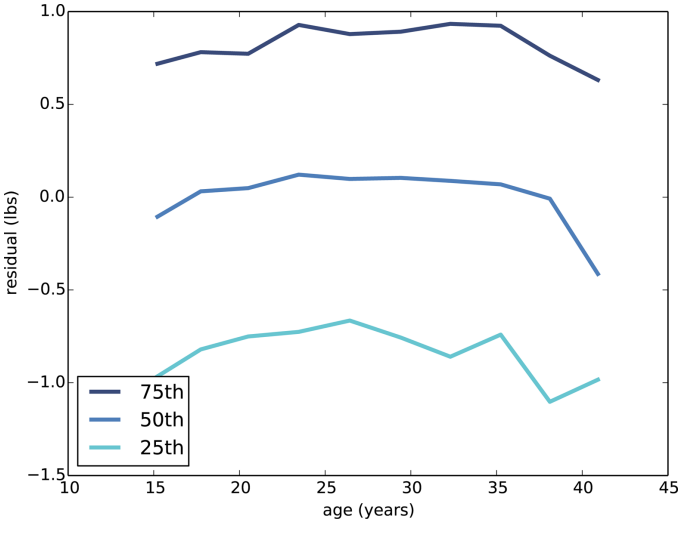


Figure 10-2. Residuals of the linear fit

Ideally these lines should be flat, indicating that the residuals are random, and parallel, indicating that the variance of the residuals is the same for all age groups. In fact, the lines are close to parallel, so that's good; but they have some curvature, indicating that the relationship is non-linear. Nevertheless, the linear fit is a simple model that is probably good enough for some purposes.

Estimation

The parameters `slope` and `inter` are estimates based on a sample; like other estimates, they are vulnerable to sampling bias, measurement error, and sampling error. As dis-

cussed in Chapter 8, sampling bias is caused by nonrepresentative sampling, measurement error is caused by errors in collecting and recording data, and sampling error is the result of measuring a sample rather than the entire population.

To assess sampling error, we ask, "If we run this experiment again, how much variability do we expect in the estimates?" We can answer this question by running simulated experiments and computing sampling distributions of the estimates.

I simulate the experiments by resampling the data; that is, I treat the observed pregnancies as if they were the entire population and draw samples, with replacement, from the observed sample.

```
def SamplingDistributions(live, iters=101):
    t = []
    for _ in range(iters):
        sample = thinkstats2.ResampleRows(live)
        ages = sample.agepreg
        weights = sample.totalwgt_lb
        estimates = thinkstats2.LeastSquares(ages, weights)
        t.append(estimates)

    inters, slopes = zip(*t)
    return inters, slopes
```

`SamplingDistributions` takes a DataFrame with one row per live birth, and `iters`, the number of experiments to simulate. It uses `ResampleRows` to resample the observed pregnancies. We've already seen `SampleRows`, which chooses random rows from a DataFrame. `thinkstats2` also provides `ResampleRows`, which returns a sample the same size as the original:

```
def ResampleRows(df):
    return SampleRows(df, len(df), replace=True)
```

After resampling, we use the simulated sample to estimate parameters. The result is two sequences: the estimated intercepts and estimated slopes.

I summarize the sampling distributions by printing the standard error and confidence interval:

```
def Summarize(estimates, actual=None):
    mean = thinkstats2.Mean(estimates)
    stderr = thinkstats2.Std(estimates, mu=actual)
    cdf = thinkstats2.Cdf(estimates)
    ci = cdf.ConfidenceInterval(90)
    print('mean, SE, CI', mean, stderr, ci)
```

`Summarize` takes a sequence of estimates and the actual value. It prints the mean of the estimates, the standard error and a 90% confidence interval.

For the intercept, the mean estimate is 6.83, with standard error 0.07 and 90% confidence interval (6.71, 6.94). The estimated slope, in more compact form, is 0.0174, SE 0.0028,

CI (0.0126, 0.0220). There is almost a factor of two between the low and high ends of this CI, so it should be considered a rough estimate.

To visualize the sampling error of the estimate, we could plot all of the fitted lines, or for a less cluttered representation, plot a 90% confidence interval for each age. Here's the code:

```
def PlotConfidenceIntervals(xs, inters, slopes,
                            percent=90, **options):
    fys_seq = []
    for inter, slope in zip(inters, slopes):
        fxs, fys = thinkstats2.FitLine(xs, inter, slope)
        fys_seq.append(fys)

    p = (100 - percent) / 2
    percents = p, 100 - p
    low, high = thinkstats2.PercentileRows(fys_seq, percents)
    thinkplot.FillBetween(fxs, low, high, **options)
```

`xs` is the sequence of mother's age. `inters` and `slopes` are the estimated parameters generated by `SamplingDistributions`. `percent` indicates which confidence interval to plot.

`PlotConfidenceIntervals` generates a fitted line for each pair of `inter` and `slope` and stores the results in a sequence, `fys_seq`. Then it uses `PercentileRows` to select the upper and lower percentiles of `y` for each value of `x`. For a 90% confidence interval, it selects the 5th and 95th percentiles. `FillBetween` draws a polygon that fills the space between two lines.

Figure 10-3 shows the 50% and 90% confidence intervals for curves fitted to birth weight as a function of mother's age. The vertical width of the region represents the effect of sampling error; the effect is smaller for values near the mean and larger for the extremes.

Goodness of Fit

There are several ways to measure the quality of a linear model, or **goodness of fit**. One of the simplest is the standard deviation of the residuals.

If you use a linear model to make predictions, `Std(res)` is the root mean squared error (RMSE) of your predictions. For example, if you use mother's age to guess birth weight, the RMSE of your guess would be 1.40 lbs.

If you guess birth weight without knowing the mother's age, the RMSE of your guess is `Std(ys)`, which is 1.41 lbs. So in this example, knowing a mother's age does not improve the predictions substantially.

Another way to measure goodness of fit is the **coefficient of determination**, usually denoted R^2 and called "R-squared":

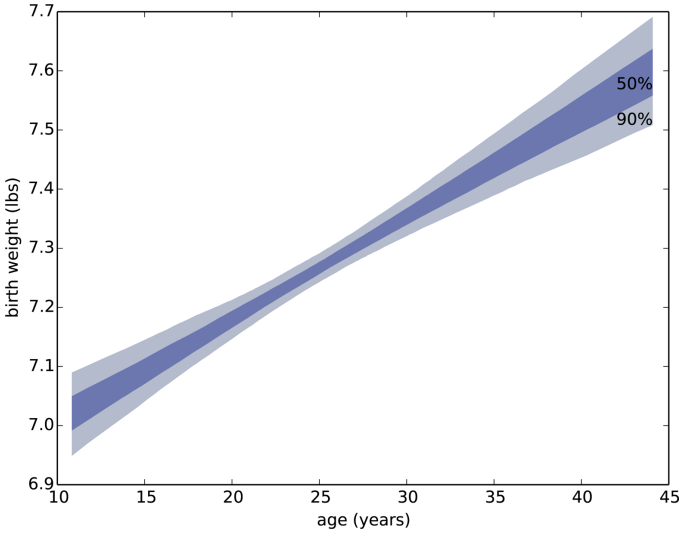


Figure 10-3. 50% and 90% confidence intervals showing variability in the fitted line due to sampling error of `inter` and `slope`

```
def CoefDetermination(ys, res):
    return 1 - Var(res) / Var(ys)
```

`Var(res)` is the MSE of your guesses using the model, `Var(ys)` is the MSE without it. So their ratio is the fraction of MSE that remains if you use the model, and R^2 is the fraction of MSE the model eliminates.

For birth weight and mother's age, R^2 is 0.0047, which means that mother's age predicts about half of 1% of variance in birth weight.

There is a simple relationship between the coefficient of determination and Pearson's coefficient of correlation: $R^2 = \rho^2$. For example, if ρ is 0.8 or -0.8, $R^2 = 0.64$.

Although ρ and R^2 are often used to quantify the strength of a relationship, they are not easy to interpret in terms of predictive power. In my opinion, `Std(res)` is the best representation of the quality of prediction, especially if it is presented in relation to `Std(ys)`.

For example, when people talk about the validity of the SAT (a standardized test used for college admission in the U.S.) they often talk about correlations between SAT scores and other measures of intelligence.

According to one study, there is a Pearson correlation of $\rho = 0.72$ between total SAT scores and IQ scores, which sounds like a strong correlation. But $R^2 = \rho^2 = 0.52$, so SAT scores account for only 52% of variance in IQ.

IQ scores are normalized with `Std(ys) = 15`, so

```
>>> var_ys = 15**2
>>> rho = 0.72
>>> r2 = rho**2
>>> var_res = (1 - r2) * var_ys
>>> std_res = math.sqrt(var_res)
10.4096
```

So using SAT score to predict IQ reduces RMSE from 15 points to 10.4 points. A correlation of 0.72 yields a reduction in RMSE of only 31%.

If you see a correlation that looks impressive, remember that R^2 is a better indicator of reduction in MSE, and reduction in RMSE is a better indicator of predictive power.

Testing a Linear Model

The effect of mother's age on birth weight is small, and has little predictive power. So is it possible that the apparent relationship is due to chance? There are several ways we might test the results of a linear fit.

One option is to test whether the apparent reduction in MSE is due to chance. In that case, the test statistic is R^2 and the null hypothesis is that there is no relationship between the variables. We can simulate the null hypothesis by permutation, as in "Testing a Correlation" on page 107, when we tested the correlation between mother's age and birth weight. In fact, because $R^2 = \rho^2$, a one-sided test of R^2 is equivalent to a two-sided test of ρ. We've already done that test, and found $p < 0.001$, so we conclude that the apparent relationship between mother's age and birth weight is statistically significant.

Another approach is to test whether the apparent slope is due to chance. The null hypothesis is that the slope is actually zero; in that case we can model the birth weights as random variations around their mean. Here's a HypothesisTest for this model:

```
class SlopeTest(thinkstats2.HypothesisTest):

    def TestStatistic(self, data):
        ages, weights = data
        _, slope = thinkstats2.LeastSquares(ages, weights)
        return slope

    def MakeModel(self):
        _, weights = self.data
        self.ybar = weights.mean()
        self.res = weights - self.ybar

    def RunModel(self):
        ages, _ = self.data
        weights = self.ybar + np.random.permutation(self.res)
        return ages, weights
```

The data are represented as sequences of ages and weights. The test statistic is the slope estimated by `LeastSquares`. The model of the null hypothesis is represented by the mean weight of all babies and the deviations from the mean. To generate simulated data, we permute the deviations and add them to the mean.

Here's the code that runs the hypothesis test:

```
live, firsts, others = first.MakeFrames()
live = live.dropna(subset=['agepreg', 'totalwgt_lb'])
ht = SlopeTest((live.agepreg, live.totalwgt_lb))
pvalue = ht.PValue()
```

The p-value is less than 0.001, so although the estimated slope is small, it is unlikely to be due to chance.

Estimating the p-value by simulating the null hypothesis is strictly correct, but there is a simpler alternative. Remember that we already computed the sampling distribution of the slope, in "Estimation" on page 120. To do that, we assumed that the observed slope was correct and simulated experiments by resampling.

Figure 10-4 shows the sampling distribution of the slope, from "Estimation" on page 120, and the distribution of slopes generated under the null hypothesis. The sampling distribution is centered about the estimated slope, 0.017 lbs/year, and the slopes under the null hypothesis are centered around 0; but other than that, the distributions are identical. The distributions are also symmetric, for reasons we will see in "Central Limit Theorem" on page 186.

So we could estimate the p-value two ways:

- Compute the probability that the slope under the null hypothesis exceeds the observed slope.
- Compute the probability that the slope in the sampling distribution falls below 0. (If the estimated slope were negative, we would compute the probability that the slope in the sampling distribution exceeds 0.)

The second option is easier because we normally want to compute the sampling distribution of the parameters anyway. And it is a good approximation unless the sample size is small *and* the distribution of residuals is skewed. Even then, it is usually good enough, because p-values don't have to be precise.

Here's the code that estimates the p-value of the slope using the sampling distribution:

```
inters, slopes = SamplingDistributions(live, iters=1001)
slope_cdf = thinkstats2.Cdf(slopes)
pvalue = slope_cdf[0]
```

Again, we find $p < 0.001$.

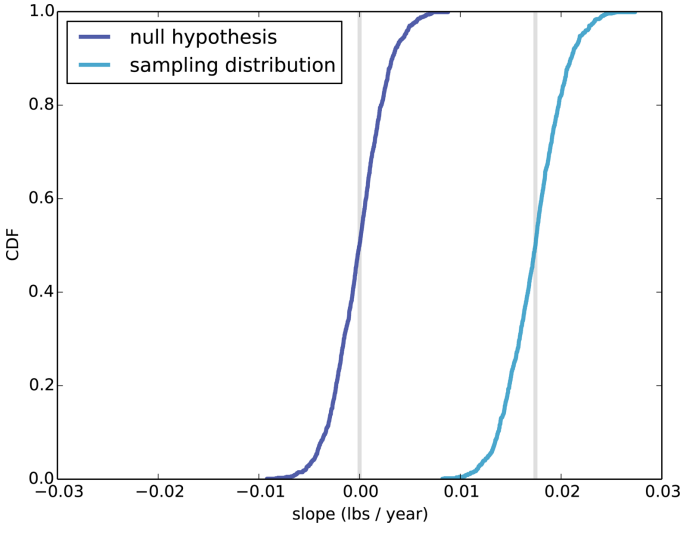


Figure 10-4. The sampling distribution of the estimated slope and the distribution of slopes generated under the null hypothesis. The vertical lines are at 0 and the observed slope, 0.017 lbs/year.

Weighted Resampling

So far we have treated the NSFG data as if it were a representative sample, but as I mentioned in "The National Survey of Family Growth" on page 2, it is not. The survey deliberately oversamples several groups in order to improve the chance of getting statistically significant results; that is, in order to improve the power of tests involving these groups.

This survey design is useful for many purposes, but it means that we cannot use the sample to estimate values for the general population without accounting for the sampling process.

For each respondent, the NSFG data includes a variable called `finalwgt`, which is the number of people in the general population the respondent represents. This value is called a **sampling weight**, or just "weight."

As an example, if you survey 100,000 people in a country of 300 million, each respondent represents 3,000 people. If you oversample one group by a factor of 2, each person in the oversampled group would have a lower weight, about 1500.

To correct for oversampling, we can use resampling; that is, we can draw samples from the survey using probabilities proportional to sampling weights. Then, for any quantity we want to estimate, we can generate sampling distributions, standard errors, and con-

fidence intervals. As an example, I will estimate mean birth weight with and without sampling weights.

In "Estimation" on page 120, we saw `ResampleRows`, which chooses rows from a DataFrame, giving each row the same probability. Now we need to do the same thing using probabilities proportional to sampling weights. `ResampleRowsWeighted` takes a DataFrame, resamples rows according to the weights in `finalwgt`, and returns a DataFrame containing the resampled rows:

```
def ResampleRowsWeighted(df):
    weights = df.finalwgt
    pmf = thinkstats2.Pmf(weights.iteritems())
    cdf = pmf.MakeCdf()
    indices = cdf.Sample(len(weights))
    sample = df.loc[indices]
    return sample
```

`pmf` maps from each row index to its normalized weight. Converting to a Cdf makes the sampling process faster. `indices` is a sequence of row indices; `sample` is a DataFrame that contains the selected rows. Since we sample with replacement, the same row might appear more than once.

Now we can compare the effect of resampling with and without weights. Without weights, we generate the sampling distribution like this:

```
estimates = [ResampleRows(live).totalwgt_lb.mean()
             for _ in range(iters)]
```

With weights, it looks like this:

```
estimates = [ResampleRowsWeighted(live).totalwgt_lb.mean()
             for _ in range(iters)]
```

The following table summarizes the results:

	Mean birth weight (lbs)	Standard error	90% CI
Unweighted	7.27	0.014	(7.24, 7.29)
Weighted	7.35	0.014	(7.32, 7.37)

In this example, the effect of weighting is small but non-negligible. The difference in estimated means, with and without weighting, is about 0.08 pounds, or 1.3 ounces. This difference is substantially larger than the standard error of the estimate, 0.014 pounds, which implies that the difference is not due to chance.

Exercises

A solution to this exercise is in `chap10soln.ipynb`.

Exercise 10-1.

Using the data from the BRFSS, compute the linear least squares fit for log(weight) versus height. How would you best present the estimated parameters for a model like this where one of the variables is log-transformed? If you were trying to guess someone's weight, how much would it help to know their height?

Like the NSFG, the BRFSS oversamples some groups and provides a sampling weight for each respondent. In the BRFSS data, the variable name for these weights is `totalwt`. Use resampling, with and without weights, to estimate the mean height of respondents in the BRFSS, the standard error of the mean, and a 90% confidence interval. How much does correct weighting affect the estimates?

Glossary

linear fit
: A line intended to model the relationship between variables.

least squares fit
: A model of a dataset that minimizes the sum of squares of the residuals.

residual
: The deviation of an actual value from a model.

goodness of fit
: A measure of how well a model fits data.

coefficient of determination
: A statistic intended to quantify goodness of fit.

sampling weight
: A value associated with an observation in a sample that indicates what part of the population it represents.

CHAPTER 11
Regression

The linear least squares fit in the previous chapter is an example of **regression**, which is the more general problem of fitting any kind of model to any kind of data. This use of the term "regression" is a historical accident; it is only indirectly related to the original meaning of the word.

The goal of regression analysis is to describe the relationship between one set of variables, called the **dependent variables**, and another set of variables, called independent or **explanatory variables**.

In the previous chapter we used mother's age as an explanatory variable to predict birth weight as a dependent variable. When there is only one dependent and one explanatory variable, that's **simple regression**. In this chapter, we move on to **multiple regression**, with more than one explanatory variable. If there is more than one dependent variable, that's multivariate regression.

If the relationship between the dependent and explanatory variable is linear, that's **linear regression**. For example, if the dependent variable is y and the explanatory variables are x_1 and x_2, we would write the following linear regression model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

where β_0 is the intercept, β_1 is the parameter associated with x_1, β_2 is the parameter associated with x_2, and ε is the residual due to random variation or other unknown factors.

Given a sequence of values for y and sequences for x_1 and x_2, we can find the parameters, β_0, β_1, and β_2, that minimize the sum of ε^2. This process is called **ordinary least squares**. The computation is similar to that of `thinkstats2.LeastSquare`, but generalized to deal with more than one explanatory variable.

The code for this chapter is in `regression.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

StatsModels

In the previous chapter I presented `thinkstats2.LeastSquares`, an implementation of simple linear regression intended to be easy to read. For multiple regression we'll switch to StatsModels, a Python package that provides several forms of regression and other analyses. If you are using Anaconda, you already have StatsModels; otherwise you might have to install it.

As an example, I'll run the model from the previous chapter with StatsModels:

```
import statsmodels.formula.api as smf

live, firsts, others = first.MakeFrames()
formula = 'totalwgt_lb ~ agepreg'
model = smf.ols(formula, data=live)
results = model.fit()
```

`statsmodels` provides two interfaces (APIs); the "formula" API uses strings to identify the dependent and explanatory variables. It uses a syntax called `patsy`; in this example, the `~` operator separates the dependent variable on the left from the explanatory variables on the right.

`smf.ols` takes the formula string and the DataFrame, `live`, and returns an OLS object that represents the model. The name `ols` stands for "ordinary least squares."

The `fit` method fits the model to the data and returns a RegressionResults object that contains the results.

The results are also available as attributes. `params` is a Series that maps from variable names to their parameters, so we can get the intercept and slope like this:

```
inter = results.params['Intercept']
slope = results.params['agepreg']
```

The estimated parameters are 6.83 and 0.0175, the same as from `LeastSquares`.

`pvalues` is a Series that maps from variable names to the associated p-values, so we can check whether the estimated slope is statistically significant:

```
slope_pvalue = results.pvalues['agepreg']
```

The p-value associated with `agepreg` is `5.7e-11`, which is less than 0.001, as expected.

`results.rsquared` contains R^2, which is 0.0047. `results` also provides `f_pvalue`, which is the p-value associated with the model as a whole, similar to testing whether R^2 is statistically significant.

And `results` provides `resid`, a sequence of residuals, and `fittedvalues`, a sequence of fitted values corresponding to `agepreg`.

The results object provides `summary()`, which represents the results in a readable format.

```
print(results.summary())
```

But it prints a lot of information that is not relevant (yet), so I use a simpler function called `SummarizeResults`. Here are the results of this model:

```
Intercept       6.83    (0)
agepreg         0.0175  (5.72e-11)
R^2 0.004738
Std(ys) 1.408
Std(res) 1.405
```

`Std(ys)` is the standard deviation of the dependent variable, which is the RMSE if you have to guess birth weights without the benefit of any explanatory variables. `Std(res)` is the standard deviation of the residuals, which is the RMSE if your guesses are informed by the mother's age. As we have already seen, knowing the mother's age provides no substantial improvement to the predictions.

Multiple Regression

In "Comparing CDFs" on page 44, we saw that first babies tend to be lighter than others, and this effect is statistically significant. But it is a strange result because there is no obvious mechanism that would cause first babies to be lighter. So we might wonder whether this relationship is **spurious**.

In fact, there is a possible explanation for this effect. We have seen that birth weight depends on mother's age, and we might expect that mothers of first babies are younger than others.

With a few calculations we can check whether this explanation is plausible. Then we'll use multiple regression to investigate more carefully. First, let's see how big the difference in weight is:

```
diff_weight = firsts.totalwgt_lb.mean() - others.totalwgt_lb.mean()
```

First babies are 0.125 lbs lighter, or 2 ounces. And the difference in ages:

```
diff_age = firsts.agepreg.mean() - others.agepreg.mean()
```

The mothers of first babies are 3.59 years younger. Running the linear model again, we get the change in birth weight as a function of age:

```
results = smf.ols('totalwgt_lb ~ agepreg', data=live).fit()
slope = results.params['agepreg']
```

The slope is 0.175 pounds per year. If we multiply the slope by the difference in ages, we get the expected difference in birth weight for first babies and others, due to mother's age:

```
slope * diff_age
```

The result is 0.063, just about half of the observed difference. So we conclude, tentatively, that the observed difference in birth weight can be partly explained by the difference in mother's age.

Using multiple regression, we can explore these relationships more systematically.

```
live['isfirst'] = live.birthord == 1
formula = 'totalwgt_lb ~ isfirst'
results = smf.ols(formula, data=live).fit()
```

The first line creates a new column named `isfirst` that is True for first babies and false otherwise. Then we fit a model using `isfirst` as an explanatory variable.

Here are the results:

```
Intercept         7.33   (0)
isfirst[T.True]  -0.125  (2.55e-05)
R^2 0.00196
```

Because `isfirst` is a boolean, `ols` treats it as a **categorical variable**, which means that the values fall into categories, like True and False, and should not be treated as numbers. The estimated parameter is the effect on birth weight when `isfirst` is true, so the result, -0.125 lbs, is the difference in birth weight between first babies and others.

The slope and the intercept are statistically significant, which means that they were unlikely to occur by chance, but the the R^2 value for this model is small, which means that `isfirst` doesn't account for a substantial part of the variation in birth weight.

The results are similar with `agepreg`:

```
Intercept       6.83    (0)
agepreg         0.0175  (5.72e-11)
R^2 0.004738
```

Again, the parameters are statistically significant, but R^2 is low.

These models confirm results we have already seen. But now we can fit a single model that includes both variables. With the formula `totalwgt_lb ~ isfirst + agepreg`, we get:

```
Intercept        6.91    (0)
isfirst[T.True] -0.0698  (0.0253)
agepreg          0.0154  (3.93e-08)
R^2 0.005289
```

In the combined model, the parameter for `isfirst` is smaller by about half, which means that part of the apparent effect of `isfirst` is actually accounted for by `agepreg`. And the p-value for `isfirst` is about 2.5%, which is on the border of statistical significance.

R^2 for this model is a little higher, which indicates that the two variables together account for more variation in birth weight than either alone (but not by much).

Nonlinear Relationships

Remembering that the contribution of `agepreg` might be non-linear, we might consider adding a variable to capture more of this relationship. One option is to create a column, `agepreg2`, that contains the squares of the ages:

```
live['agepreg2'] = live.agepreg**2
formula = 'totalwgt_lb ~ isfirst + agepreg + agepreg2'
```

Now by estimating parameters for `agepreg` and `agepreg2`, we are effectively fitting a parabola:

```
Intercept        5.69     (1.38e-86)
isfirst[T.True] -0.0504   (0.109)
agepreg          0.112    (3.23e-07)
agepreg2        -0.00185  (8.8e-06)
R^2 0.007462
```

The parameter of `agepreg2` is negative, so the parabola curves downward, which is consistent with the shape of the lines in Figure 10-2.

The quadratic model of `agepreg` accounts for more of the variability in birth weight; the parameter for `isfirst` is smaller in this model, and no longer statistically significant.

Using computed variables like `agepreg2` is a common way to fit polynomials and other functions to data. This process is still considered linear regression, because the dependent variable is a linear function of the explanatory variables, regardless of whether some variables are non-linear functions of others.

The following table summarizes the results of these regressions:

	isfirst	agepreg	agepreg2	R^2
Model 1	-0.125 *	–	–	0.002
Model 2	–	0.0175 *	–	0.0047
Model 3	-0.0698 (0.025)	0.0154 *	–	0.0053
Model 4	-0.0504 (0.11)	0.112 *	-0.00185 *	0.0075

The columns in this table are the explanatory variables and the coefficient of determination, R^2. Each entry is an estimated parameter and either a p-value in parentheses or an asterisk to indicate a p-value less that 0.001.

We conclude that the apparent difference in birth weight is explained, at least in part, by the difference in mother's age. When we include mother's age in the model, the effect of `isfirst` gets smaller, and the remaining effect might be due to chance.

In this example, mother's age acts as a **control variable**; including `agepreg` in the model "controls for" the difference in age between first-time mothers and others, making it possible to isolate the effect (if any) of `isfirst`.

Data Mining

So far we have used regression models for explanation; for example, in the previous section we discovered that an apparent difference in birth weight is actually due to a difference in mother's age. But the R^2 values of those models is very low, which means that they have little predictive power. In this section we'll try to do better.

Suppose one of your co-workers is expecting a baby and there is an office pool to guess the baby's birth weight (if you are not familiar with betting pools, see Wikipedia).

Now suppose that you *really* want to win the pool. What could you do to improve your chances? Well, the NSFG dataset includes 244 variables about each pregnancy and another 3087 variables about each respondent. Maybe some of those variables have predictive power. To find out which ones are most useful, why not try them all?

Testing the variables in the pregnancy table is easy, but in order to use the variables in the respondent table, we have to match up each pregnancy with a respondent. In theory we could iterate through the rows of the pregnancy table, use the `caseid` to find the corresponding respondent, and copy the values from the correspondent table into the pregnancy table. But that would be slow.

A better option is to recognize this process as a **join** operation as defined in SQL and other relational database languages (see). Join is implemented as a DataFrame method, so we can perform the operation like this:

```
live = live[live.prglngth>30]
resp = chap01soln.ReadFemResp()
resp.index = resp.caseid
join = live.join(resp, on='caseid', rsuffix='_r')
```

The first line selects records for pregnancies longer than 30 weeks, assuming that the office pool is formed several weeks before the due date.

The next line reads the respondent file. The result is a DataFrame with integer indices; in order to look up respondents efficiently, I replace `resp.index` with `resp.caseid`.

The `join` method is invoked on `live`, which is considered the "left" table, and passed `resp`, which is the "right" table. The keyword argument `on` indicates the variable used to match up rows from the two tables.

In this example some column names appear in both tables, so we have to provide `rsuffix`, which is a string that will be appended to the names of overlapping columns from the right table. For example, both tables have a column named `race` that encodes the race of the respondent. The result of the join contains two columns named `race` and `race_r`.

The pandas implementation is fast. Joining the NSFG tables takes less than a second on an ordinary desktop computer. Now we can start testing variables.

```
t = []
for name in join.columns:
    try:
        if join[name].var() < 1e-7:
            continue

        formula = 'totalwgt_lb ~ agepreg + ' + name
        model = smf.ols(formula, data=join)
        if model.nobs < len(join)/2:
            continue

        results = model.fit()
    except (ValueError, TypeError):
        continue

    t.append((results.rsquared, name))
```

For each variable we construct a model, compute R^2, and append the results to a list. The models all include `agepreg`, since we already know that it has some predictive power.

I check that each explanatory variable has some variability; otherwise the results of the regression are unreliable. I also check the number of observations for each model. Variables that contains a large number of `nans` are not good candidates for prediction.

For most of these variables, we haven't done any cleaning. Some of them are encoded in ways that don't work very well for linear regression. As a result, we might overlook some variables that would be useful if they were cleaned properly. But maybe we will find some good candidates.

Prediction

The next step is to sort the results and select the variables that yield the highest values of R^2.

```
t.sort(reverse=True)
for mse, name in t[:30]:
    print(name, mse)
```

The first variable on the list is `totalwgt_lb`, followed by `birthwgt_lb`. Obviously, we can't use birth weight to predict birth weight.

Similarly `prglngth` has useful predictive power, but for the office pool we assume pregnancy length (and the related variables) are not known yet.

The first useful predictive variable is `babysex` which indicates whether the baby is male or female. In the NSFG dataset, boys are about 0.3 lbs heavier. So, assuming that the sex of the baby is known, we can use it for prediction.

Next is `race`, which indicates whether the respondent is white, black, or other. As an explanatory variable, race can be problematic. In datasets like the NSFG, race is correlated with many other variables, including income and other socioeconomic factors. In a regression model, race acts as a **proxy variable**, so apparent correlations with race are often caused, at least in part, by other factors.

The next variable on the list is `nbrnaliv`, which indicates whether the pregnancy yielded multiple births. Twins and triplets tend to be smaller than other babies, so if we know whether our hypothetical co-worker is expecting twins, that would help.

Next on the list is `paydu`, which indicates whether the respondent owns her home. It is one of several income-related variables that turn out to be predictive. In datasets like the NSFG, income and wealth are correlated with just about everything. In this example, income is related to diet, health, health care, and other factors likely to affect birth weight.

Some of the other variables on the list are things that would not be known until later, like `bfeedwks`, the number of weeks the baby was breast fed. We can't use these variables for prediction, but you might want to speculate on reasons `bfeedwks` might be correlated with birth weight.

Sometimes you start with a theory and use data to test it. Other times you start with data and go looking for possible theories. The second approach, which this section demonstrates, is called **data mining**. An advantage of data mining is that it can discover unexpected patterns. A hazard is that many of the patterns it discovers are either random or spurious.

Having identified potential explanatory variables, I tested a few models and settled on this one:

```
formula = ('totalwgt_lb ~ agepreg + C(race) + babysex==1 + '
           'nbrnaliv>1 + paydu==1 + totincr')
results = smf.ols(formula, data=join).fit()
```

This formula uses some syntax we have not seen yet: `C(race)` tells the formula parser (Patsy) to treat race as a categorical variable, even though it is encoded numerically.

The encoding for `babysex` is 1 for male, 2 for female; writing `babysex==1` converts it to boolean, True for male and false for female.

Similarly `nbrnaliv>1` is True for multiple births and `paydu==1` is True for respondents who own their houses.

`totincr` is encoded numerically from 1-14, with each increment representing about $5000 in annual income. So we can treat these values as numerical, expressed in units of $5000.

Here are the results of the model:

```
Intercept               6.63    (0)
C(race)[T.2]            0.357   (5.43e-29)
C(race)[T.3]            0.266   (2.33e-07)
babysex == 1[T.True]    0.295   (5.39e-29)
nbrnaliv > 1[T.True]   -1.38    (5.1e-37)
paydu == 1[T.True]      0.12    (0.000114)
agepreg                 0.00741 (0.0035)
totincr                 0.0122  (0.00188)
```

The estimated parameters for race are larger than I expected, especially since we control for income. The encoding is 1 for black, 2 for white, and 3 for other. Babies of black mothers are lighter than babies of other races by 0.27–0.36 lbs.

As we've already seen, boys are heavier by about 0.3 lbs; twins and other multiplets are lighter by 1.4 lbs.

People who own their homes have heavier babies by about 0.12 lbs, even when we control for income. The parameter for mother's age is smaller than what we saw in "Multiple Regression" on page 131, which suggests that some of the other variables are correlated with age, probably including `paydu` and `totincr`.

All of these variables are statistically significant, some with very low p-values, but R^2 is only 0.06, still quite small. RMSE without using the model is 1.27 lbs; with the model it drops to 1.23. So your chance of winning the pool is not substantially improved. Sorry!

Logistic Regression

In the previous examples, some of the explanatory variables were numerical and some categorical (including boolean). But the dependent variable was always numerical.

Linear regression can be generalized to handle other kinds of dependent variables. If the dependent variable is boolean, the generalized model is called **logistic regression**. If the dependent variable is an integer count, it's called **Poisson regression**.

As an example of logistic regression, let's consider a variation on the office pool scenario. Suppose a friend of yours is pregnant and you want to predict whether the baby is a boy or a girl. You could use data from the NSFG to find factors that affect the "sex ratio", which is conventionally defined to be the probability of having a boy.

If you encode the dependent variable numerically, for example 0 for a girl and 1 for a boy, you could apply ordinary least squares, but there would be problems. The linear model might be something like this:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

Where y is the dependent variable, and x_1 and x_2 are explanatory variables. Then we could find the parameters that minimize the residuals.

The problem with this approach is that it produces predictions that are hard to interpret. Given estimated parameters and values for x_1 and x_2, the model might predict $y = 0.5$, but the only meaningful values of y are 0 and 1.

It is tempting to interpret a result like that as a probability; for example, we might say that a respondent with particular values of x_1 and x_2 has a 50% chance of having a boy. But it is also possible for this model to predict $y = 1.1$ or $y = -0.1$, and those are not valid probabilities.

Logistic regression avoids this problem by expressing predictions in terms of **odds** rather than probabilities. If you are not familiar with odds, "odds in favor" of an event is the ratio of the probability it will occur to the probability that it will not.

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

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

```
o = p / (1-p)
```

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

```
p = o / (o+1)
```

Logistic regression is based on the following model:

$$\log o = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

Where o is the odds in favor of a particular outcome; in the example, o would be the odds of having a boy.

Suppose we have estimated the parameters β_0, β_1, and β_2 (I'll explain how in a minute). And suppose we are given values for x_1 and x_2. We can compute the predicted value of $\log o$, and then convert to a probability:

```
o = np.exp(log_o)
p = o / (o+1)
```

So in the office pool scenario we could compute the predictive probability of having a boy. But how do we estimate the parameters?

Estimating Parameters

Unlike linear regression, logistic regression does not have a closed form solution, so it is solved by guessing an initial solution and improving it iteratively.

The usual goal is to find the maximum-likelihood estimate (MLE), which is the set of parameters that maximizes the likelihood of the data. For example, suppose we have the following data:

```
>>> y = np.array([0, 1, 0, 1])
>>> x1 = np.array([0, 0, 0, 1])
>>> x2 = np.array([0, 1, 1, 1])
```

And we start with the initial guesses $\beta_0 = -1.5$, $\beta_1 = 2.8$, and $\beta_2 = 1.1$:

```
>>> beta = [-1.5, 2.8, 1.1]
```

Then for each row we can compute `log_o`:

```
>>> log_o = beta[0] + beta[1] * x1 + beta[2] * x2
[-1.5 -0.4 -0.4  2.4]
```

And convert from log odds to probabilities:

```
>>> o = np.exp(log_o)
[  0.223   0.670   0.670  11.02  ]

>>> p = o / (o+1)
[ 0.182  0.401  0.401  0.916 ]
```

Notice that when `log_o` is greater than 0, `o` is greater than 1 and `p` is greater than 0.5.

The likelihood of an outcome is `p` when `y==1` and `1-p` when `y==0`. For example, if we think the probability of a boy is 0.8 and the outcome is a boy, the likelihood is 0.8; if the outcome is a girl, the likelihood is 0.2. We can compute that like this:

```
>>> likes = y * p + (1-y) * (1-p)
[ 0.817  0.401  0.598  0.916 ]
```

The overall likelihood of the data is the product of `likes`:

```
>>> like = np.prod(likes)
0.18
```

For these values of `beta`, the likelihood of the data is 0.18. The goal of logistic regression is to find parameters that maximize this likelihood. To do that, most statistics packages use an iterative solver like Newton's method.

Implementation

StatsModels provides an implementation of logistic regression called `logit`, named for the function that converts from probability to log odds. To demonstrate its use, I'll look for variables that affect the sex ratio.

Again, I load the NSFG data and select pregnancies longer than 30 weeks:

```
live, firsts, others = first.MakeFrames()
df = live[live.prglngth>30]
```

`logit` requires the dependent variable to be binary (rather than boolean), so I create a new column named `boy`, using `astype(int)` to convert to binary integers:

```
df['boy'] = (df.babysex==1).astype(int)
```

Factors that have been found to affect sex ratio include parents' age, birth order, race, and social status. We can use logistic regression to see if these effects appear in the NSFG data. I'll start with the mother's age:

```
import statsmodels.formula.api as smf

model = smf.logit('boy ~ agepreg', data=df)
results = model.fit()
SummarizeResults(results)
```

`logit` takes the same arguments as `ols`, a formula in Patsy syntax and a DataFrame. The result is a Logit object that represents the model. It contains attributes called `endog` and `exog` that contain the **endogenous variable**, another name for the dependent variable, and the **exogenous variables**, another name for the explanatory variables. Since they are NumPy arrays, it is sometimes convenient to convert them to DataFrames:

```
endog = pandas.DataFrame(model.endog, columns=[model.endog_names])
exog = pandas.DataFrame(model.exog, columns=model.exog_names)
```

The result of `model.fit` is a BinaryResults object, which is similar to the RegressionResults object we got from `ols`. Here is a summary of the results:

```
Intercept   0.00579   (0.953)
agepreg     0.00105   (0.783)
R^2 6.144e-06
```

The parameter of `agepreg` is positive, which suggests that older mothers are more likely to have boys, but the p-value is 0.783, which means that the apparent effect could easily be due to chance.

The coefficient of determination, R^2, does not apply to logistic regression, but there are several alternatives that are used as "pseudo R^2 values." These values can be useful for comparing models. For example, here's a model that includes several factors believed to be associated with sex ratio:

```
formula = 'boy ~ agepreg + hpagelb + birthord + C(race)'
model = smf.logit(formula, data=df)
results = model.fit()
```

Along with mother's age, this model includes father's age at birth (`hpagelb`), birth order (`birthord`), and race as a categorical variable. Here are the results:

```
Intercept      -0.0301     (0.772)
C(race)[T.2]   -0.0224     (0.66)
C(race)[T.3]   -0.000457   (0.996)
agepreg        -0.00267    (0.629)
hpagelb         0.0047     (0.266)
birthord        0.00501    (0.821)
R^2 0.000144
```

None of the estimated parameters are statistically significant. The pseudo-R^2 value is a little higher, but that could be due to chance.

Accuracy

In the office pool scenario, we are most interested in the accuracy of the model: the number of successful predictions, compared with what we would expect by chance.

In the NSFG data, there are more boys than girls, so the baseline strategy is to guess "boy" every time. The accuracy of this strategy is just the fraction of boys:

```
actual = endog['boy']
baseline = actual.mean()
```

Since `actual` is encoded in binary integers, the mean is the fraction of boys, which is 0.507.

Here's how we compute the accuracy of the model:

```
predict = (results.predict() >= 0.5)
true_pos = predict * actual
true_neg = (1 - predict) * (1 - actual)
```

`results.predict` returns a NumPy array of probabilities, which we round off to 0 or 1. Multiplying by `actual` yields 1 if we predict a boy and get it right, 0 otherwise. So, `true_pos` indicates "true positives".

Similarly, `true_neg` indicates the cases where we guess "girl" and get it right. Accuracy is the fraction of correct guesses:

```
acc = (sum(true_pos) + sum(true_neg)) / len(actual)
```

The result is 0.512, slightly better than the baseline, 0.507. But, you should not take this result too seriously. We used the same data to build and test the model, so the model may not have predictive power on new data.

Nevertheless, let's use the model to make a prediction for the office pool. Suppose your friend is 35 years old and white, her husband is 39, and they are expecting their third child:

```
columns = ['agepreg', 'hpagelb', 'birthord', 'race']
new = pandas.DataFrame([[35, 39, 3, 2]], columns=columns)
y = results.predict(new)
```

To invoke `results.predict` for a new case, you have to construct a DataFrame with a column for each variable in the model. The result in this case is 0.52, so you should guess "boy." But if the model improves your chances of winning, the difference is very small.

Exercises

My solution to these exercises is in `chap11soln.ipynb`.

Exercise 11-1.

Suppose one of your co-workers is expecting a baby and you are participating in an office pool to predict the date of birth. Assuming that bets are placed during the 30th week of pregnancy, what variables could you use to make the best prediction? You should limit yourself to variables that are known before the birth, and likely to be available to the people in the pool.

Exercise 11-2.

The Trivers-Willard hypothesis suggests that for many mammals the sex ratio depends on "maternal condition"; that is, factors like the mother's age, size, health, and social status.

Some studies have shown this effect among humans, but results are mixed. In this chapter we tested some variables related to these factors, but didn't find any with a statistically significant effect on sex ratio.

As an exercise, use a data mining approach to test the other variables in the pregnancy and respondent files. Can you find any factors with a substantial effect?

Exercise 11-3.

If the quantity you want to predict is a count, you can use Poisson regression, which is implemented in StatsModels with a function called `poisson`. It works the same way as `ols` and `logit`. As an exercise, let's use it to predict how many children a woman has born; in the NSFG dataset, this variable is called `numbabes`.

Suppose you meet a woman who is 35 years old, black, and a college graduate whose annual household income exceeds $75,000. How many children would you predict she has born?

Exercise 11-4.

If the quantity you want to predict is categorical, you can use multinomial logistic regression, which is implemented in StatsModels with a function called `mnlogit`. As an exercise, let's use it to guess whether a woman is married, cohabitating, widowed, divorced, separated, or never married; in the NSFG dataset, marital status is encoded in a variable called `rmarital`.

Suppose you meet a woman who is 25 years old, white, and a high school graduate whose annual household income is about $45,000. What is the probability that she is married, cohabitating, etc?

Glossary

regression
: One of several related processes for estimating parameters that fit a model to data.

dependent variables
: The variables in a regression model we would like to predict. Also known as endogenous variables.

explanatory variables
: The variables used to predict or explain the dependent variables. Also known as independent, or exogenous, variables.

simple regression
: A regression with only one dependent and one explanatory variable.

multiple regression
: A regression with multiple explanatory variables, but only one dependent variable.

linear regression
: A regression based on a linear model.

ordinary least squares
: A linear regression that estimates parameters by minimizing the squared error of the residuals.

spurious relationship
: A relationship between two variables that is caused by a statistical artifact or a factor, not included in the model, that is related to both variables.

control variable
: A variable included in a regression to eliminate or "control for" a spurious relationship.

proxy variable
: A variable that contributes information to a regression model indirectly because of a relationship with another factor, so it acts as a proxy for that factor.

categorical variable
: A variable that can have one of a discrete set of unordered values.

join
: An operation that combines data from two DataFrames using a key to match up rows in the two frames.

data mining
: An approach to finding relationships between variables by testing a large number of models.

logistic regression
: A form of regression used when the dependent variable is boolean.

Poisson regression
: A form of regression used when the dependent variable is a non-negative integer, usually a count.

odds
: An alternative way of representing a probability, p, as the ratio of the probability and its complement, $p / (1 - p)$.

CHAPTER 12

Time Series Analysis

A **time series** is a sequence of measurements from a system that varies in time. One famous example is the "hockey stick graph" that shows global average temperature over time.

The example I work with in this chapter comes from Zachary M. Jones, a researcher in political science who studies the black market for cannabis in the US. He collected data from a website called Price of Weed that crowdsources market information by asking participants to report the price, quantity, quality, and location of cannabis transactions. The goal of his project is to investigate the effect of policy decisions, like legalization, on markets. I find this project appealing because it is an example that uses data to address important political questions, like drug policy.

I hope you will find this chapter interesting, but I'll take this opportunity to reiterate the importance of maintaining a professional attitude to data analysis. Whether and which drugs should be illegal are important and difficult public policy questions; our decisions should be informed by accurate data reported honestly.

The code for this chapter is in `timeseries.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

Importing and Cleaning

The data I downloaded from Mr. Jones's site is in the repository for this book. The following code reads it into a pandas DataFrame:

```
transactions = pandas.read_csv('mj-clean.csv', parse_dates=[5])
```

`parse_dates` tells `read_csv` to interpret values in column 5 as dates and convert them to NumPy `datetime64` objects.

The DataFrame has a row for each reported transaction and the following columns:

city
: String city name

state
: Two-letter state abbreviation

price
: Price paid in dollars

amount
: Quantity purchased in grams

quality
: High, medium, or low quality, as reported by the purchaser

date
: Date of report, presumed to be shortly after date of purchase

ppg
: Price per gram in dollars

state.name
: String state name

lat
: Approximate latitude of the transaction, based on city name

lon
: Approximate longitude of the transaction

Each transaction is an event in time, so we could treat this dataset as a time series. But the events are not equally spaced in time; the number of transactions reported each day varies from 0 to several hundred. Many methods used to analyze time series require the measurements to be equally spaced, or at least things are simpler if they are.

In order to demonstrate these methods, I divide the dataset into groups by reported quality, and then transform each group into an equally spaced series by computing the mean daily price per gram.

```
    def GroupByQualityAndDay(transactions):
        groups = transactions.groupby('quality')
        dailies = {}
        for name, group in groups:
            dailies[name] = GroupByDay(group)

        return dailies
```

`groupby` is a DataFrame method that returns a GroupBy object, `groups`; used in a for loop, it iterates the names of the groups and the DataFrames that represent them. Since the values of `quality` are `low`, `medium`, and `high`, we get three groups with those names.

The loop iterates through the groups and calls `GroupByDay`, which computes the daily average price and returns a new DataFrame:

```
def GroupByDay(transactions, func=np.mean):
    grouped = transactions[['date', 'ppg']].groupby('date')
    daily = grouped.aggregate(func)

    daily['date'] = daily.index
    start = daily.date[0]
    one_year = np.timedelta64(1, 'Y')
    daily['years'] = (daily.date - start) / one_year

    return daily
```

The parameter, `transactions`, is a DataFrame that contains columns `date` and `ppg`. We select these two columns, then group by `date`.

The result, `grouped`, is a map from each date to a DataFrame that contains prices reported on that date. `aggregate` is a GroupBy method that iterates through the groups and applies a function to each column of the group; in this case there is only one column, `ppg`. So the result of `aggregate` is a DataFrame with one row for each date and one column, `ppg`.

Dates in these DataFrames are stored as NumPy `datetime64` objects, which are represented as 64-bit integers in nanoseconds. For some of the analyses coming up, it will be convenient to work with time in more human-friendly units, like years. So `GroupByDay` adds a column named `date` by copying the `index`, then adds `years`, which contains the number of years since the first transaction as a floating-point number.

The resulting DataFrame has columns `ppg`, `date`, and `years`.

Plotting

The result from `GroupByQualityAndDay` is a map from each quality to a DataFrame of daily prices. Here's the code I use to plot the three time series:

```
thinkplot.PrePlot(rows=3)
for i, (name, daily) in enumerate(dailies.items()):
    thinkplot.SubPlot(i+1)
    title = 'price per gram ($)' if i==0 else ''
    thinkplot.Config(ylim=[0, 20], title=title)
    thinkplot.Scatter(daily.index, daily.ppg, s=10, label=name)
    if i == 2:
        pyplot.xticks(rotation=30)
    else:
        thinkplot.Config(xticks=[])
```

`PrePlot` with `rows=3` means that we are planning to make three subplots laid out in three rows. The loop iterates through the DataFrames and creates a scatter plot for each. It is common to plot time series with line segments between the points, but in this case there are many data points and prices are highly variable, so adding lines would not help.

Since the labels on the x-axis are dates, I use `pyplot.xticks` to rotate the "ticks" 30 degrees, making them more readable.

Figure 12-1 shows the result. One apparent feature in these plots is a gap around November 2013. It's possible that data collection was not active during this time, or the data might not be available. We will consider ways to deal with this missing data later.

Visually, it looks like the price of high quality cannabis is declining during this period, and the price of medium quality is increasing. The price of low quality might also be increasing, but it is harder to tell, since it seems to be more volatile. Keep in mind that quality data is reported by volunteers, so trends over time might reflect changes in how participants apply these labels.

Linear Regression

Although there are methods specific to time series analysis, for many problems a simple way to get started is by applying general-purpose tools like linear regression. The following function takes a DataFrame of daily prices and computes a least squares fit, returning the model and results objects from StatsModels:

```
def RunLinearModel(daily):
    model = smf.ols('ppg ~ years', data=daily)
    results = model.fit()
    return model, results
```

Then we can iterate through the qualities and fit a model to each:

```
for name, daily in dailies.items():
    model, results = RunLinearModel(daily)
    print(name)
    regression.SummarizeResults(results)
```

Here are the results:

Quality	Intercept	Slope	R^2
High	13.450	-0.708	0.444
Medium	8.879	0.283	0.050
Low	5.362	0.568	0.030

The estimated slopes indicate that the price of high quality cannabis dropped by about 71 cents per year during the observed interval; for medium quality it increased by 28

cents per year, and for low quality it increased by 57 cents per year. These estimates are all statistically significant with very small p-values.

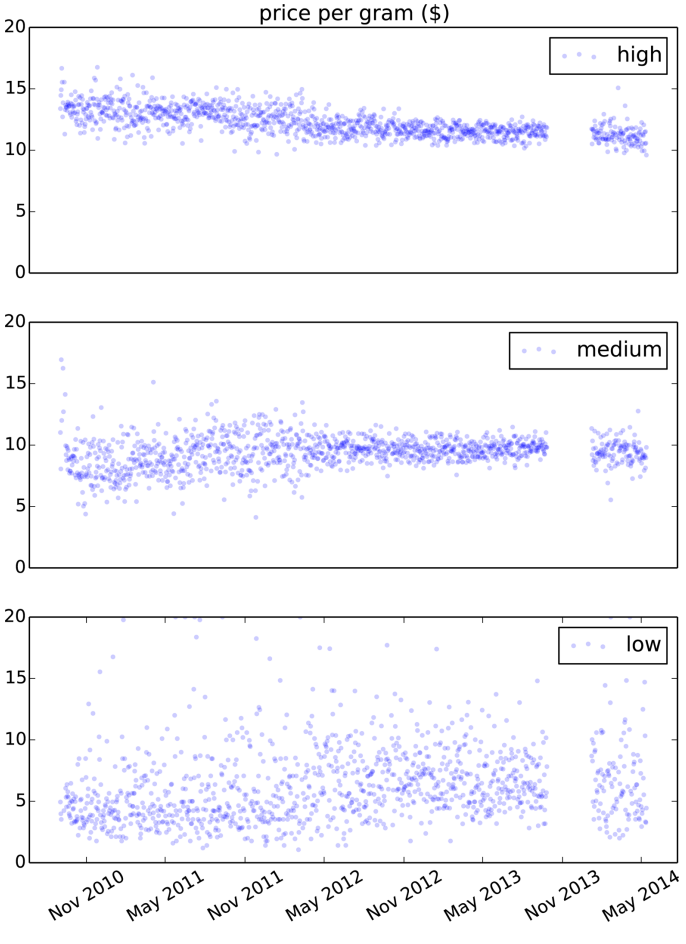


Figure 12-1. Time series of daily price per gram for high, medium, and low quality cannabis

The R^2 value for high quality cannabis is 0.44, which means that time as an explanatory variable accounts for 44% of the observed variability in price. For the other qualities, the change in price is smaller, and variability in prices is higher, so the values of R^2 are smaller (but still statistically significant).

The following code plots the observed prices and the fitted values:

```
def PlotFittedValues(model, results, label=''):
    years = model.exog[:,1]
    values = model.endog
    thinkplot.Scatter(years, values, s=15, label=label)
    thinkplot.Plot(years, results.fittedvalues, label='model')
```

As we saw in "Implementation" on page 140, `model` contains `exog` and `endog`, NumPy arrays with the exogenous (explanatory) and endogenous (dependent) variables.

`PlotFittedValues` makes a scatter plot of the data points and a line plot of the fitted values. Figure 12-2 shows the results for high quality cannabis.

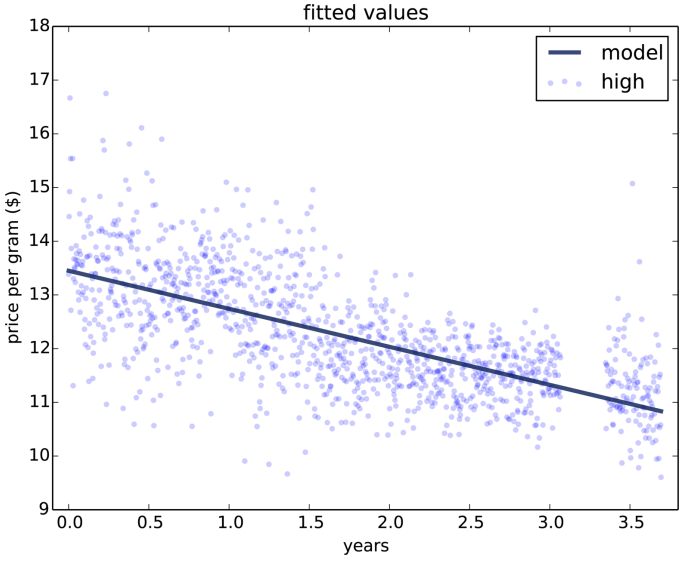


Figure 12-2. Time series of daily price per gram for high quality cannabis, and a linear least squares fit

The model seems like a good linear fit for the data; nevertheless, linear regression is not the most appropriate choice for this data:

- First, there is no reason to expect the long-term trend to be a line or any other simple function. In general, prices are determined by supply and demand, both of which vary over time in unpredictable ways.
- Second, the linear regression model gives equal weight to all data, recent and past. For purposes of prediction, we should probably give more weight to recent data.

- Finally, one of the assumptions of linear regression is that the residuals are uncorrelated noise. With time series data, this assumption is often false because successive values are correlated.

The next section presents an alternative that is more appropriate for time series data.

Moving Averages

Most time series analysis is based on the modeling assumption that the observed series is the sum of three components:

Trend
: A smooth function that captures persistent changes

Seasonality
: Periodic variation, possibly including daily, weekly, monthly, or yearly cycles

Noise
: Random variation around the longterm trend

Regression is one way to extract the trend from a series, as we saw in the previous section. But if the trend is not a simple function, a good alternative is a **moving average**. A moving average divides the series into overlapping regions, called **windows**, and computes the average of the values in each window.

One of the simplest moving averages is the **rolling mean**, which computes the mean of the values in each window. For example, if the window size is 3, the rolling mean computes the mean of values 0 through 2, 1 through 3, 2 through 4, etc.

pandas provides `rolling_mean`, which takes a Series and a window size and returns a new Series.

```
>>> series = np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> pandas.rolling_mean(series, 3)
array([ nan,  nan,   1,   2,   3,   4,   5,   6,   7,   8])
```

The first two values are `nan`; the next value is the mean of the first three elements, 0, 1, and 2. The next value is the mean of 1, 2, and 3. And so on.

Before we can apply `rolling_mean` to the cannabis data, we have to deal with missing values. There are a few days in the observed interval with no reported transactions for one or more quality categories, and a period in 2013 when data collection was not active.

In the DataFrames we have used so far, these dates are absent; the index skips days with no data. For the analysis that follows, we need to represent this missing data explicitly. We can do that by "reindexing" the DataFrame:

```
dates = pandas.date_range(daily.index.min(), daily.index.max())
reindexed = daily.reindex(dates)
```

The first line computes a date range that includes every day from the beginning to the end of the observed interval. The second line creates a new DataFrame with all of the data from `daily`, but including rows for all dates, filled with `nan`.

Now we can plot the rolling mean like this:

```
roll_mean = pandas.rolling_mean(reindexed.ppg, 30)
thinkplot.Plot(roll_mean.index, roll_mean)
```

The window size is 30, so each value in `roll_mean` is the mean of 30 values from `reindexed.ppg`.

Figure 12-3 (left) shows the result. The rolling mean seems to do a good job of smoothing out the noise and extracting the trend. The first 29 values are `nan`, and wherever there's a missing value, it's followed by another 29 `nan`s. There are ways to fill in these gaps, but they are a minor nuisance.

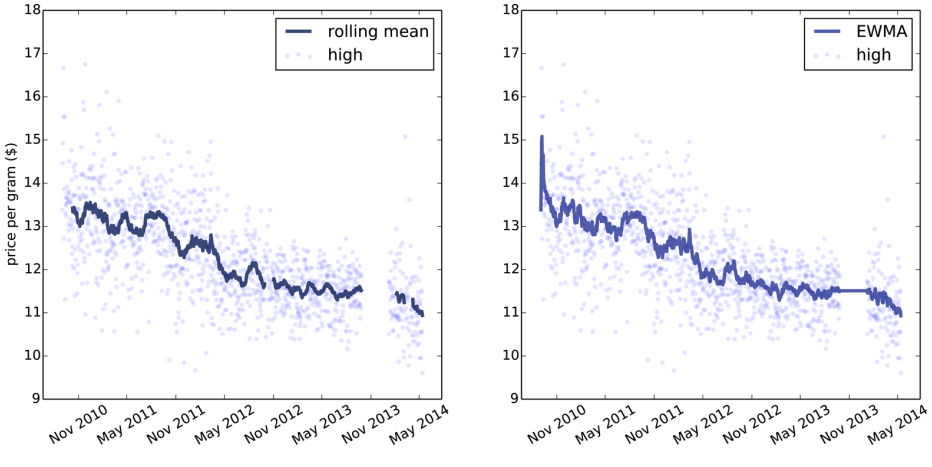


Figure 12-3. Daily price and a rolling mean (left) and exponentially-weighted moving average (right)

An alternative is the **exponentially-weighted moving average** (EWMA), which has two advantages. First, as the name suggests, it computes a weighted average where the most recent value has the highest weight and the weights for previous values drop off exponentially. Second, the pandas implementation of EWMA handles missing values better.

```
ewma = pandas.ewma(reindexed.ppg, span=30)
thinkplot.Plot(ewma.index, ewma)
```

The **span** parameter corresponds roughly to the window size of a moving average; it controls how fast the weights drop off, so it determines the number of points that make a non-negligible contribution to each average.

Figure 12-3 (right) shows the EWMA for the same data. It is similar to the rolling mean, where they are both defined, but it has no missing values, which makes it easier to work with. The values are noisy at the beginning of the time series, because they are based on fewer data points.

Missing Values

Now that we have characterized the trend of the time series, the next step is to investigate seasonality, which is periodic behavior. Time series data based on human behavior often exhibits daily, weekly, monthly, or yearly cycles. In the next section I present methods to test for seasonality, but they don't work well with missing data, so we have to solve that problem first.

A simple and common way to fill missing data is to use a moving average. The Series method `fillna` does just what we want:

```
reindexed.ppg.fillna(ewma, inplace=True)
```

Wherever `reindexed.ppg` is `nan`, `fillna` replaces it with the corresponding value from `ewma`. The `inplace` flag tells `fillna` to modify the existing Series rather than create a new one.

A drawback of this method is that it understates the noise in the series. We can solve that problem by adding in resampled residuals:

```
resid = (reindexed.ppg - ewma).dropna()
fake_data = ewma + thinkstats2.Resample(resid, len(reindexed))
reindexed.ppg.fillna(fake_data, inplace=True)
```

`resid` contains the residual values, not including days when `ppg` is `nan`. `fake_data` contains the sum of the moving average and a random sample of residuals. Finally, `fillna` replaces `nan` with values from `fake_data`.

Figure 12-4 shows the result. The filled data is visually similar to the actual values. Since the resampled residuals are random, the results are different every time; later we'll see how to characterize the error created by missing values.

Serial Correlation

As prices vary from day to day, you might expect to see patterns. If the price is high on Monday, you might expect it to be high for a few more days; and if it's low, you might expect it to stay low. A pattern like this is called **serial correlation**, because each value is correlated with the next one in the series.

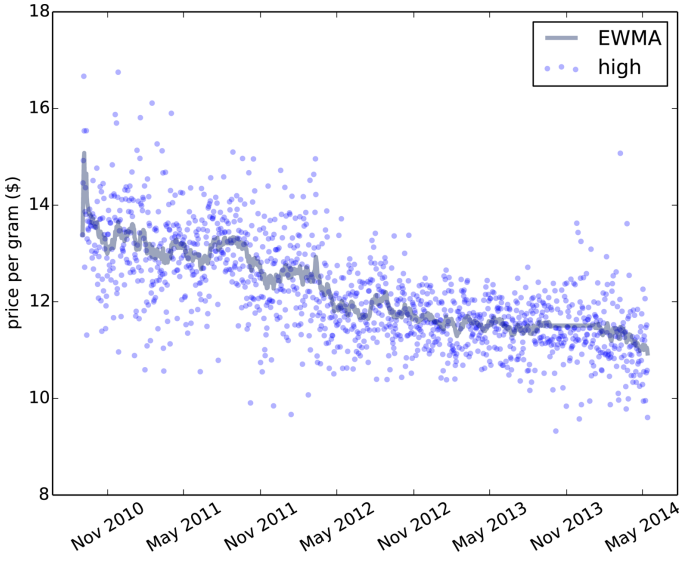


Figure 12-4. Daily price with filled data

To compute serial correlation, we can shift the time series by an interval called a **lag**, and then compute the correlation of the shifted series with the original:

```
def SerialCorr(series, lag=1):
    xs = series[lag:]
    ys = series.shift(lag)[lag:]
    corr = thinkstats2.Corr(xs, ys)
    return corr
```

After the shift, the first `lag` values are `nan`, so I use a slice to remove them before computing `Corr`.

If we apply `SerialCorr` to the raw price data with lag 1, we find serial correlation 0.48 for the high quality category, 0.16 for medium and 0.10 for low. In any time series with a long-term trend, we expect to see strong serial correlations; for example, if prices are falling, we expect to see values above the mean in the first half of the series and values below the mean in the second half.

It is more interesting to see if the correlation persists if you subtract away the trend. For example, we can compute the residual of the EWMA and then compute its serial correlation:

```
ewma = pandas.ewma(reindexed.ppg, span=30)
resid = reindexed.ppg - ewma
corr = SerialCorr(resid, 1)
```

With lag=1, the serial correlations for the de-trended data are -0.022 for high quality, -0.015 for medium, and 0.036 for low. These values are small, indicating that there is little or no one-day serial correlation in this series.

To check for weekly, monthly, and yearly seasonality, I ran the analysis again with different lags. Here are the results:

Lag	High	Medium	Low
1	-0.029	-0.014	0.034
7	0.02	-0.042	-0.0097
30	0.014	-0.0064	-0.013
365	0.045	0.015	0.033

In the next section, we'll test whether these correlations are statistically significant (they are not), but at this point we can tentatively conclude that there are no substantial seasonal patterns in these series, at least not with these lags.

Autocorrelation

If you think a series might have some serial correlation, but you don't know which lags to test, you can test them all! The **autocorrelation function** is a function that maps from lag to the serial correlation with the given lag. "Autocorrelation" is another name for serial correlation, used more often when the lag is not 1.

StatsModels, which we used for linear regression in "StatsModels" on page 130, also provides functions for time series analysis, including `acf`, which computes the autocorrelation function:

```
import statsmodels.tsa.stattools as smtsa
acf = smtsa.acf(filled.resid, nlags=365, unbiased=True)
```

`acf` computes serial correlations with lags from 0 through `nlags`. The `unbiased` flag tells `acf` to correct the estimates for the sample size. The result is an array of correlations. If we select daily prices for high quality, and extract correlations for lags 1, 7, 30, and 365, we can confirm that `acf` and `SerialCorr` yield approximately the same results:

```
>>> acf[0], acf[1], acf[7], acf[30], acf[365]
1.000, -0.029, 0.020, 0.014, 0.044
```

With `lag=0`, `acf` computes the correlation of the series with itself, which is always 1.

Figure 12-5 (left) shows autocorrelation functions for the three quality categories, with `nlags=40`. The gray region shows the normal variability we would expect if there is no actual autocorrelation; anything that falls outside this range is statistically significant, with a p-value less than 5%. Since the false positive rate is 5%, and we are computing 120 correlations (40 lags for each of 3 times series), we expect to see about 6 points

outside this region. In fact, there are 7. We conclude that there are no autocorrelations in these series that could not be explained by chance.

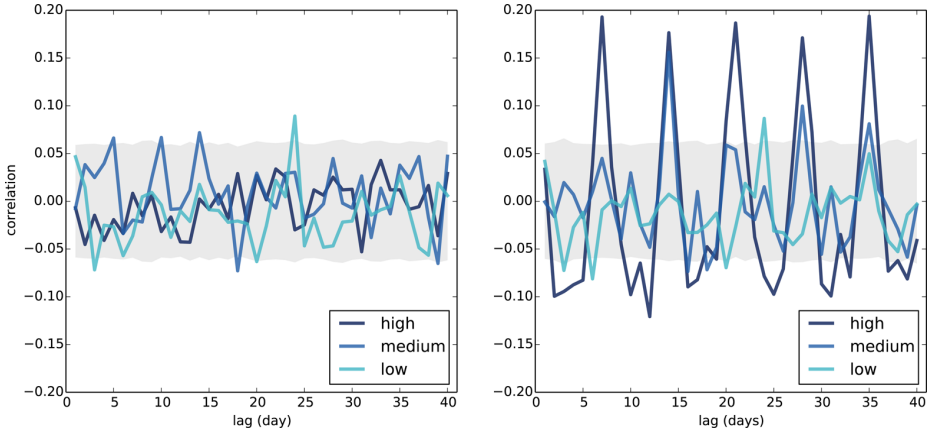


Figure 12-5. Autocorrelation function for daily prices (left), and daily prices with a simulated weekly seasonality (right)

I computed the gray regions by resampling the residuals. You can see my code in `timeseries.py`; the function is called `SimulateAutocorrelation`.

To see what the autocorrelation function looks like when there is a seasonal component, I generated simulated data by adding a weekly cycle. Assuming that demand for cannabis is higher on weekends, we might expect the price to be higher. To simulate this effect, I select dates that fall on Friday or Saturday and add a random amount to the price, chosen from a uniform distribution from $0 to $2.

```
def AddWeeklySeasonality(daily):
    frisat = (daily.index.dayofweek==4) | (daily.index.dayofweek==5)
    fake = daily.copy()
    fake.ppg[frisat] += np.random.uniform(0, 2, frisat.sum())
    return fake
```

`frisat` is a boolean Series, `True` if the day of the week is Friday or Saturday. `fake` is a new DataFrame, initially a copy of `daily`, which we modify by adding random values to `ppg`. `frisat.sum()` is the total number of Fridays and Saturdays, which is the number of random values we have to generate.

Figure 12-5 (right) shows autocorrelation functions for prices with this simulated seasonality. As expected, the correlations are highest when the lag is a multiple of 7. For high and medium quality, the new correlations are statistically significant. For low quality they are not, because residuals in this category are large; the effect would have to be bigger to be visible through the noise.

Prediction

Time series analysis can be used to investigate, and sometimes explain, the behavior of systems that vary in time. It can also make predictions.

The linear regressions we used in "Linear Regression" on page 148 can be used for prediction. The RegressionResults class provides `predict`, which takes a DataFrame containing the explanatory variables and returns a sequence of predictions. Here's the code:

```
def GenerateSimplePrediction(results, years):
    n = len(years)
    inter = np.ones(n)
    d = dict(Intercept=inter, years=years)
    predict_df = pandas.DataFrame(d)
    predict = results.predict(predict_df)
    return predict
```

`results` is a RegressionResults object; `years` is the sequence of time values we want predictions for. The function constructs a DataFrame, passes it to `predict`, and returns the result.

If all we want is a single, best-guess prediction, we're done. But for most purposes it is important to quantify error. In other words, we want to know how accurate the prediction is likely to be.

There are three sources of error we should take into account:

Sampling error
: The prediction is based on estimated parameters, which depend on random variation in the sample. If we run the experiment again, we expect the estimates to vary.

Random variation
: Even if the estimated parameters are perfect, the observed data varies randomly around the long-term trend, and we expect this variation to continue in the future.

Modeling error
: We have already seen evidence that the long-term trend is not linear, so predictions based on a linear model will eventually fail.

Another source of error to consider is unexpected future events. Agricultural prices are affected by weather, and all prices are affected by politics and law. As I write this, cannabis is legal in two states and legal for medical purposes in 20 more. If more states legalize it, the price is likely to go down. But if the federal government cracks down, the price might go up.

Modeling errors and unexpected future events are hard to quantify. Sampling error and random variation are easier to deal with, so we'll do that first.

To quantify sampling error, I use resampling, as we did in "Estimation" on page 120. As always, the goal is to use the actual observations to simulate what would happen if we ran the experiment again. The simulations are based on the assumption that the estimated parameters are correct, but the random residuals could have been different. Here is a function that runs the simulations:

```
def SimulateResults(daily, iters=101):
    model, results = RunLinearModel(daily)
    fake = daily.copy()

    result_seq = []
    for i in range(iters):
        fake.ppg = results.fittedvalues + Resample(results.resid)
        _, fake_results = RunLinearModel(fake)
        result_seq.append(fake_results)

    return result_seq
```

`daily` is a DataFrame containing the observed prices; `iters` is the number of simulations to run.

`SimulateResults` uses `RunLinearModel`, from "Linear Regression" on page 148, to estimate the slope and intercept of the observed values.

Each time through the loop, it generates a "fake" dataset by resampling the residuals and adding them to the fitted values. Then it runs a linear model on the fake data and stores the RegressionResults object.

The next step is to use the simulated results to generate predictions:

```
def GeneratePredictions(result_seq, years, add_resid=False):
    n = len(years)
    d = dict(Intercept=np.ones(n), years=years, years2=years**2)
    predict_df = pandas.DataFrame(d)

    predict_seq = []
    for fake_results in result_seq:
        predict = fake_results.predict(predict_df)
        if add_resid:
            predict += thinkstats2.Resample(fake_results.resid, n)
        predict_seq.append(predict)

    return predict_seq
```

`GeneratePredictions` takes the sequence of results from the previous step, as well as `years`, which is a sequence of floats that specifies the interval to generate predictions for, and `add_resid`, which indicates whether it should add resampled residuals to the straight-line prediction. `GeneratePredictions` iterates through the sequence of RegressionResults and generates a sequence of predictions.

Finally, here's the code that plots a 90% confidence interval for the predictions:

```
def PlotPredictions(daily, years, iters=101, percent=90):
    result_seq = SimulateResults(daily, iters=iters)
    p = (100 - percent) / 2
    percents = p, 100-p

    predict_seq = GeneratePredictions(result_seq, years, True)
    low, high = thinkstats2.PercentileRows(predict_seq, percents)
    thinkplot.FillBetween(years, low, high, alpha=0.3, color='gray')

    predict_seq = GeneratePredictions(result_seq, years, False)
    low, high = thinkstats2.PercentileRows(predict_seq, percents)
    thinkplot.FillBetween(years, low, high, alpha=0.5, color='gray')
```

`PlotPredictions` calls `GeneratePredictions` twice: once with `add_resid=True` and again with `add_resid=False`. It uses `PercentileRows` to select the 5th and 95th percentiles for each year, then plots a gray region between these bounds.

Figure 12-6 shows the result. The dark gray region represents a 90% confidence interval for the sampling error; that is, uncertainty about the estimated slope and intercept due to sampling.

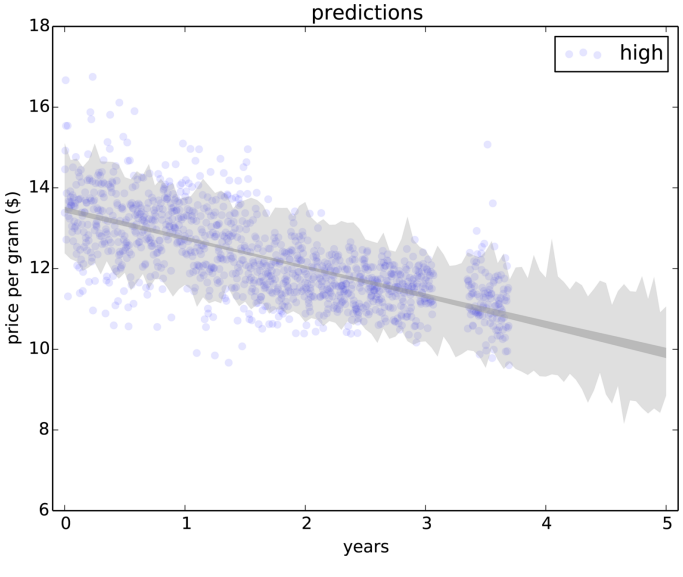


Figure 12-6. Predictions based on linear fits, showing variation due to sampling error and prediction error

The lighter region shows a 90% confidence interval for prediction error, which is the sum of sampling error and random variation.

These regions quantify sampling error and random variation, but not modeling error. In general modeling error is hard to quantify, but in this case we can address at least one source of error, unpredictable external events.

The regression model is based on the assumption that the system is **stationary**; that is, that the parameters of the model don't change over time. Specifically, it assumes that the slope and intercept are constant, as well as the distribution of residuals.

But looking at the moving averages in Figure 12-3, it seems like the slope changes at least once during the observed interval, and the variance of the residuals seems bigger in the first half than the second.

As a result, the parameters we get depend on the interval we observe. To see how much effect this has on the predictions, we can extend `SimulateResults` to use intervals of observation with different start and end dates. My implementation is in `timeseries.py`.

Figure 12-7 shows the result for the medium quality category. The lightest gray area shows a confidence interval that includes uncertainty due to sampling error, random variation, and variation in the interval of observation.

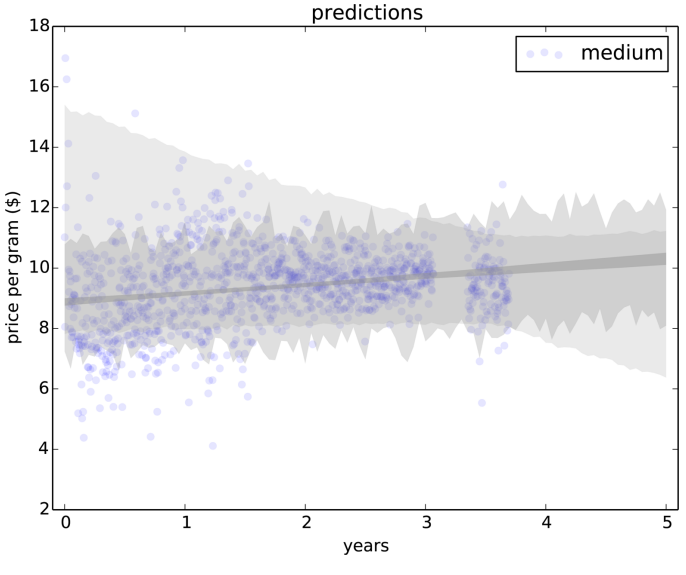


Figure 12-7. Predictions based on linear fits, showing variation due to the interval of observation

The model based on the entire interval has positive slope, indicating that prices were increasing. But the most recent interval shows signs of decreasing prices, so models

based on the most recent data have negative slope. As a result, the widest predictive interval includes the possibility of decreasing prices over the next year.

Further Reading

Time series analysis is a big topic; this chapter has only scratched the surface. An important tool for working with time series data is autoregression, which I did not cover here, mostly because it turns out not to be useful for the example data I worked with.

But once you have learned the material in this chapter, you are well prepared to learn about autoregression. One resource I recommend is Philipp Janert's book, *Data Analysis with Open Source Tools*, O'Reilly Media, 2011. His chapter on time series analysis picks up where this one leaves off.

Exercises

My solution to these exercises is in `chap12soln.py`.

Exercise 12-1.

The linear model I used in this chapter has the obvious drawback that it is linear, and there is no reason to expect prices to change linearly over time. We can add flexibility to the model by adding a quadratic term, as we did in "Nonlinear Relationships" on page 133.

Use a quadratic model to fit the time series of daily prices, and use the model to generate predictions. You will have to write a version of `RunLinearModel` that runs that quadratic model, but after that you should be able to reuse code in `timeseries.py` to generate predictions.

Exercise 12-2.

Write a definition for a class named `SerialCorrelationTest` that extends `HypothesisTest` from "HypothesisTest" on page 102. It should take a series and a lag as data, compute the serial correlation of the series with the given lag, and then compute the p-value of the observed correlation.

Use this class to test whether the serial correlation in raw price data is statistically significant. Also test the residuals of the linear model and (if you did the previous exercise), the quadratic model.

Exercise 12-3.

There are several ways to extend the EWMA model to generate predictions. One of the simplest is something like this:

1. Compute the EWMA of the time series and use the last point as an intercept, `inter`.
2. Compute the EWMA of differences between successive elements in the time series and use the last point as a slope, `slope`.
3. To predict values at future times, compute `inter + slope * dt`, where `dt` is the difference between the time of the prediction and the time of the last observation.

Use this method to generate predictions for a year after the last observation. A few hints:

- Use `timeseries.FillMissing` to fill in missing values before running this analysis. That way the time between consecutive elements is consistent.
- Use `Series.diff` to compute differences between successive elements.
- Use `reindex` to extend the DataFrame index into the future.
- Use `fillna` to put your predicted values into the DataFrame.

Glossary

time series
: A dataset where each value is associated with a timestamp, often a series of measurements and the times they were collected.

window
: A sequence of consecutive values in a time series, often used to compute a moving average.

moving average
: One of several statistics intended to estimate the underlying trend in a time series by computing averages (of some kind) for a series of overlapping windows.

rolling mean
: A moving average based on the mean value in each window.

exponentially-weighted moving average (EWMA)
: A moving average based on a weighted mean that gives the highest weight to the most recent values, and exponentially decreasing weights to earlier values.

span
: A parameter of EWMA that determines how quickly the weights decrease.

serial correlation
: Correlation between a time series and a shifted or lagged version of itself.

lag
: The size of the shift in a serial correlation or autocorrelation.

autocorrelation
: A more general term for a serial correlation with any amount of lag.

autocorrelation function
: A function that maps from lag to serial correlation.

stationary
: A model is stationary if the parameters and the distribution of residuals does not change over time.

CHAPTER 13
Survival Analysis

Survival analysis is a way to describe how long things last. It is often used to study human lifetimes, but it also applies to "survival" of mechanical and electronic components, or more generally to intervals in time before an event.

If someone you know has been diagnosed with a life-threatening disease, you might have seen a "5-year survival rate," which is the probability of surviving five years after diagnosis. That estimate and related statistics are the result of survival analysis.

The code in this chapter is in `survival.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

Survival Curves

The fundamental concept in survival analysis is the **survival curve**, $S(t)$, which is a function that maps from a duration, t, to the probability of surviving longer than t. If you know the distribution of durations, or "lifetimes", finding the survival curve is easy; it's just the complement of the CDF:

$$S(t) = 1 - \text{CDF}(t)$$

where $CDF(t)$ is the probability of a lifetime less than or equal to t.

For example, in the NSFG dataset, we know the duration of 11189 complete pregnancies. We can read this data and compute the CDF:

```
preg = nsfg.ReadFemPreg()
complete = preg.query('outcome in [1, 3, 4]').prglngth
cdf = thinkstats2.Cdf(complete, label='cdf')
```

The outcome codes `1, 3, 4` indicate live birth, stillbirth, and miscarriage. For this analysis I am excluding induced abortions, ectopic pregnancies, and pregnancies that were in progress when the respondent was interviewed.

The DataFrame method `query` takes a boolean expression and evaluates it for each row, selecting the rows that yield True.

Figure 13-1 (top) shows the CDF of pregnancy length and its complement, the survival function. To represent the survival function, I define an object that wraps a Cdf and adapts the interface:

```
class SurvivalFunction(object):
    def __init__(self, cdf, label=''):
        self.cdf = cdf
        self.label = label or cdf.label

    @property
    def ts(self):
        return self.cdf.xs

    @property
    def ss(self):
        return 1 - self.cdf.ps
```

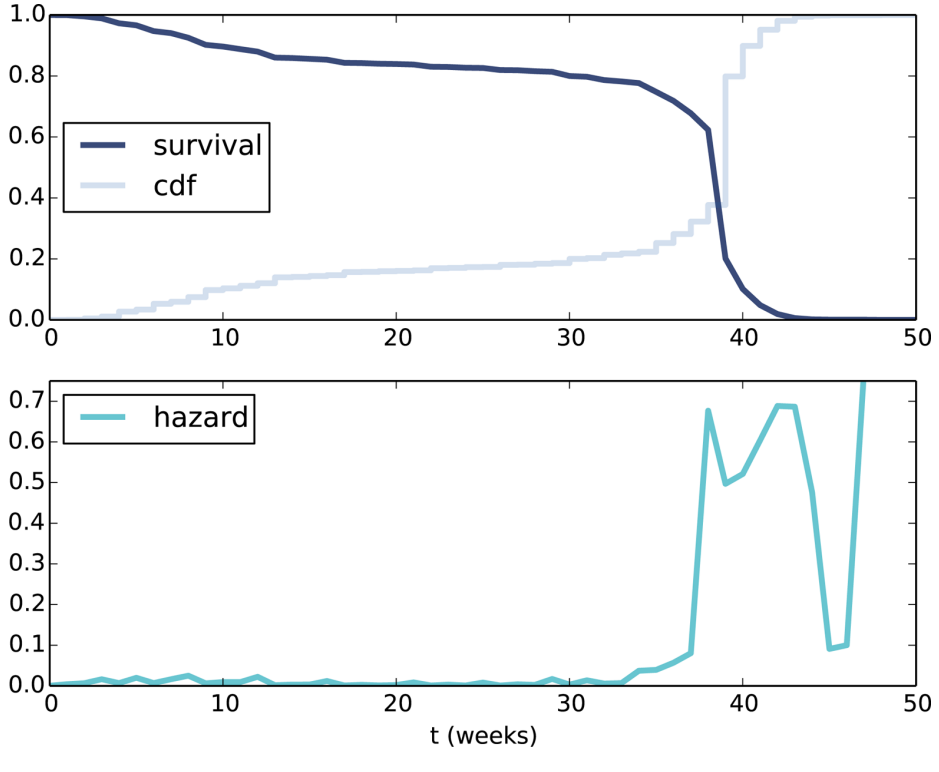


Figure 13-1. Cdf and survival function for pregnancy length (top), hazard function (bottom)

`SurvivalFunction` provides two properties: `ts`, which is the sequence of lifetimes, and `ss`, which is the survival function. In Python, a "property" is a method that can be invoked as if it were a variable.

We can instantiate a `SurvivalFunction` by passing the CDF of lifetimes:

```
sf = SurvivalFunction(cdf)
```

`SurvivalFunction` also provides `__getitem__` and `Prob`, which evaluate the survival function:

```
# class SurvivalFunction

    def __getitem__(self, t):
        return self.Prob(t)

    def Prob(self, t):
        return 1 - self.cdf.Prob(t)
```

For example, `sf[13]` is the fraction of pregnancies that proceed past the first trimester:

```
>>> sf[13]
0.86022
>>> cdf[13]
0.13978
```

About 86% of pregnancies proceed past the first trimester; about 14% do not.

`SurvivalFunction` provides `Render`, so we can plot `sf` using the functions in `thinkplot`:

```
thinkplot.Plot(sf)
```

Figure 13-1 (top) shows the result. The curve is nearly flat between 13 and 26 weeks, which shows that few pregnancies end in the second trimester. And the curve is steepest around 39 weeks, which is the most common pregnancy length.

Hazard Function

From the survival function we can derive the **hazard function**; for pregnancy lengths, the hazard function maps from a time, t, to the fraction of pregnancies that continue until t and then end at t. To be more precise:

$$\lambda(t) = \frac{S(t) - S(t+1)}{S(t)}$$

The numerator is the fraction of lifetimes that end at t, which is also PMF(t).

`SurvivalFunction` provides `MakeHazard`, which calculates the hazard function:

```
# class SurvivalFunction

    def MakeHazard(self, label=''):
        ss = self.ss
        lams = {}
        for i, t in enumerate(self.ts[:-1]):
            hazard = (ss[i] - ss[i+1]) / ss[i]
            lams[t] = hazard

        return HazardFunction(lams, label=label)
```

The `HazardFunction` object is a wrapper for a pandas Series:

```
class HazardFunction(object):

    def __init__(self, d, label=''):
        self.series = pandas.Series(d)
        self.label = label
```

`d` can be a dictionary or any other type that can initialize a Series, including another Series. `label` is a string used to identify the HazardFunction when plotted.

`HazardFunction` provides `__getitem__`, so we can evaluate it like this:

```
>>> hf = sf.MakeHazard()
>>> hf[39]
0.49689
```

So of all pregnancies that proceed until week 39, about 50% end in week 39.

Figure 13-1 (bottom) shows the hazard function for pregnancy lengths. For times after week 42, the hazard function is erratic because it is based on a small number of cases. Other than that the shape of the curve is as expected: it is highest around 39 weeks, and a little higher in the first trimester than in the second.

The hazard function is useful in its own right, but it is also an important tool for estimating survival curves, as we'll see in the next section.

Estimating Survival Curves

If someone gives you the CDF of lifetimes, it is easy to compute the survival and hazard functions. But in many real-world scenarios, we can't measure the distribution of lifetimes directly. We have to infer it.

For example, suppose you are following a group of patients to see how long they survive after diagnosis. Not all patients are diagnosed on the same day, so at any point in time, some patients have survived longer than others. If some patients have died, we know their survival times. For patients who are still alive, we don't know survival times, but we have a lower bound.

If we wait until all patients are dead, we can compute the survival curve, but if we are evaluating the effectiveness of a new treatment, we can't wait that long! We need a way to estimate survival curves using incomplete information.

As a more cheerful example, I will use NSFG data to quantify how long respondents "survive" until they get married for the first time. The range of respondents' ages is 14 to 44 years, so the dataset provides a snapshot of women at different stages in their lives.

For women who have been married, the dataset includes the date of their first marriage and their age at the time. For women who have not been married, we know their age when interviewed, but have no way of knowing when or if they will get married.

Since we know the age at first marriage for *some* women, it might be tempting to exclude the rest and compute the CDF of the known data. That is a bad idea. The result would be doubly misleading: (1) older women would be overrepresented, because they are more likely to be married when interviewed, and (2) married women would be overrepresented! In fact, this analysis would lead to the conclusion that all women get married, which is obviously incorrect.

Kaplan-Meier Estimation

In this example it is not only desirable but necessary to include observations of unmarried women, which brings us to one of the central algorithms in survival analysis, **Kaplan-Meier estimation**.

The general idea is that we can use the data to estimate the hazard function, then convert the hazard function to a survival function. To estimate the hazard function, we consider, for each age, (1) the number of women who got married at that age and (2) the number of women "at risk" of getting married, which includes all women who were not married at an earlier age.

Here's the code:

```
def EstimateHazardFunction(complete, ongoing, label=''):

    n = len(complete)
    hist_complete = thinkstats2.Hist(complete)
    sf_complete = SurvivalFunction(thinkstats2.Cdf(complete))

    m = len(ongoing)
    sf_ongoing = SurvivalFunction(thinkstats2.Cdf(ongoing))

    lams = {}
    for t, ended in sorted(hist_complete.Items()):
        at_risk = ended + n * sf_complete[t] + m * sf_ongoing[t]
        lams[t] = ended / at_risk

    return HazardFunction(lams, label=label)
```

`complete` is the set of complete observations; in this case, the ages when respondents got married. `ongoing` is the set of incomplete observations; that is, the ages of unmarried women when they were interviewed.

First, we precompute `hist_complete`, which is the Hist of ages when women were married, `sf_complete`, the survival function for married women, and `sf_ongoing`, the survival function for unmarried women.

The loop iterates through the ages when respondents got married. For each value of `t`, we have `ended`, which is the number of women who got married at age `t`. Then we compute the number of women "at risk", which is the sum of:

- `ended`, the number of respondents married at age `t`,
- `n * sf_complete[t]`, the number of respondents married after age `t`.
- `m * sf_ongoing[t]`, the number of unmarried respondents interviewed after `t`, and therefore known not to have been married at or before `t`.

The estimated value of the hazard function at `t` is the ratio of `ended` to `at_risk`.

`lams` is a dictionary that maps from t to $\lambda(t)$. The result is a `HazardFunction` object.

The Marriage Curve

To test this function, we have to do some data cleaning and transformation. The NSFG variables we need are:

`cmbirth`
: The respondent's date of birth, known for all respondents.

`cmintvw`
: The date the respondent was interviewed, known for all respondents.

`cmmarrhx`
: The date the respondent was first married, if applicable and known.

`evrmarry`
: This is 1 if the respondent had been married prior to the date of interview; otherwise, 0.

The first three variables are encoded in "century-months"; that is, the integer number of months since December 1899. So century-month 1 is January 1900.

First, we read the respondent file and replace invalid values of `cmmarrhx`:

```
resp = chap01soln.ReadFemResp()
resp.cmmarrhx.replace([9997, 9998, 9999], np.nan, inplace=True)
```

Then we compute each respondent's age when married and age when interviewed:

```
resp['agemarry'] = (resp.cmmarrhx - resp.cmbirth) / 12.0
resp['age'] = (resp.cmintvw - resp.cmbirth) / 12.0
```

Next we extract `complete`, which is the age at marriage for women who have been married, and `ongoing`, which is the age at interview for women who have not:

```
complete = resp[resp.evrmarry==1].agemarry
ongoing = resp[resp.evrmarry==0].age
```

Finally we compute the hazard function.

```
hf = EstimateHazardFunction(complete, ongoing)
```

Figure 13-2 (top) shows the estimated hazard function; it is low in the teens, higher in the 20s, and declining in the 30s. It increases again in the 40s, but that is an artifact of the estimation process; as the number of respondents "at risk" decreases, a small number of women getting married yields a large estimated hazard. The survival function will smooth out this noise.

Estimating the Survival Function

Once we have the hazard function, we can estimate the survival function. The chance of surviving past time `t` is the chance of surviving all times up through `t`, which is the cumulative product of the complementary hazard function:

$$[1 - \lambda(0)][1 - \lambda(1)]...[1 - \lambda(t)]$$

The `HazardFunction` class provides `MakeSurvival`, which computes this product:

```
# class HazardFunction:

    def MakeSurvival(self):
        ts = self.series.index
        ss = (1 - self.series).cumprod()
        cdf = thinkstats2.Cdf(ts, 1-ss)
        sf = SurvivalFunction(cdf)
        return sf
```

`ts` is the sequence of times where the hazard function is estimated. `ss` is the cumulative product of the complementary hazard function, so it is the survival function.

Because of the way `SurvivalFunction` is implemented, we have to compute the complement of `ss`, make a Cdf, and then instantiate a SurvivalFunction object.

Figure 13-2 (bottom) shows the result. The survival curve is steepest between 25 and 35, when most women get married. Between 35 and 45, the curve is nearly flat, indicating that women who do not marry before age 35 are unlikely to get married.

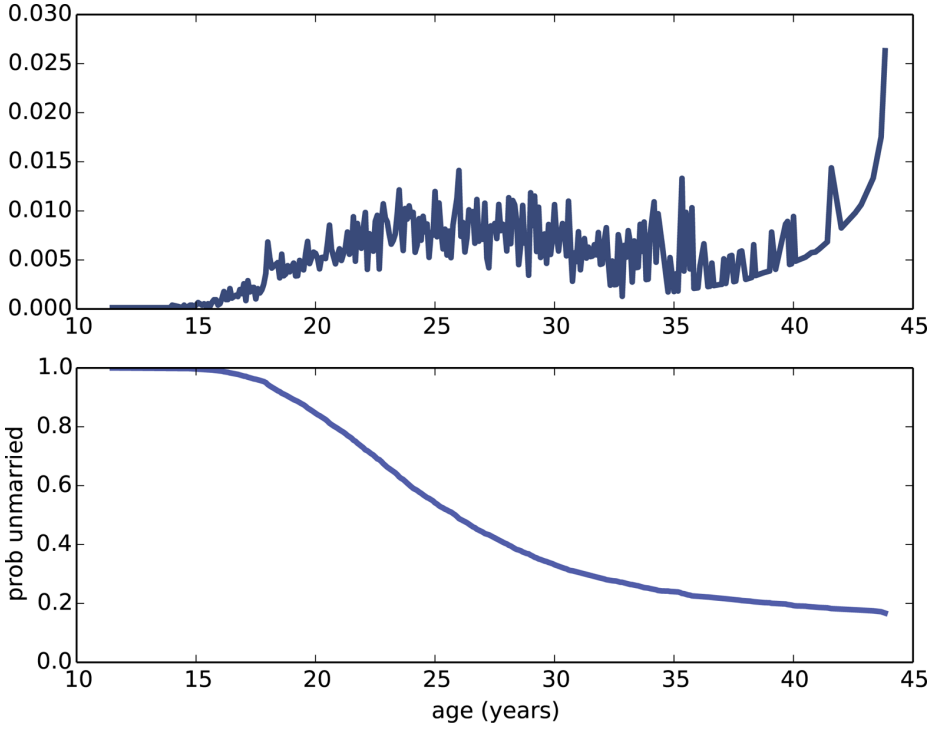


Figure 13-2. Hazard function for age at first marriage (top) and survival function (bottom)

A curve like this was the basis of a famous magazine article in 1986; *Newsweek* reported that a 40-year old unmarried woman was "more likely to be killed by a terrorist" than get married. These statistics were widely reported and became part of popular culture, but they were wrong then (because they were based on faulty analysis) and turned out to be even more wrong (because of cultural changes that were already in progress and continued). In 2006, *Newsweek* ran an another article admitting that they were wrong.

I encourage you to read more about this article, the statistics it was based on, and the reaction. It should remind you of the ethical obligation to perform statistical analysis with care, interpret the results with appropriate skepticism, and present them to the public accurately and honestly.

Confidence Intervals

Kaplan-Meier analysis yields a single estimate of the survival curve, but it is also important to quantify the uncertainty of the estimate. As usual, there are three possible sources of error: measurement error, sampling error, and modeling error.

In this example, measurement error is probably small. People generally know when they were born, whether they've been married, and when. And they can be expected to report this information accurately.

We can quantify sampling error by resampling. Here's the code:

```
def ResampleSurvival(resp, iters=101):
    low, high = resp.agemarry.min(), resp.agemarry.max()
    ts = np.arange(low, high, 1/12.0)

    ss_seq = []
    for i in range(iters):
        sample = thinkstats2.ResampleRowsWeighted(resp)
        hf, sf = EstimateSurvival(sample)
        ss_seq.append(sf.Probs(ts))

    low, high = thinkstats2.PercentileRows(ss_seq, [5, 95])
    thinkplot.FillBetween(ts, low, high)
```

`ResampleSurvival` takes `resp`, a DataFrame of respondents, and `iters`, the number of times to resample. It computes `ts`, which is the sequence of ages where we will evaluate the survival functions.

Inside the loop, `ResampleSurvival`:

- Resamples the respondents using `ResampleRowsWeighted`, which we saw in "Weighted Resampling" on page 126.
- Calls `EstimateSurvival`, which uses the process in the previous sections to estimate the hazard and survival curves, and
- Evaluates the survival curve at each age in `ts`.

`ss_seq` is a sequence of evaluated survival curves. `PercentileRows` takes this sequence and computes the 5th and 95th percentiles, returning a 90% confidence interval for the survival curve.

Figure 13-3 shows the result along with the survival function we estimated in the previous section. The confidence interval takes into account the sampling weights, unlike the estimated curve. The discrepancy between them indicates that the sampling weights have a substantial effect on the estimate—we will have to keep that in mind.

Cohort Effects

One of the challenges of survival analysis is that different parts of the estimated curve are based on different groups of respondents. The part of the curve at time `t` is based on respondents whose age was at least `t` when they were interviewed. So the leftmost part of the curve includes data from all respondents, but the rightmost part includes only the oldest respondents.

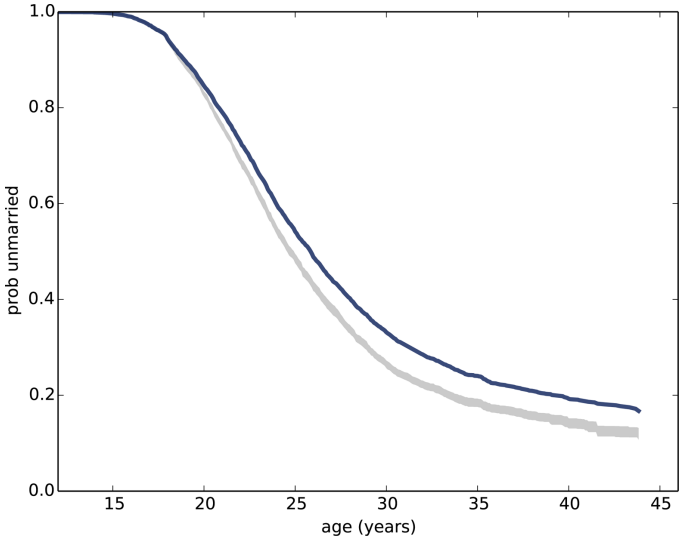


Figure 13-3. Survival function for age at first marriage and a 90% confidence interval based on weighted resampling.

If the relevant characteristics of the respondents are not changing over time, that's fine, but in this case it seems likely that marriage patterns are different for women born in different generations. We can investigate this effect by grouping respondents according to their decade of birth. Groups like this, defined by date of birth or similar events, are called **cohorts**, and differences between the groups are called **cohort effects**.

To investigate cohort effects in the NSFG marriage data, I gathered the Cycle 6 data from 2002 used throughout this book; the Cycle 7 data from 2006–2010 used in "Replication" on page 113; and the Cycle 5 data from 1995. In total these datasets include 30,769 respondents.

```
resp5 = ReadFemResp1995()
resp6 = ReadFemResp2002()
resp7 = ReadFemResp2010()
resps = [resp5, resp6, resp7]
```

For each DataFrame, `resp`, I use `cmbirth` to compute the decade of birth for each respondent:

```
month0 = pandas.to_datetime('1899-12-15')
dates = [month0 + pandas.DateOffset(months=cm)
         for cm in resp.cmbirth]
resp['decade'] = (pandas.DatetimeIndex(dates).year - 1900) // 10
```

`cmbirth` is encoded as the integer number of months since December 1899; `month0` represents that date as a Timestamp object. For each birth date, we instantiate a

`DateOffset` that contains the century-month and add it to `month0`; the result is a sequence of Timestamps, which is converted to a `DateTimeIndex`. Finally, we extract `year` and compute decades.

To take into account the sampling weights, and also to show variability due to sampling error, I resample the data, group respondents by decade, and plot survival curves:

```
for i in range(iters):
    samples = [thinkstats2.ResampleRowsWeighted(resp)
               for resp in resps]
    sample = pandas.concat(samples, ignore_index=True)
    groups = sample.groupby('decade')

    EstimateSurvivalByDecade(groups, alpha=0.2)
```

Data from the three NSFG cycles use different sampling weights, so I resample them separately and then use `concat` to merge them into a single DataFrame. The parameter `ignore_index` tells `concat` not to match up respondents by index; instead it creates a new index from 0 to 30768.

`EstimateSurvivalByDecade` plots survival curves for each cohort:

```
def EstimateSurvivalByDecade(resp):
    for name, group in groups:
        hf, sf = EstimateSurvival(group)
        thinkplot.Plot(sf)
```

Figure 13-4 shows the results.

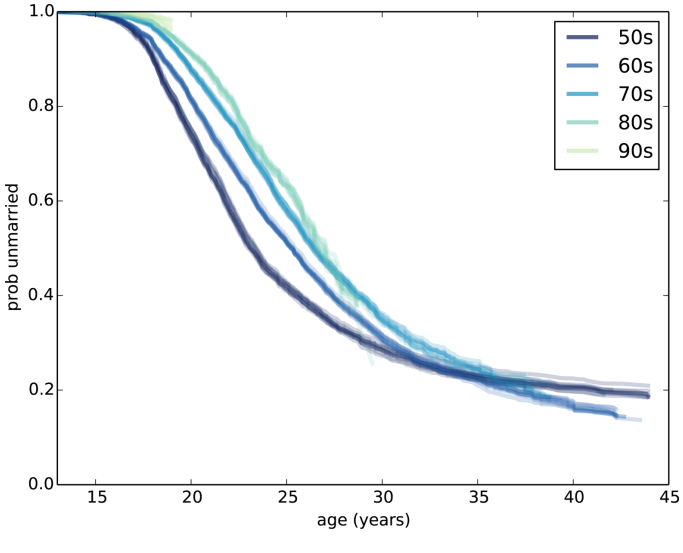


Figure 13-4. Survival functions for respondents born during different decades.

Several patterns are visible:

- Women born in the 1950s married earliest, with successive cohorts marrying later and later, at least until age 30 or so.
- Women born in the 1960s follow a surprising pattern. Prior to age 25, they were marrying at slower rates than their predecessors. After age 25, they were marrying faster. By age 32, they had overtaken the 50s cohort, and at age 44 they are substantially more likely to have married.

  Women born in the 60s turned 25 between 1985 and 1995. Remembering that the *Newsweek* article I mentioned was published in 1986, it is tempting to imagine that the article triggered a marriage boom. That explanation would be too pat, but it is possible that the article and the reaction to it were indicative of a mood that affected the behavior of this cohort.
- The pattern of the 1970s cohort is similar. They are less likely than their predecessors to be married before age 25, but at age 35 they have caught up with both of the previous cohorts.
- Women born in the 1980s are even less likely to marry before age 25. What happens after that is not clear; for more data, we have to wait for the next cycle of the NSFG.

In the meantime we can make some predictions.

Extrapolation

The survival curve for the '70s cohort ends at about age 38; for the '80s cohort it ends at age 28, and for the '90s cohort we hardly have any data at all.

We can extrapolate these curves by "borrowing" data from the previous cohort. HazardFunction provides a method, `Extend`, that copies the tail from another longer HazardFunction:

```
# class HazardFunction

    def Extend(self, other):
        last = self.series.index[-1]
        more = other.series[other.series.index > last]
        self.series = pandas.concat([self.series, more])
```

As we saw in "Hazard Function" on page 167, the HazardFunction contains a Series that maps from t to $\lambda(t)$. `Extend` finds `last`, which is the last index in `self.series`, selects values from `other` that come later than `last`, and appends them onto `self.series`.

Now we can extend the HazardFunction for each cohort, using values from the predecessor:

```
def PlotPredictionsByDecade(groups):
    hfs = []
    for name, group in groups:
        hf, sf = EstimateSurvival(group)
        hfs.append(hf)

    thinkplot.PrePlot(len(hfs))
    for i, hf in enumerate(hfs):
        if i > 0:
            hf.Extend(hfs[i-1])
        sf = hf.MakeSurvival()
        thinkplot.Plot(sf)
```

`groups` is a GroupBy object with respondents grouped by decade of birth. The first loop computes the HazardFunction for each group.

The second loop extends each HazardFunction with values from its predecessor, which might contain values from the previous group, and so on. Then it converts each HazardFunction to a SurvivalFunction and plots it.

Figure 13-5 shows the results; I've removed the 50s cohort to make the predictions more visible. These results suggest that by age 40, the most recent cohorts will converge with the 60s cohort, with fewer than 20% never married.

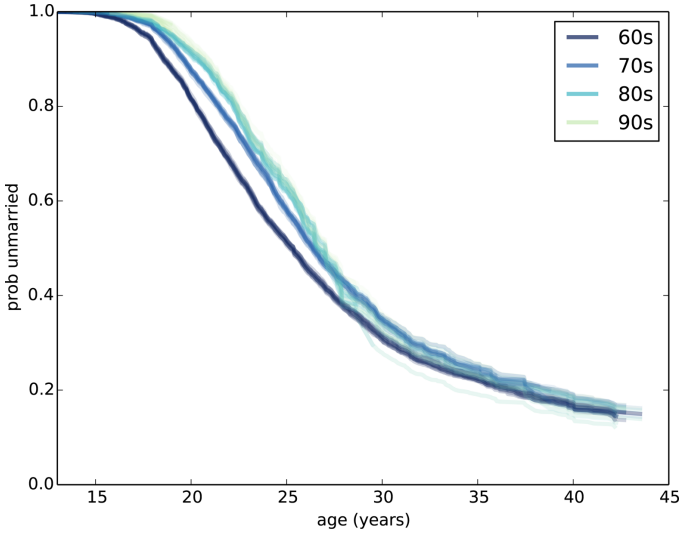


Figure 13-5. Survival functions for respondents born during different decades, with predictions for the later cohorts.

Expected Remaining Lifetime

Given a survival curve, we can compute the expected remaining lifetime as a function of current age. For example, given the survival function of pregnancy length from "Survival Curves" on page 165, we can compute the expected time until delivery.

The first step is to extract the PMF of lifetimes. `SurvivalFunction` provides a method that does that:

```
# class SurvivalFunction

    def MakePmf(self, filler=None):
        pmf = thinkstats2.Pmf()
        for val, prob in self.cdf.Items():
            pmf.Set(val, prob)

        cutoff = self.cdf.ps[-1]
        if filler is not None:
            pmf[filler] = 1-cutoff

        return pmf
```

Remember that the SurvivalFunction contains the Cdf of lifetimes. The loop copies the values and probabilities from the Cdf into a Pmf.

`cutoff` is the highest probability in the Cdf, which is 1 if the Cdf is complete, and otherwise less than 1. If the Cdf is incomplete, we plug in the provided value, `filler`, to cap it off.

The Cdf of pregnancy lengths is complete, so we don't have to worry about this detail yet.

The next step is to compute the expected remaining lifetime, where "expected" means average. `SurvivalFunction` provides a method that does that, too:

```
# class SurvivalFunction

    def RemainingLifetime(self, filler=None, func=thinkstats2.Pmf.Mean):
        pmf = self.MakePmf(filler=filler)
        d = {}
        for t in sorted(pmf.Values())[:-1]:
            pmf[t] = 0
            pmf.Normalize()
            d[t] = func(pmf) - t

        return pandas.Series(d)
```

`RemainingLifetime` takes `filler`, which is passed along to `MakePmf`, and `func` which is the function used to summarize the distribution of remaining lifetimes.

`pmf` is the Pmf of lifetimes extracted from the SurvivalFunction. `d` is a dictionary that contains the results, a map from current age, `t`, to expected remaining lifetime.

The loop iterates through the values in the Pmf. For each value of `t` it computes the conditional distribution of lifetimes, given that the lifetime exceeds `t`. It does that by removing values from the Pmf one at a time and renormalizing the remaining values.

Then it uses `func` to summarize the conditional distribution. In this example the result is the mean pregnancy length, given that the length exceeds `t`. By subtracting `t` we get the mean remaining pregnancy length.

Figure 13-6 (left) shows the expected remaining pregnancy length as a function of the current duration. For example, during Week 0, the expected remaining duration is about 34 weeks. That's less than full term (39 weeks) because terminations of pregnancy in the first trimester bring the average down.

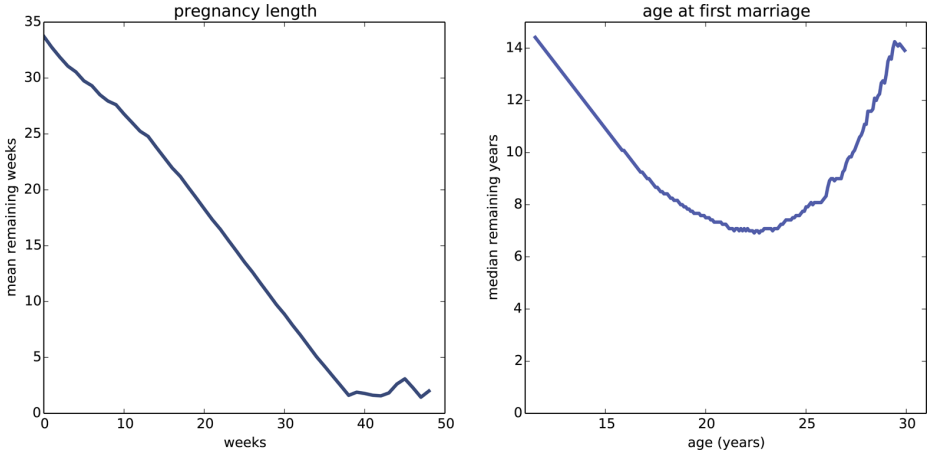


Figure 13-6. Expected remaining lifetime for pregnancy length (left) and years until first marriage (right)

The curve drops slowly during the first trimester. After 13 weeks, the expected remaining lifetime has dropped by only 9 weeks, to 25. After that the curve drops faster, by about a week per week.

Between Week 37 and 42, the curve levels off between 1 and 2 weeks. At any time during this period, the expected remaining lifetime is the same; with each week that passes, the destination gets no closer. Processes with this property are called *memoryless* because the past has no effect on the predictions. This behavior is the mathematical basis of the infuriating mantra of obstetrics nurses: "any day now."

Figure 13-6 (right) shows the median remaining time until first marriage, as a function of age. For an 11 year-old girl, the median time until first marriage is about 14 years. The curve decreases until age 22 when the median remaining time is about 7 years. After that it increases again: by age 30 it is back where it started, at 14 years.

Based on this data, young women have decreasing remaining "lifetimes". Mechanical components with this property are called **NBUE** for "new better than used in expectation," meaning that a new part is expected to last longer.

Women older than 22 have increasing remaining time until first marriage. Components with this property are called **UBNE** for "used better than new in expectation." That is, the older the part, the longer it is expected to last. Newborns and cancer patients are also UBNE; their life expectancy increases the longer they live.

For this example I computed median, rather than mean, because the Cdf is incomplete; the survival curve projects that about 20% of respondents will not marry before age 44. The age of first marriage for these women is unknown, and might be non-existent, so we can't compute a mean.

I deal with these unknown values by replacing them with `np.inf`, a special value that represents infinity. That makes the mean infinity for all ages, but the median is well-defined as long as more than 50% of the remaining lifetimes are finite, which is true until age 30. After that it is hard to define a meaningful expected remaining lifetime.

Here's the code that computes and plots these functions:

```
rem_life1 = sf1.RemainingLifetime()
thinkplot.Plot(rem_life1)

func = lambda pmf: pmf.Percentile(50)
rem_life2 = sf2.RemainingLifetime(filler=np.inf, func=func)
thinkplot.Plot(rem_life2)
```

`sf1` is the survival function for pregnancy length; in this case we can use the default values for `RemainingLifetime`.

`sf2` is the survival function for age at first marriage; `func` is a function that takes a Pmf and computes its median (50th percentile).

Exercises

My solution to this exercise is in `chap13soln.py`.

Exercise 13-1.

In NSFG Cycles 6 and 7, the variable `cmdivorcx` contains the date of divorce for the respondent's first marriage, if applicable, encoded in century-months.

Compute the duration of marriages that have ended in divorce, and the duration, so far, of marriages that are ongoing. Estimate the hazard and survival function for the duration of marriage.

Use resampling to take into account sampling weights, and plot data from several resamples to visualize sampling error.

Consider dividing the respondents into groups by decade of birth, and possibly by age at first marriage.

Glossary

survival analysis
: A set of methods for describing and predicting lifetimes, or more generally time until an event occurs.

survival curve
: A function that maps from a time, *t*, to the probability of surviving past *t*.

hazard function
: A function that maps from *t* to the fraction of people alive until *t* who die at *t*.

Kaplan-Meier estimation
: An algorithm for estimating hazard and survival functions.

cohort
: a group of subjects defined by an event, like date of birth, in a particular interval of time.

cohort effect
: a difference between cohorts.

NBUE
: A property of expected remaining lifetime, "New better than used in expectation."

UBNE
: A property of expected remaining lifetime, "Used better than new in expectation."

CHAPTER 14

Analytic Methods

This book has focused on computational methods like simulation and resampling, but some of the problems we solved have analytic solutions that can be much faster.

I present some of these methods in this chapter, and explain how they work. At the end of the chapter, I make suggestions for integrating computational and analytic methods for exploratory data analysis.

The code in this chapter is in `normal.py`. For information about downloading and working with this code, see "Using the Code" on page xi.

Normal Distributions

As a motivating example, let's review the problem from "Sampling Distributions" on page 94:

> Suppose you are a scientist studying gorillas in a wildlife preserve. Having weighed 9 gorillas, you find sample mean $\bar{x} = 90$ kg and sample standard deviation, $S = 7.5$ kg. If you use $\bar{x}$ to estimate the population mean, what is the standard error of the estimate?

To answer that question, we need the sampling distribution of $\bar{x}$. In "Sampling Distributions" on page 94 we approximated this distribution by simulating the experiment (weighing 9 gorillas), computing $\bar{x}$ for each simulated experiment, and accumulating the distribution of estimates.

The result is an approximation of the sampling distribution. Then we use the sampling distribution to compute standard errors and confidence intervals:

- The standard deviation of the sampling distribution is the standard error of the estimate; in the example, it is about 2.5 kg.
- The interval between the 5th and 95th percentile of the sampling distribution is a 90% confidence interval. If we run the experiment many times, we expect the esti-

mate to fall in this interval 90% of the time. In the example, the 90% CI is (86, 94) kg.

Now we'll do the same calculation analytically. We take advantage of the fact that the weights of adult female gorillas are roughly normally distributed. Normal distributions have two properties that make them amenable for analysis: they are "closed" under linear transformation and addition. To explain what that means, I need some notation.

If the distribution of a quantity, X, is normal with parameters μ and σ, you can write

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

where the symbol $\sim$ means "is distributed" and the script letter $\mathcal{N}$ stands for "normal."

A linear transformation of X is something like $X' = aX + b$, where a and b are real numbers.A family of distributions is closed under linear transformation if X' is in the same family as X. The normal distribution has this property; if $X \sim \mathcal{N}(\mu, \sigma^2)$,

$$X' \sim \mathcal{N}(a\mu + b, a^2\sigma^2) \qquad (1)$$

Normal distributions are also closed under addition. If $Z = X + Y$ and $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$ and $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$ then

$$Z \sim \mathcal{N}(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2) \qquad (2)$$

In the special case $Z = X + X$, we have

$$Z \sim \mathcal{N}(2\mu_X, 2\sigma_X^2)$$

and in general if we draw n values of X and add them up, we have

$$Z \sim \mathcal{N}(n\mu_X, n\sigma_X^2) \qquad (3)$$

Sampling Distributions

Now we have everything we need to compute the sampling distribution of $\bar{x}$. Remember that we compute $\bar{x}$ by weighing n gorillas, adding up the total weight, and dividing by n.

Assume that the distribution of gorilla weights, X, is approximately normal:

$$X \sim \mathcal{N}(\mu, \sigma^2)$$

If we weigh n gorillas, the total weight, Y, is distributed

$$Y \sim \mathcal{N}(n\mu, n\sigma^2)$$

using Equation 3. And if we divide by n, the sample mean, Z, is distributed

$$Z \sim \mathcal{N}(\mu, \sigma^2 / n)$$

using Equation 1 with $a = 1 / n$.

The distribution of Z is the sampling distribution of $\bar{x}$. The mean of Z is μ, which shows that $\bar{x}$ is an unbiased estimate of μ. The variance of the sampling distribution is σ^2 / n.

So the standard deviation of the sampling distribution, which is the standard error of the estimate, is $\sigma / \sqrt{n}$. In the example, σ is 7.5 kg and n is 9, so the standard error is 2.5 kg. That result is consistent with what we estimated by simulation, but much faster to compute!

We can also use the sampling distribution to compute confidence intervals. A 90% confidence interval for $\bar{x}$ is the interval between the 5th and 95th percentiles of Z. Since Z is normally distributed, we can compute percentiles by evaluating the inverse CDF.

There is no closed form for the CDF of the normal distribution or its inverse, but there are fast numerical methods and they are implemented in SciPy, as we saw in "The Normal Distribution" on page 52. `thinkstats2` provides a wrapper function that makes the SciPy function a little easier to use:

```
def EvalNormalCdfInverse(p, mu=0, sigma=1):
    return scipy.stats.norm.ppf(p, loc=mu, scale=sigma)
```

Given a probability, `p`, it returns the corresponding percentile from a normal distribution with parameters `mu` and `sigma`. For the 90% confidence interval of $\bar{x}$, we compute the 5th and 95th percentiles like this:

```
>>> thinkstats2.EvalNormalCdfInverse(0.05, mu=90, sigma=2.5)
85.888

>>> thinkstats2.EvalNormalCdfInverse(0.95, mu=90, sigma=2.5)
94.112
```

So if we run the experiment many times, we expect the estimate, $\bar{x}$, to fall in the range (85.9, 94.1) about 90% of the time. Again, this is consistent with the result we got by simulation.

Representing Normal Distributions

To make these calculations easier, I have defined a class called `Normal` that represents a normal distribution and encodes the equations in the previous sections. Here's what it looks like:

```
class Normal(object):

    def __init__(self, mu, sigma2):
        self.mu = mu
        self.sigma2 = sigma2

    def __str__(self):
        return 'N(%g, %g)' % (self.mu, self.sigma2)
```

So we can instantiate a Normal that represents the distribution of gorilla weights:

```
>>> dist = Normal(90, 7.5**2)
>>> dist
N(90, 56.25)
```

`Normal` provides `Sum`, which takes a sample size, `n`, and returns the distribution of the sum of `n` values, using Equation 3:

```
    def Sum(self, n):
        return Normal(n * self.mu, n * self.sigma2)
```

Normal also knows how to multiply and divide using Equation 1:

```
    def __mul__(self, factor):
        return Normal(factor * self.mu, factor**2 * self.sigma2)

    def __div__(self, divisor):
        return 1 / divisor * self
```

So we can compute the sampling distribution of the mean with sample size 9:

```
>>> dist_xbar = dist.Sum(9) / 9
>>> dist_xbar.sigma
2.5
```

The standard deviation of the sampling distribution is 2.5 kg, as we saw in the previous section. Finally, Normal provides `Percentile`, which we can use to compute a confidence interval:

```
>>> dist_xbar.Percentile(5), dist_xbar.Percentile(95)
85.888 94.113
```

And that's the same answer we got before. We'll use the Normal class again later, but before we go on, we need one more bit of analysis.

Central Limit Theorem

As we saw in the previous sections, if we add values drawn from normal distributions, the distribution of the sum is normal. Most other distributions don't have this property; if we add values drawn from other distributions, the sum does not generally have an analytic distribution.

But if we add up `n` values from almost any distribution, the distribution of the sum converges to normal as `n` increases.

More specifically, if the distribution of the values has mean and standard deviation μ and σ, the distribution of the sum is approximately $\mathcal{N}(n\mu, n\sigma^2)$.

This result is the Central Limit Theorem (CLT). It is one of the most useful tools for statistical analysis, but it comes with caveats:

- The values have to be drawn independently. If they are correlated, the CLT doesn't apply (although this is seldom a problem in practice).

- The values have to come from the same distribution (although this requirement can be relaxed).
- The values have to be drawn from a distribution with finite mean and variance. So most Pareto distributions are out.
- The rate of convergence depends on the skewness of the distribution. Sums from an exponential distribution converge for small `n`. Sums from a lognormal distribution require larger sizes.

The Central Limit Theorem explains the prevalence of normal distributions in the natural world. Many characteristics of living things are affected by genetic and environmental factors whose effect is additive. The characteristics we measure are the sum of a large number of small effects, so their distribution tends to be normal.

Testing the CLT

To see how the Central Limit Theorem works, and when it doesn't, let's try some experiments. First, we'll try an exponential distribution:

```
def MakeExpoSamples(beta=2.0, iters=1000):
    samples = []
    for n in [1, 10, 100]:
        sample = [np.sum(np.random.exponential(beta, n))
                  for _ in range(iters)]
        samples.append((n, sample))
    return samples
```

`MakeExpoSamples` generates samples of sums of exponential values (I use "exponential values" as shorthand for "values from an exponential distribution"). `beta` is the parameter of the distribution; `iters` is the number of sums to generate.

To explain this function, I'll start from the inside and work my way out. Each time we call `np.random.exponential`, we get a sequence of `n` exponential values and compute its sum. `sample` is a list of these sums, with length `iters`.

It is easy to get `n` and `iters` confused: `n` is the number of terms in each sum; `iters` is the number of sums we compute in order to characterize the distribution of sums.

The return value is a list of `(n, sample)` pairs. For each pair, we make a normal probability plot:

```
def NormalPlotSamples(samples, plot=1, ylabel=''):
    for n, sample in samples:
        thinkplot.SubPlot(plot)
        thinkstats2.NormalProbabilityPlot(sample)

        thinkplot.Config(title='n=%d' % n, ylabel=ylabel)
        plot += 1
```

`NormalPlotSamples` takes the list of pairs from `MakeExpoSamples` and generates a row of normal probability plots.

Figure 14-1 (top row) shows the results. With `n=1`, the distribution of the sum is still exponential, so the normal probability plot is not a straight line. But with `n=10` the distribution of the sum is approximately normal, and with `n=100` it is all but indistinguishable from normal.

Figure 14-1 (bottom row) shows similar results for a lognormal distribution. Lognormal distributions are generally more skewed than exponential distributions, so the distribution of sums takes longer to converge. With `n=10` the normal probability plot is nowhere near straight, but with `n=100` it is approximately normal.

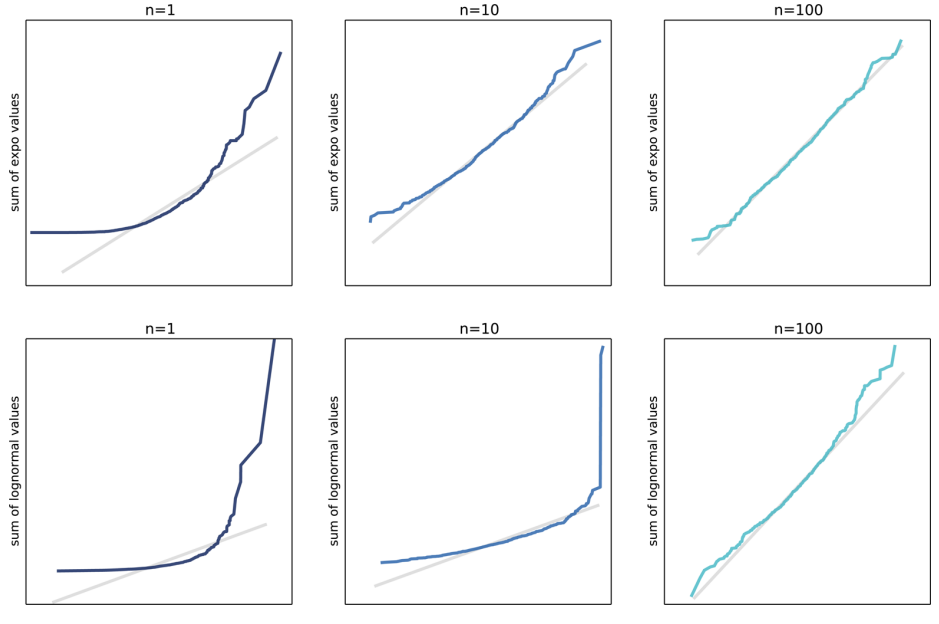


Figure 14-1. Distributions of sums of exponential values (top row) and lognormal values (bottom row)

Pareto distributions are even more skewed than lognormal. Depending on the parameters, many Pareto distributions do not have finite mean and variance. As a result, the Central Limit Theorem does not apply. Figure 14-2 (top row) shows distributions of sums of Pareto values. Even with `n=100` the normal probability plot is far from straight.

I also mentioned that CLT does not apply if the values are correlated. To test that, I generate correlated values from an exponential distribution. The algorithm for generating correlated values is (1) generate correlated normal values, (2) use the normal CDF

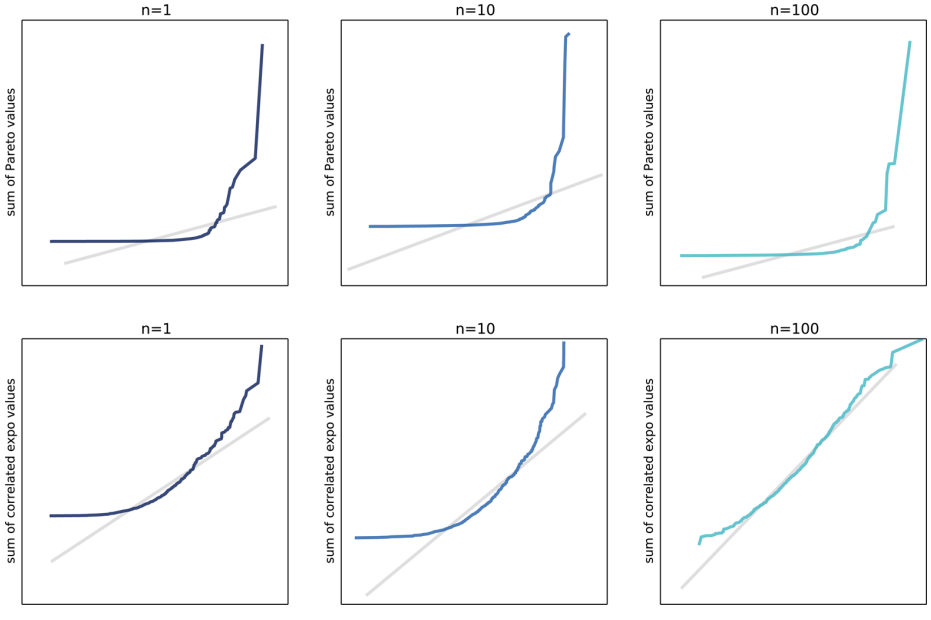


Figure 14-2. Distributions of sums of Pareto values (top row) and correlated exponential values (bottom row)

to transform the values to uniform, and (3) use the inverse exponential CDF to transform the uniform values to exponential.

`GenerateCorrelated` returns an iterator of `n` normal values with serial correlation `rho`:

```
def GenerateCorrelated(rho, n):
    x = random.gauss(0, 1)
    yield x

    sigma = math.sqrt(1 - rho**2)
    for _ in range(n-1):
        x = random.gauss(x*rho, sigma)
        yield x
```

The first value is a standard normal value. Each subsequent value depends on its predecessor: if the previous value is `x`, the mean of the next value is `x*rho`, with variance `1-rho**2`. Note that `random.gauss` takes the standard deviation as the second argument, not variance.

`GenerateExpoCorrelated` takes the resulting sequence and transforms it to exponential:

```
def GenerateExpoCorrelated(rho, n):
    normal = list(GenerateCorrelated(rho, n))
    uniform = scipy.stats.norm.cdf(normal)
    expo = scipy.stats.expon.ppf(uniform)
    return expo
```

`normal` is a list of correlated normal values. `uniform` is a sequence of uniform values between 0 and 1. `expo` is a correlated sequence of exponential values. `ppf` stands for "percent point function," which is another name for the inverse CDF.

Figure 14-2 (bottom row) shows distributions of sums of correlated exponential values with `rho=0.9`. The correlation slows the rate of convergence; nevertheless, with `n=100` the normal probability plot is nearly straight. So even though CLT does not strictly apply when the values are correlated, moderate correlations are seldom a problem in practice.

These experiments are meant to show how the Central Limit Theorem works, and what happens when it doesn't. Now let's see how we can use it.

Applying the CLT

To see why the Central Limit Theorem is useful, let's get back to the example in "Testing a Difference in Means" on page 104: testing the apparent difference in mean pregnancy length for first babies and others. As we've seen, the apparent difference is about 0.078 weeks:

```
>>> live, firsts, others = first.MakeFrames()
>>> delta = firsts.prglngth.mean() - others.prglngth.mean()
0.078
```

Remember the logic of hypothesis testing: we compute a p-value, which is the probability of the observed difference under the null hypothesis; if it is small, we conclude that the observed difference is unlikely to be due to chance.

In this example, the null hypothesis is that the distribution of pregnancy lengths is the same for first babies and others. So we can compute the sampling distribution of the mean like this:

```
    dist1 = SamplingDistMean(live.prglngth, len(firsts))
    dist2 = SamplingDistMean(live.prglngth, len(others))
```

Both sampling distributions are based on the same population, which is the pool of all live births. `SamplingDistMean` takes this sequence of values and the sample size, and returns a Normal object representing the sampling distribution:

```
def SamplingDistMean(data, n):
    mean, var = data.mean(), data.var()
    dist = Normal(mean, var)
    return dist.Sum(n) / n
```

`mean` and `var` are the mean and variance of `data`. We approximate the distribution of the data with a normal distribution, `dist`.

In this example, the data are not normally distributed, so this approximation is not very good. But then we compute `dist.Sum(n) / n`, which is the sampling distribution of the mean of `n` values. Even if the data are not normally distributed, the sampling distribution of the mean is, by the Central Limit Theorem.

Next, we compute the sampling distribution of the difference in the means. The `Normal` class knows how to perform subtraction using Equation 2:

```
    def __sub__(self, other):
        return Normal(self.mu - other.mu,
                      self.sigma2 + other.sigma2)
```

So we can compute the sampling distribution of the difference like this:

```
>>> dist = dist1 - dist2
N(0, 0.0032)
```

The mean is 0, which makes sense because we expect two samples from the same distribution to have the same mean, on average. The variance of the sampling distribution is 0.0032.

`Normal` provides `Prob`, which evaluates the normal CDF. We can use `Prob` to compute the probability of a difference as large as `delta` under the null hypothesis:

```
>>> 1 - dist.Prob(delta)
0.084
```

Which means that the p-value for a one-sided test is 0.84. For a two-sided test we would also compute

```
>>> dist.Prob(-delta)
0.084
```

Which is the same because the normal distribution is symmetric. The sum of the tails is 0.168, which is consistent with the estimate in "Testing a Difference in Means" on page 104, which was 0.17.

Correlation Test

In "Testing a Correlation" on page 107 we used a permutation test for the correlation between birth weight and mother's age, and found that it is statistically significant, with p-value less than 0.001.

Now we can do the same thing analytically. The method is based on this mathematical result: given two variables that are normally distributed and uncorrelated, if we generate a sample with size n, compute Pearson's correlation, r, and then compute the transformed correlation:

$$t = r\sqrt{\frac{n-2}{1-r^2}}$$

the distribution of t is Student's t-distribution with parameter $n - 2$. The t-distribution is an analytic distribution; the CDF can be computed efficiently using gamma functions.

We can use this result to compute the sampling distribution of correlation under the null hypothesis; that is, if we generate uncorrelated sequences of normal values, what is the distribution of their correlation? `StudentCdf` takes the sample size, `n`, and returns the sampling distribution of correlation:

```
def StudentCdf(n):
    ts = np.linspace(-3, 3, 101)
    ps = scipy.stats.t.cdf(ts, df=n-2)
    rs = ts / np.sqrt(n - 2 + ts**2)
    return thinkstats2.Cdf(rs, ps)
```

`ts` is a NumPy array of values for t, the transformed correlation. `ps` contains the corresponding probabilities, computed using the CDF of the Student's t-distribution implemented in SciPy. The parameter of the t-distribution, `df`, stands for "degrees of freedom." I won't explain that term, but you can read about it at this Wikipedia page.

To get from `ts` to the correlation coefficients, `rs`, we apply the inverse transform,

$$r = t / \sqrt{n - 2 + t^2}$$

The result is the sampling distribution of r under the null hypothesis. Figure 14-3 shows this distribution along with the distribution we generated in "Testing a Correlation" on page 107 by resampling. They are nearly identical. Although the actual distributions are not normal, Pearson's coefficient of correlation is based on sample means and variances. By the Central Limit Theorem, these moment-based statistics are normally distributed even if the data are not.

From Figure 14-3, we can see that the observed correlation, 0.07, is unlikely to occur if the variables are actually uncorrelated. Using the analytic distribution, we can compute just how unlikely:

```
t = r * math.sqrt((n-2) / (1-r))
p_value = 1 - scipy.stats.t.cdf(t, df=n-2)
```

We compute the value of `t` that corresponds to `r=0.07`, and then evaluate the t-distribution at `t`. The result is `6.4e-12`. This example demonstrates an advantage of the analytic method: we can compute very small p-values. But in practice it usually doesn't matter.

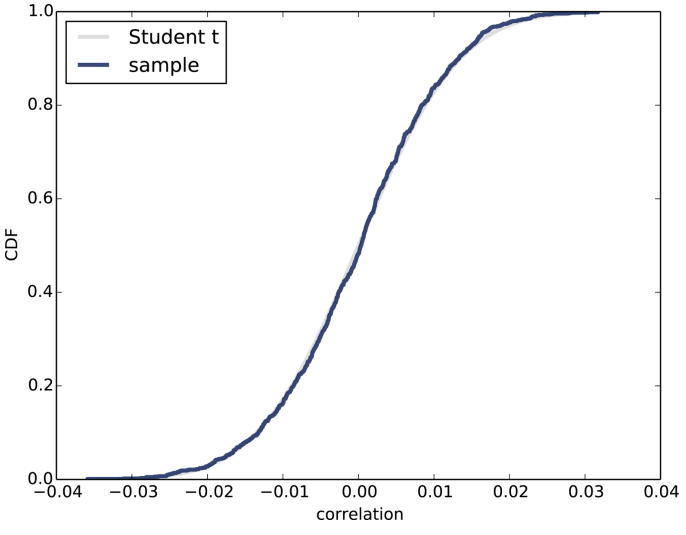


Figure 14-3. Sampling distribution of correlations for uncorrelated normal variables

Chi-Squared Test

In "Chi-Squared Tests" on page 109 we used the chi-squared statistic to test whether a die is crooked. The chi-squared statistic measures the total normalized deviation from the expected values in a table:

$$\chi^2 = \sum_i \frac{(O_i - E_i)^2}{E_i}$$

One reason the chi-squared statistic is widely used is that its sampling distribution under the null hypothesis is analytic; by a remarkable coincidence,[1] it is called the chi-squared distribution. Like the t-distribution, the chi-squared CDF can be computed efficiently using gamma functions.

SciPy provides an implementation of the chi-squared distribution, which we use to compute the sampling distribution of the chi-squared statistic:

```
def ChiSquaredCdf(n):
    xs = np.linspace(0, 25, 101)
    ps = scipy.stats.chi2.cdf(xs, df=n-1)
    return thinkstats2.Cdf(xs, ps)
```

Figure 14-4 shows the analytic result along with the distribution we got by resampling. They are very similar, especially in the tail, which is the part we usually care most about.

1. Not really.

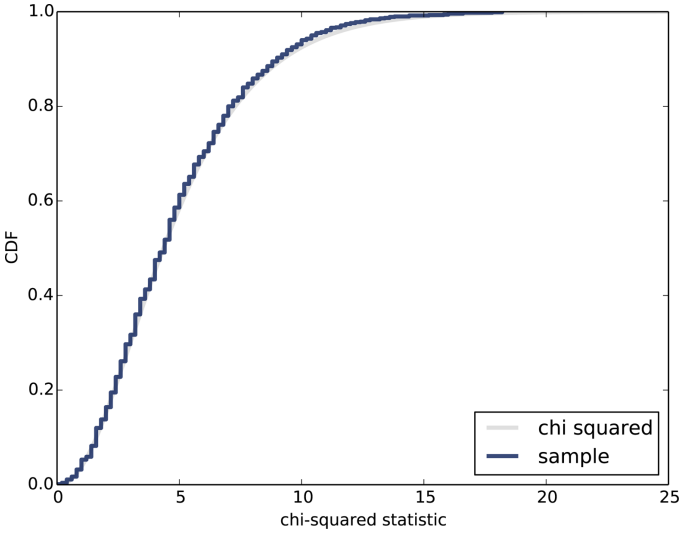


Figure 14-4. Sampling distribution of chi-squared statistics for a fair six-sided die.

We can use this distribution to compute the p-value of the observed test statistic, `chi2`:

```
p_value = 1 - scipy.stats.chi2.cdf(chi2, df=n-1)
```

The result is 0.041, which is consistent with the result from "Chi-Squared Tests" on page 109.

The parameter of the chi-squared distribution is "degrees of freedom" again. In this case the correct parameter is `n-1`, where `n` is the size of the table, 6. Choosing this parameter can be tricky; to be honest, I am never confident that I have it right until I generate something like Figure 14-4 to compare the analytic results to the resampling results.

Discussion

This book focuses on computational methods like resampling and permutation. These methods have several advantages over analysis:

- They are easier to explain and understand. For example, one of the most difficult topics in an introductory statistics class is hypothesis testing. Many students don't really understand what p-values are. I think the approach I presented in Chapter 9—simulating the null hypothesis and computing test statistics—makes the fundamental idea clearer.

- They are robust and versatile. Analytic methods are often based on assumptions that might not hold in practice. Computational methods require fewer assumptions, and can be adapted and extended more easily.
- They are debuggable. Analytic methods are often like a black box: you plug in numbers and they spit out results. But it's easy to make subtle errors, hard to be confident that the results are right, and hard to find the problem if they are not. Computational methods lend themselves to incremental development and testing, which fosters confidence in the results.

But there is one drawback: computational methods can be slow. Taking into account these pros and cons, I recommend the following process:

1. Use computational methods during exploration. If you find a satisfactory answer and the runtime is acceptable, you can stop.
2. If run time is not acceptable, look for opportunities to optimize. Using analytic methods is one of several methods of optimization.
3. If replacing a computational method with an analytic method is appropriate, use the computational method as a basis of comparison, providing mutual validation between the computational and analytic results.

For the vast majority of problems I have worked on, I didn't have to go past Step 1.

Exercises

A solution to these exercises is in `chap14soln.py`

Exercise 14-1.

In "The lognormal Distribution" on page 55, we saw that the distribution of adult weights is approximately lognormal. One possible explanation is that the weight a person gains each year is proportional to their current weight. In that case, adult weight is the product of a large number of multiplicative factors:

$$w = w_0 f_1 f_2 \ldots f_n$$

where w is adult weight, w_0 is birth weight, and f_i is the weight gain factor for year i.

The log of a product is the sum of the logs of the factors:

$$\log w = \log w_0 + \log f_1 + \log f_2 + \ldots + \log f_n$$

So by the Central Limit Theorem, the distribution of $\log w$ is approximately normal for large n, which implies that the distribution of w is lognormal.

To model this phenomenon, choose a distribution for f that seems reasonable, then generate a sample of adult weights by choosing a random value from the distribution

of birth weights, choosing a sequence of factors from the distribution of f, and computing the product. What value of n is needed to converge to a lognormal distribution?

Exercise 14-2.

In "Applying the CLT" on page 190 we used the Central Limit Theorem to find the sampling distribution of the difference in means, δ, under the null hypothesis that both samples are drawn from the same population.

We can also use this distribution to find the standard error of the estimate and confidence intervals, but that would only be approximately correct. To be more precise, we should compute the sampling distribution of δ under the alternate hypothesis that the samples are drawn from different populations.

Compute this distribution and use it to calculate the standard error and a 90% confidence interval for the difference in means.

Exercise 14-3.

In a recent paper[2], Stein et al. investigate the effects of an intervention intended to mitigate gender-stereotypical task allocation within student engineering teams.

Before and after the intervention, students responded to a survey that asked them to rate their contribution to each aspect of class projects on a 7-point scale.

Before the intervention, male students reported higher scores for the programming aspect of the project than female students; on average men reported a score of 3.57 with standard error 0.28. Women reported 1.91, on average, with standard error 0.32.

Compute the sampling distribution of the gender gap (the difference in means), and test whether it is statistically significant. Because you are given standard errors for the estimated means, you don't need to know the sample size to figure out the sampling distributions.

After the intervention, the gender gap was smaller: the average score for men was 3.44 (SE 0.16); the average score for women was 3.18 (SE 0.16). Again, compute the sampling distribution of the gender gap and test it.

Finally, estimate the change in gender gap; what is the sampling distribution of this change, and is it statistically significant?

2. "Evidence for the persistent effects of an intervention to mitigate gender-sterotypical task allocation within student engineering teams," Proceedings of the IEEE Frontiers in Education Conference, 2014.

Index

A

abstractions, 61
accuracy (models), 141
acf function, 155
Adams, Cecil, 25
addition, closed under, 184
adult weight, 56–57, 79–80, 97, 195
age, 118, 130–131, 134, 140, 169, 171
age groups, 47
aggregate method, 147
alpha parameter, 81
Anaconda distribution, xii, 130
analytic distributions, 49, 63, 192
analytic methods, 183
anecdotal evidence, 1, 12
apparent effect, 101
artifacts, 82
Australia, baby birth times, 50
autocorrelation function, 155–156, 163
average, 22

B

bar plots, 28
Bayesian inference, 101
Bayesian statistics, 96
Behavioral Risk Factor Surveillance System (BRFSS), x, 57, 79, 87, 128
betting pool, 134, 137, 142
bias
  confirmation, 2
  observer, 32, 35–37
  sampling, 97, 100, 121
  selection, 2, 37
  skewness and, 73
biased coin, 103
biased estimators, 93–95, 99–100, 185
binary integers, 140
binary searches, 71–72
binning data, 39, 69, 82
birth times, 50
birth weight
  about, 195
  comparing CDFs for, 44
  computing distribution of, 47
  correlation test and, 191
  hypothesis testing, 105–107, 113–114
  least squares fit, 118–119
  limitation of PMFs, 39
  normal distribution, 52
  normal probability plot, 54–57
  predictions with linear model, 122–124
  random samples, 45
  regression analysis, 129, 131–134, 136–137
  variables for, 6, 17
  weighted resampling, 127
bisect module, 72
Blue Man Group, 62

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

Boolean type, 9, 17, 79, 132, 136–137, 140, 156, 166
bracket operator, 16–17, 28, 43
BRFSS (Behavioral Risk Factor Surveillance System), x, 57, 79, 87, 128
Bureau of Labor Statistics, 76

C

cannabis, 145
casinos, 108
categorical variables, 132, 136–137, 141, 143–144
causal relationships, 88
causation, 88
CCDF (complementary CDF), 51, 59, 63, 165, 171
Cdf class
  about, 42
  computing percentile-based statistics, 44–46
  distribution framework, 69
  implementing, 71–72
  random numbers and, 45
  survival function and, 166, 178
  weighted resampling, 127
CDFs (cumulative distribution functions)
  about, 41–42, 48
  characterizing relationships, 82
  comparing, 44
  comparing percentile ranks, 47
  complementary, 51, 59, 63, 165, 171
  derivatives of, 65
  difference in pregnancy length, 105
  distribution framework, 69
  exponential distributions, 49–52
  inverse, 43, 46, 48, 60, 71, 185, 189–190
  limits of PMFs, 39
  lognormal distribution, 56
  normal distribution, 52
  Pareto distribution, 57–60
  percentile-based statistics, 44
  percentiles, 40
  random numbers, 45
  representing, 42
  simulating null hypothesis, 111
Central Limit Theorem (CLT), x, 186–192, 195
central moments, 72, 77
central tendency, 22, 26, 44, 73
century-months, 170, 175
chi-squared distribution, 193
chi-squared statistic, 109, 111
chi-squared test, 109, 115, 193
city size, 58
class size paradox, 30–33
cleaning data, ix, 7, 13, 135, 145–147
clinically significant effect, 24, 26
cloning repositories, xi
closed form, 52, 139, 185
CLT (Central Limit Theorem), x, 186–192, 195
coefficient of determination, 122–124, 128, 130, 132–137, 140
cohort, 181
cohort effects, 174–176, 181
cohorts, 174
complementary CDF, 51, 59, 63, 165, 171
compression (data), 61
computational methods, ix, 118, 183, 194
confidence interval, 96, 97–100, 121, 127, 158–160, 172–173, 183, 185, 196
confirmation bias, 2
confirmatory results, 113
continuous distribution, 69
contradiction, proof by, 102
contributor list, xiii
control groups, 88–89
control variables, ix, 134, 137, 143
controlled trials, 88–89
correct positive rate, 112
correlation
  about, 83, 89
  causation and, 88
  computing, ix
  Pearson's, 83, 85–87, 89, 107
  residuals and, 118
  serial, 153–156, 162
  Spearman's rank correlation, 83, 87–89, 107
  testing, 107
cost functions, 118
covariance, 84, 89
crooked die, 108
cross-sectional studies, 3, 12
cumulative distribution functions (see CDFs)
cumulative probability, 48, 69, 71
cumulative product, 171
Current Population Survey, 76
cycles, 3

D

data cleaning, ix, 7, 13, 135, 145–147

data collection, 2
data mining, 134–136, 142, 144
DataFrame data structure
  about, 4–6
  autocorrelation functions and, 156
  cohort effects and, 175
  comparing values, 20
  counting values, 8
  difference in pregnancy length, 105
  estimating statistics, 76
  exponential distribution and, 50
  extracting height and weight, 79
  groupby method, 82
  importing and cleaning data, 145–147
  index column, 5, 34–35, 37
  interpreting values, 10–11
  join operation and, 134, 144
  linear regression and, 148–151
  logistic regression and, 140, 142
  NSFG variables, 17
  OLS object and, 130
  plotting for time series, 147
  query method, 166
  reindexing, 151
  resampling and, 121
  sample size and, 114
  survival function and, 173
  weighted resampling, 127
date of birth, 142, 170
datetime64 object, 145–147
DateTimeIndex, 175
debugging, x, 195
deciles, 45
degrees of freedom, 94, 192, 194
density, 65, 68
dependent variables, 129–131, 133, 137–138, 140, 143, 150
derivatives, 65
descriptive statistics, 2
dice, 93, 108
dictionaries, 15, 70
DictWrapper class, 70–71
digitize function, 82
discrete distribution, 69
discretizing, 69, 77
distribution framework, 69
distributions
  about, 15, 25
  analytic, 49, 63, 192
  chi-squared, 193
  comparing, 44
  continuous, 69
  discrete, 69
  effect size in, 23
  empirical, 49, 62–63
  estimation game, 91–93
  exploratory process, ix
  exponential, 49–52, 60, 62, 65, 69, 98–99, 187
  first babies, 20–21
  lognormal, 56–57, 60, 63, 87, 187–188, 195
  normal, 17, 26, 52–57, 62, 65, 69, 83, 87, 91, 93, 183
  NSFG variables, 17
  outliers in, 19–20
  Pareto, 57–60, 61–63, 187–189
  plotting histograms, 16
  reporting results, 24
  representing histograms, 16
  sampling, 94–97, 99–100, 121, 121, 125–127, 183–186, 190–192
  Student's t, 192
  summarizing, 22
  symmetric, 74, 125, 191
  uniform, 17, 26, 48, 60, 80, 84, 156
  variance in, 23
  Weibull, 62
dot product, 84
dropna function, 74–75, 82, 108, 119, 125, 153

E

effect size, 24, 26, 105, 112, 114
empirical distributions, 49, 62–63
endogenous variables (see dependent variables)
error (see specific type of error)
EstimatedPdf class, 68
estimation
  about, 2, 91, 100
  exploratory process, ix
  exponential distributions, 98–99
  guessing distribution, 91–93
  guessing variance, 93–94
  Kaplan-Meier, 169, 181
  KDE, 67–68
  linear least squares, 120–122
  regression model parameters, 139
  sampling bias, 97
  sampling distributions, 94–97

survival curves, 168
survival function, 171
estimators
about, 91–92, 100
biased, 93–95, 99–100, 185
unbiased, 93
ethics, professional, 10, 24, 88, 145, 172
EWMA (exponentially-weighted moving average), 152, 154, 162
exogenous variables (see explanatory variables)
expected remaining lifetime, 178–180
explanatory variables, 129–133, 135–138, 140, 143, 149, 157
exploratory data analysis
about, 1
cleaning data, 7
DataFrame data structure, 4–6
designing visualizations, 30
importing data, 3
interpreting data, 9
NSFG survey, 2–3
statistical approach to, 2
validating data, 8
variables used, 6
exponential distributions, 49–52, 60, 62, 65, 69, 98–99, 187
exponentially-weighted moving average (EWMA), 152, 154, 162
extrapolation, 176

F

false negative, 111, 115
false positive, 111, 115, 156
field size, 47
FillBetween function, 122, 159, 173
fillna method, 153, 162
first babies case study, 1, 20–21, 110
FitLine function, 118
fork, xi
frequency, 15–16, 25, 27, 37, 70, 108–109

G

Gaussian distribution (see normal distribution)
generative process, 61
Git repositories, xi
GitHub hosting service, xi
goodness of fit, 122–124, 128
gorilla study, 94–97, 183–186
groupby method, 82, 147–147, 175, 177

H

hashable types, 70
hazard function, 167–171, 181
HazardFunction class, 168, 170, 176
height, human, 62, 79–80
hexbin plots, 81
Hist object, 16–20, 25, 27–28, 30, 69–70, 104, 109, 170
histograms, 16–20, 25
hockey, 99
hockey stick graph, 145
Holm-Bonferroni method, 113
hypothesis testing
about, 2, 114
applying CLT, 190
chi-squared tests, 109
choosing best test statistic, 105–107
classical, 101–102, 111
difference in mean, 104
effect size and, 112
exploratory process, ix
first babies case study, 110
HypothesisTest class, 102–111
replication and, 113
statistically significant effect, 111
testing correlations, 107
testing proportions, 108
HypothesisTest class, 102–111, 124, 161

I

importing data, ix, 3, 145–147
income, 76, 136–137
index column (DataFrame), 5, 34–35, 37
inf value, 180
installation considerations, xii
interarrival times, 49, 63
intercept, 117, 119
internal classes, 70
interpolation, 68
interpreting data, 9
interquartile range, 44, 48
inverse CDF, 43, 46, 48, 60, 71, 185, 189–190
IPython notebook, xii, 11
IQ scores, 123
iterative solver, 139

J

James Joyce Ramble, 47
Janert, Philipp, 161
jittering, 80, 89
join operation, 134–135, 144

K

Kaplan-Meier estimation, 169, 181
KDE (kernel density estimation), 67–68, 77
kernel density estimation (KDE), 67–68, 77

L

lag interval, 154, 155–156, 162
least squares fit, 117, 128
LeastSquares function, 118
length, pregnancy (see pregnancy length)
likelihood of an outcome, 139
line plots, 28
linear algebra, 84
linear fit, ix, 128
linear least squares
  about, 117
  estimating, 120–122
  goodness of fit, 122–124
  implementing, 118–119
  least squares fit, 117–118
  residuals and, 119
  testing linear models, 124–125
  weighted resampling, 126–127
linear models, testing, 124–125
linear regression, 129, 143, 148–151, 157
linear relationships, 86
linear transformation, 184
log odds, 139
logarithm, 195
logarithmic scale, 51
logistic regression, 137–141, 144
logit function, 140
lognormal distribution, 56–57, 63, 87, 187–188, 195
longitudinal studies, 3, 12

M

MacKay, David, 103
marital status, 143, 169–171, 176
marriage curve, 170
mass, 65
matplotlib package, xii, 16
maximum likelihood estimator (MLE), 93, 98, 100, 118, 139
mean
  about, 22, 52
  computing, 36
  rolling, 151, 162
  standard scores and, 83
  testing difference in, 104
mean error, 94
mean squared error (MSE), 92–93, 95, 98–100, 123
MeanVar function, 85
measurement error, 91, 97, 100, 117, 121, 173
median, 44, 48, 91–92, 180
memoryless processes, 179
missing values, 8, 108, 148, 151–153, 162
MLE (maximum likelihood estimator), 93, 98, 100, 118, 139
mode, 22, 25
modeling error, 157–160, 172
models
  about, 49, 61, 63
  estimated PDFs, 68
  exponential distributions, 49–52
  generating random numbers, 60–61
  hypothesis testing and, 102–105
  income distribution, 76
  least squares fit, 120
  linear regression, 148–151
  lognormal distribution, 55, 60
  measuring goodness of fit, 122–124
  normal distribution, 52
  normal probability plot, 54–55, 60
  Pareto distribution, 57–60
  regression (see regression analysis)
  relationships between variables, 117
  stationary, 160, 163
  testing linear, 124–125
moment of inertia, 73
moments, 72, 77
moving averages, 151, 162
MSE (mean squared error), 92–93, 95, 98–100, 123
multiple births, 136
multiple regression, ix, 129, 131–133, 143
multiple tests, 113

N

NaN (not a number), 7, 9, 17, 82, 108, 119, 125, 135, 153–154, 170
National Survey of Family Growth (NSFG), x, 3, 30, 39, 43, 52, 101, 104, 114, 134
natural experiments, 88, 89
NBUE, 180, 181
Newsweek magazine, 172, 176
Newton's method, 139
noise, 80, 151, 152–153, 157–160, 171
nonlinear relationships, 86–87, 120, 133
normal distribution
  about, 17, 26, 52, 183
  estimation and, 91
  for heights, 62
  normal probability plot and, 54–57
  PDFs of, 65
  smoothing and, 69
  Spearman's rank correlation and, 87
  standard deviation and, 83
  variance and, 93
normal probability plot, 54–57, 60, 64, 187–188, 190
normalization, 27, 37
NSFG (National Survey of Family Growth), x, 3, 30, 39, 43, 52, 101, 104, 114, 134
null hypothesis, 101–105, 107, 109–111, 114–115, 124–125, 190–196
NumPy package
  about, xii
  array of probabilities and, 72
  binning data, 82
  converting date/time values, 145–147
  correlation test and, 192
  covariance and, 84
  exploratory data analysis, 5, 10
  KDE and, 68–68
  logistic regression and, 140–141
  normal probability plot and, 54
  random numbers and, 34
  scatter plots and, 80, 150
  testing CLT, 187

O

observer bias, 32, 35–37
odds, 138, 144
OLS object, 130
one-sided test, 106, 115, 191
ordinary least squares, 129–130, 138, 143
orthogonal vector, 84
outliers, 20–20, 22, 26, 74–75, 81–82, 84, 87, 91
oversampling, 3, 13, 33, 127

P

p-value
  about, x
  autocorrelation functions and, 156
  computing small values, 192
  hypothesis testing and, 102, 104–109, 111, 113–115, 190
  linear models and, 125–125
  linear regression and, 149
  regression analysis and, 130, 133, 137, 140
pandas package
  about, xii
  binning data, 82
  cohort effects and, 175
  computing correlation, 87
  computing statistics, 23
  counting values, 15
  covariance and, 85
  data structures supported, 4–5
  extrapolating survival curves, 176
  hazard function and, 168
  importing and cleaning data, 145–147
  NaN values and, 8, 17
  normal probability plot and, 54
  regression analysis and, 135, 140
  rolling mean and, 151
  serial correlation and, 155
parabolas, 133
parameters
  computing sampling distribution of, 125
  data compression and, 61
  estimating, 121, 138–139
  exponential distributions, 49–52
  linear regression models, 129–130
  lognormal distribution, 56
  minimizing residuals, 117
  normal distribution, 52
  Pareto distribution, 58
Pareto distribution, 57–60, 61–63, 187–189
Pareto World, 62
Pareto, Vilfredo, 57
Patsy syntax, 130, 136, 140
Pdf class, 66, 69

PDFs (probability density functions)
about, 65–67, 67–68, 77
Cdf implementation, 71–72
distribution framework, 69
Hist implementation, 69–70
moments and, 72
Pmf implementation, 70
skewness and, 73–75
Pearson coefficient of correlation, 83, 85–87, 107, 123, 192
Pearson median skewness, 74–75, 77
Pearson, Karl, 85
percent point function, 190
percentile rank, 40–41, 44–48, 83
percentiles, 41, 44, 48, 52, 82, 185
permutation, 104, 107, 112, 114, 124, 191
permutation test, 106, 115
plots
bar, 28
hexbin, 81
line, 28
normal probability, 54–57, 60, 64, 187–188, 190
for PMFs, 28–30
scatter, 79–82, 86, 89, 119, 150
for time series, 147
Pmf class
about, 27–28
class size paradox, 30–33
computing mean and variance, 36
distribution framework, 69
implementing, 70
KDE and, 68
passing with, 72
PDFs and, 66–67
plotting PMFs, 28–30
survival function and, 178
PMFs (probability mass functions)
about, 27–28, 37
computing mean, 36
DataFrame indexing, 34–35
designing visualizations, 30
limits of, 39, 39
plotting, 28–30
Poisson regression, 137, 142, 144
population, 3, 12, 58, 94
power (tests), 112, 115
prediction, 86, 93, 122–124, 135–137, 157–161, 176
preferential attachment, 61
pregnancy length
applying CLT, 190
effect size, 23
first babies and, 1, 20–21
histogram of, 18–20
hypothesis testing, 101–102, 104–107, 110, 112–114
normal probability plot and, 55
plotting PMFs, 29
regression analysis and, 136
representing CDFs for, 43
survival analysis and, 165–168, 178–180
variables for, 6
variance of, 23
Price of Weed website, 145
probability, 27, 37, 138
probability density, 65, 77
probability density functions (PDFs)
about, 65–67, 67–68, 77
Cdf implementation, 71–72
distribution framework, 69
Hist implementation, 69–70
moments and, 72
Pmf implementation, 70
skewness and, 73–75
probability mass functions (PMFs)
about, 27–28
albout, 37
class size paradox, 30–33
computing mean, 36
DataFrame indexing, 34–35
designing visualizations, 30
plotting, 28–30
product, 195
proof by contradiction, 102
properties (Python), 167
proportions, testing, 108
proxy variables, 136, 144
pseudo r-squared, 140
pumpkin weight, 23
pyplot interface, 16, 29, 148

Q

quadratic model, 133, 161
quantiles, 45, 48
quantizing, 69
quartiles, 44
query method, 166

quintiles, 45

R

r-squared (coefficient of determination), 122–124, 128, 130, 132–137, 140
race, 135–137, 140
race times, 47
random module, 61–62
random numbers, 45, 48, 54, 60–61, 63, 97
randomized controlled trials, 88–89
rank, 83, 87, 89
raw data, 6, 13
raw moments, 72, 77
recodes (variables), 6, 13
records, 12
regression, 129, 143
regression analysis
  accuracy of, 141–142
  causal relationships and, 88
  data mining and, 134–136
  estimating parameters, 139
  goal of, 129
  implementing models, 140
  logistic regression, 137–141
  multiple regression, 131–133
  nonlinear relationships and, 133
  prediction and, 135–137
  StatsModels package, 130–131
RegressionResults object, 130
reindexing, 151–153, 162
relationships between variables
  about, 79
  causation and, 88
  characterizing, 82
  correlation and, 83, 88
  covariance and, 84
  modeling, 117
  nonlinear, 86–87, 120
  Pearson's correlation, 85–87
  scatter plots, 79–82
  Spearman's rank correlation, 87–89
relay races, 36
replacement, 45, 48, 79, 112, 114, 121, 127
repositories (Git), xi
representative studies, 3, 13
Resample function, 112
resampling
  about, 114
  autocorrelation function and, 156
  chi-squared test and, 193
  cohort effects and, 175
  correlation test and, 192
  missing values and, 153
  quantifying sampling error, 158, 173
  simulating experiments, 121
  weighted, 126–127, 173
resampling test, 115
residuals, 117–120, 128, 131, 151, 156–160
respondents, 3, 12, 57
RMSE (root MSE), 92–93, 98–100, 122, 131
robust statistic, 74–75, 77, 84, 87
rolling mean, 151, 162
root MSE (RMSE), 92–93, 98–100, 122, 131

S

sample mean, 95, 99
sample median, 98–99
sample size, 95, 114, 186
sample skewness, 73, 77
sample variance, 23, 93
SampleRows function, 79
samples, 3, 12, 42
sampling bias, 97, 100, 121
sampling distribution, 94–97, 99–100, 121, 121, 125–127, 183–186, 190–192
sampling error, 95, 100, 121, 157, 159, 172, 175
sampling weight, 126–128, 173
SAT scores, 123
saturation, 81, 89
scatter plots, 79–82, 86, 89, 119, 150
SciPy package, xii, 52, 62, 66–68, 185, 192, 193
seasonality, 151, 153, 155
selection bias, 2, 37
self-selection, 97
sensitivity (tests), 112
serial correlation, 153–156, 162
Series data structure
  about, 5
  autocorrelation functions and, 156
  computing correlation, 87
  computing differences between elements, 162
  counting values, 8–10
  DataFrame indexing and, 34
  extrapolating survival curves, 176
  fillna method, 153
  hazard function and, 168

mapping variable names and parameters, 130
normal probability plot and, 54
NSFG variables, 17
rolling mean and, 151
sex, 136–137
sex ratio, 140, 142
shape, 44
significant effect (see clinically significant results; statistically significant effect)
simple regression, 129, 143
simulation, 68, 95, 103, 121, 156, 160, 185
skewness, 73–77, 84, 87–87, 125, 187–188
slope, 117, 119, 160, 162
smoothing, 61, 69, 152
soccer, 99
span parameter, 153, 162
Spearman coefficient of correlation, 83, 87–89, 107
spread, 22, 26, 44
spurious relationships, 88, 131, 143
squared residuals, 118
standard deviation
about, 23, 26, 83, 89
chi-squared statistic and, 110
computing sampling distributions, 196
lognormal distribution and, 57
moments and, 73
normal distribution and, 52
normal probability plot and, 54–55
PDFs and, 66
Pearson median skewness and, 74
Pearson's correlation and, 85
pooled, 24
of residuals, 122
RMSE and, 131
sampling distributions and, 95–96, 183, 185–186
standard scores and, 83, 83
testing for difference in, 107
standard error, 96, 97–100, 121, 127, 183, 185, 196
standard normal distribution, 64, 189
standard scores, 83, 85, 89
standardized moments, 73, 77
stationary model, 160, 163
statistically significant effect
about, 102, 108, 115
correlation test and, 191
difference in birth weight, 105
difference in pregnancy length, 105, 106, 110–111
linear regression and, 149–150
multiple regression and, 130
regression analysis and, 133, 137, 141–142
replication and, 113
serial correlation and, 155–156
testing linear models, 124
testing proportions, 108–109
threshold of, 104, 111
weighted resampling and, 126
StatsModels package, xii, 130–131, 140, 142, 148, 155
step functions, 42, 67
Straight Dope, The, 25
Student's t-distribution, 192
studies
cross-sectional, 3, 12
gorilla, 94–97, 183–186
longitudinal, 3, 12
representative, 3, 13
summary statistics, 22, 26, 44
survival analysis
about, 165, 181
cohort effects, 173–176
confidence intervals and, 172
estimating survival function, 171
expected remaining lifetime, 178–180
extrapolation, 176
hazard function, 167–169
Kaplan-Meier estimation, 169
marriage curve, 170
survival curves, 165–169
survival curves, 165–169, 181
survival rate, 165
SurvivalFunction class, 166, 171
symmetric distributions, 74, 125, 191

T

tail, 22, 26, 193
telephone sampling, 97
test statistic, 102–103, 105–107, 109, 114, 194
tests
chi-squared, 109
choosing best test statistic, 105–107
for linear models, 124–125
multiple, 113
one-sided, 106, 115, 191

testing CLT, 187–190
testing correlations, 107
testing proportions, 108
two-sided, 106, 115, 191
underpowered, 113
thinkplot module
exponential distribution and, 50–51
extracting height and weight, 79
hexbin plots and, 81
normal probability plot and, 54
PDFs and, 67–68, 74–75
plotting CDFs, 43, 45
plotting distribution of test statistics, 105
plotting distributions, 32
plotting fitted lines, 122
plotting for time series, 147
plotting histograms, 16–17, 21
plotting percentiles of weight versus weight, 82
plotting PMFs, 28–30
survival functions and, 167
testing CLT, 187
threshold, 104, 111
time series
about, 145, 162
autocorrelation function and, 155–156
importing and cleaning data, 145–147
linear regression and, 148–151
missing values and, 153
moving averages and, 151–153
plotting for, 147
prediction and, 157–161
serial correlation, 153–156
transactions, 146
transparency, 81
treatment groups, 88–89
trends, 151
Trivers-Willard hypothesis, 142
true negative, 141
true positive, 141
two-sided test, 106, 115, 191

U

UBNE, 180, 181
unbiased estimators, 93
underpowered tests, 113
uniform distribution, 17, 26, 48, 60, 80, 84, 156
units, 83, 85
US Census Bureau, 58, 76

V

validating data, 8
variables, relationships between (see relationships between variables)
variance, 23, 26, 36, 93–94
visualization, ix, 30, 68, 120, 147, 177

W

Weibull distribution, 62
weight
adult, 79–80, 195
birth (see birth weight)
gorilla, 184
pumpkin, 23
sampling, 126–128
weighted resampling, 126–127, 173
windows, 151–153, 162
wrappers, 70, 168, 185

X

xticks command, 148

About the Author

Allen Downey is an Associate Professor of Computer Science at the Olin College of Engineering. He has taught computer science at Wellesley College, Colby College, and U.C. Berkeley. He has a PhD in Computer Science from U.C. Berkeley and Master's and Bachelor's degrees from MIT.

Colophon

The animal on the cover of *Think Stats*, second edition is an archerfish, or spinner fish (*Toxotidae*). This family of fish preys on land-based insects and small animals, using their specialized mouths to shoot them down with water droplets. This family consists of seven species, which can be found from India to the Philippines, Australia, and Polynesia.

The archerfish has a deep body; the space between the dorsal fin and mouth forms a straight line. The protractile mouth has a lower jaw that juts out. The shape of its mouth lends itself directly to feeding: the narrow groove in the roof of its mouth allows it to squirt a jet of water at its victim by pressing its tongue against the groove and contracting its gills to force the powerful jet of water out, which can travel up to five meters. Archerfish learn how to shoot when they reach 2.5 cm long. Often they are innaccurate at first and hunt in small schools, eventually learning from experience.

The archerfish's eyes are also valuable tools for feeding. It has particularly good eyesight and is able to compensate for light refraction as it passes through the air-water interface when aiming at prey. Once it spots its prey, the archerfish rotates its eye so the image of the prey falls on a particular portion of the eye. Often, the archerfish will leap out of the water to grab the insect in its mouth, if within reach.

Archerfish are usually small, at 5–10 cm, but can grow up to 40 cm long. Archerfish are popular aquarium fish.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *Dover*. The cover fonts are URW Typewriter 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.