Think Stats: Probability and Statistics for Programmers

Version 2.0.10

Think Stats

Probability and Statistics for Programmers

Version 2.0.10

Allen B. Downey

Green Tea Press

Needham, Massachusetts

Green Tea Press
9 Washburn Ave
Needham MA 02492

The original form of this book is LaTeX source code. Compiling this code has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book is available from http://thinkstats2.com.

The cover for this book is based on a photo by Paul Friel (http://flickr.com/people/frielp/), who made it available under the Creative Commons Attribution license. The original photo is at http://flickr.com/photos/frielp/11999738/.

Preface

Why I wrote this book

Think Stats: Probability and Statistics for Programmers is an introduction to the practical tools of exploratory data analysis.

When I start working with a new dataset, I follow these steps:

- 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 values mean and what distributions of the variables are, and identifying 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 likely that the apparent effect could be 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 books takes a computational approach, which has several advantages:

- I present most ideas using simple 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 running 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 Monte Carlo 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 data that has 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 exploratory data 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.

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. Often there are phrases, and errors, that propagate from one book to the next; Stephen Jay Gould pointed out an example in his essay, "The Case of the Creeping Fox Terrier."[1]

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, the bugbear of librarians everywhere. 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 (of course) Google.

Using the code

The code and data used in this book are available from `https://github.com/AllenDowney/ThinkStats2`. Git is a version control system that allows you to keep track of the files that make up a project. A collection of files

[1] A breed of dog that is about half the size of a Hyracotherium (see `http://wikipedia.org/wiki/Hyracotherium`).

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, but if you do, you'll have your own repository on GitHub that you can use to keep track of code you write while working on this book.
- You can also **clone** my repository, which means that you create a copy of the files on your computer. You don't need a GitHub account to do this, but you won't be able to write your changes back to my repository.
- If you don't want to use Git at all, you can download the files in a Zip file.

The code in this book is written to work in both Python 2 and Python 3 with no translation. However, you will need the following packages installed:

- pandas for representing and analyzing data, `http://pandas.pydata.org/`.
- NumPy for basic numerical computation, `http://www.numpy.org/`.
- SciPy for scientific computation including statistics, `http://www.scipy.org/`.
- matplotlib for visualization, `http://matplotlib.org/`.

Although these are commonly-used packages, they are not included with all Python installations, and they can be hard to install in some environments. I developed this book using Anaconda from Continuum Analytics, which is a free Python distribution that includes all 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 privilege. And it supports both Python 2 and Python 3. You can download Anaconda from `http://continuum.io/downloads`.

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 some 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 at `http://ipython.org/ipython-doc/stable/notebook/notebook.html`.

Allen B. Downey
Needham MA

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

Contributor List

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

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

- 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.

Contents

Chapter 1

Data science for programmers

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 the gestation period 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?

1.1 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.

1.2 The National Survey of Family Growth

Since 1973 the U.S. 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."[1]

We will use data collected by this survey to investigate whether first babies tend to come late, and 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.

The people who participate in a survey are called **respondents**; a group of respondents is called a **cohort**. 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. Of course 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. The reason for oversampling is 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.

[1] See `http://cdc.gov/nchs/nsfg.htm`.

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 `http://www.cdc.gov/nchs/nsfg/nsfg_cycle6.htm`

1.3 Importing the data

The code and data used in this book are available from `https://github.com/AllenDowney/ThinkStats2`. For information about downloading and working with this code, see Section .

Once you download the code, 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."

Let's see what it does. Pregnancy data from Cycle 6 of the NSFG is in a file called `2002FemPreg.dat.gz`. As the suffixes suggest, it is a gzip-compressed data file in plain text (ASCII), with fixed width columns. Each line in the file 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 the dictionary in `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 library that contains many classes and functions used in this book. `thinkstats2.py` provides functions to read the Stata dictionary and the NSFG data file. Here's how it's 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')
    CleanFremPreg(df)
    return df
```

`thinkstats2.ReadStataDct` takes the name of the dictionary file and returns a `FixedWidthVariables` object, which contains the information from the dictionary file and provides `ReadFixedWidth`, which reads the data file.

1.4 DataFrames

The result of `ReadFixedWidth` is a DataFrame, which is the fundamental data structure provided by pandas. 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.

```
>>> import nsfg
>>> df = nsfg.ReadFemPreg()
>>> df
...
[13593 rows x 243 columns]
```

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 243 columns/variables.

The DataFrame attribute `columns` returns a sequence of column names:

```
>>> 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 variable 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
```

The indices are the integers from 0 to 13593. In this case the elements are also integers.

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.

1.5 Variables

We have already seen two variables in the NSFG dataset, `caseid` and `pregordr`, and we have seen that there are 243 variables in total. For the explorations in this book, I'll 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 counter that indicates the order of pregnancies for a respondent.
- `birthord` is a counter that indicates the order of live births for a respondent; for example, the code for a first child is 1. For outcomes other than live birth, 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. Members of oversampled groups have lower weights.

If you read the codebook carefully, you will see that most of these 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, `prglength` 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 unless there is a compelling reason to process the raw data yourself.

1.6 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 birth. In addition it uses 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 easily generate bogus results, like a 99-pound baby. The `replace` method replaces these values with `NaN`, a special floating-point value that represents "not a number."

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!

1.7 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 the method `value_counts`, which 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 notice one value that is almost certainly 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 `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; in this case, it replaces them with `NaN`.

1.8 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 match up each pregnancy record with a respondent. Here's a function that does that:

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

`df` is the DataFrame with pregnancy data. The `iteritems` method creates an iterator that 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. 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]
    print(caseid, preg.outcome[indices].values)
```

Using a list as an index into `preg.outcome` selects the indicated values and yields a Series. Instead of printing the whole Series, I selected the `values` attribute, which is a NumPy array. The result looks like this:

```
10229 [4 4 4 4 4 4 1]
```

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.

1.9 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 `http://ipython.org/ipython-doc/stable/notebook/notebook.html`.

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 `chap01ex_soln.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.

Similarly, you might want to make a dictionary that maps from each `caseid` to an index into the respondent DataFrame.

A solution to this exercise is in `chap01ex_soln.py`

Exercise 1.3 The best way to learn about data science is to work on a project you are interested in. Is there a question like, "Do first babies arrive late," that you want to investigate?

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 at `http://www3.norc.org/gss+website/`, and the European Social Survey at `http://www.europeansocialsurvey.org/`.

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!

1.10 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 kinds of things, 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.
- **respondent**: A person who responds to a survey.
- **cohort**: A group of respondents.
- **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 sub-population in order to avoid errors due to small sample sizes.
- **record**: In a database, a collection of information about a single person or other object of study.
- **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

2.1 Histograms

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. For details, see `https://docs.python.org/2/library/collections.html#collections.Counter`.

Or in pandas, we can use the Series method `value_counts`, which we saw in the previous chapter.

2.2 Representing histograms

The Python module that accompanies this book, `thinkstats2.py`, provides a class named Hist, which represents a histogram. You can read the documentation at `http://thinkstats2.com/thinkstats2.html`.

The `Hist` constructor can take a sequence, dictionary, pandas Series, or another Hist. You can test it in Python's interactive mode:

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

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

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

If you are planning to look up all of the frequencies, it is more efficient to use `Items`, which returns an unsorted list of value-frequency pairs:

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

2.3 Plotting histograms

There are a number of Python packages for making figures and graphs. One of the most popular is `pyplot`, which is part of the `matplotlib` package (see `http://matplotlib.org`).

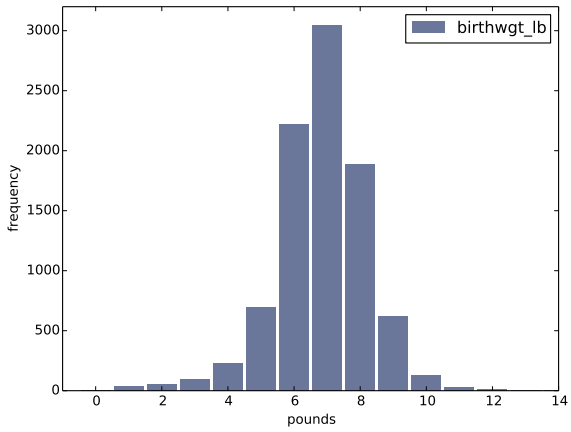

Figure 2.1: Histogram of the pound part of birth weight.

This package is included in many Python installations. To see whether you have it, launch the Python interpreter and run:

```
>>> import matplotlib.pyplot as pyplot
>>> pyplot.pie([1,2,3])
>>> pyplot.show()
```

If you have `matplotlib` you should see a simple pie chart; otherwise you will have to install it.

Histograms are most often plotted as bar charts. The `pyplot` function to draw a bar chart is `bar`. Hist objects provide a method called `Render` that returns a sorted list of values and a list of the corresponding frequencies, which is the format `bar` expects:

```
>>> vals, freqs = hist.Render()
>>> rectangles = pyplot.bar(vals, freqs)
>>> pyplot.show()
```

The code that accompanies this book includes `thinkplot.py`, which provides functions for plotting histograms and other objects we will see soon. You can read the documentation at `http://thinkstats2.com/thinkplot.html`.

To plot the same histogram with `thinkplot.py`, try this:

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

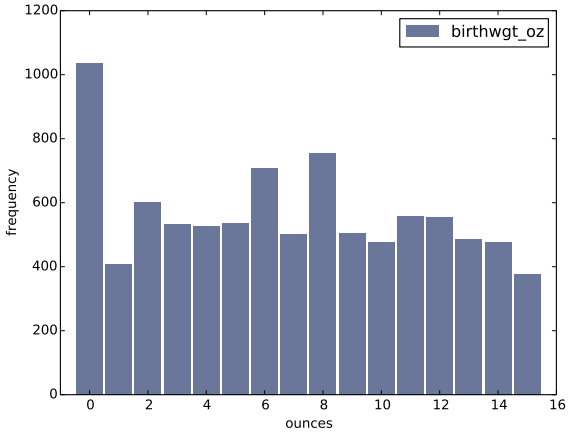

Figure 2.2: Histogram of the ounce part of birth weight.

Most of the functions in `thinkplot.py` are wrappers for functions in `matplotlib`, customized to work with the objects defined in `thinkstats2.py`.

2.4 NSFG variables

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

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 Section 1.6 we transformed `agepreg` from centiyears to years, and combined `birthwgt_lb` and `birthwgt_oz` into a single quantity, `totalwgt_lb`. In this section I'll 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'll generate and plot the histogram of `birthwgt_lb` for live births (this code is in `first.py`).

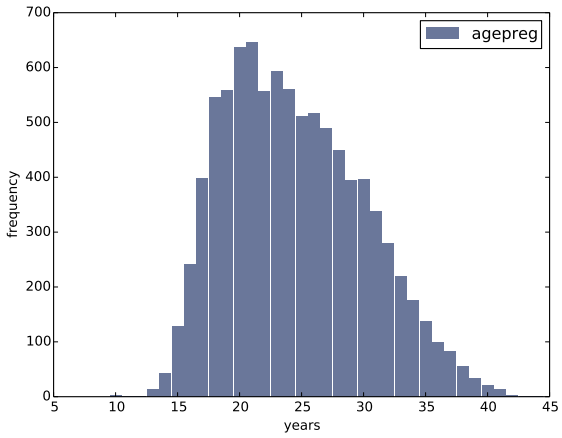

Figure 2.3: Histogram of mother's age at end of pregnancy.

```
hist = thinkstats2.Hist(live.birthwgt_lb, label='birthwgt_lb')
thinkplot.Hist(hist)
thinkplot.Show(xlabel='weeks', 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 **Gaussian** or distribution, also called a **normal** distribution. But unlike a true Gaussian 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, because of biological limits.

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, again for biological reasons; premature babies are common,

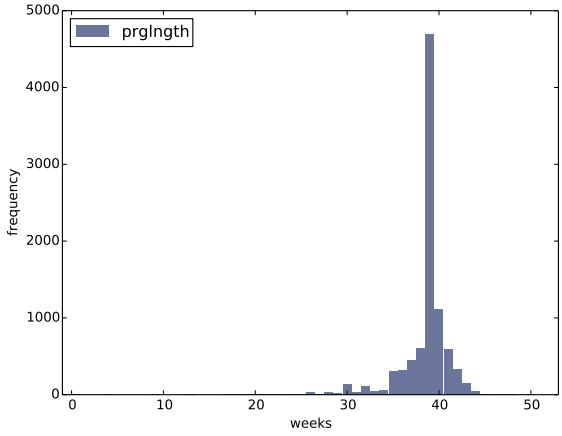

Figure 2.4: Histogram of pregnancy length in weeks.

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

2.5 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 measurements of rare events.

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

```
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.

2.6 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 as an argument the number of histograms we are planning to plot; it uses this information to choose an appropriate collection of colors.

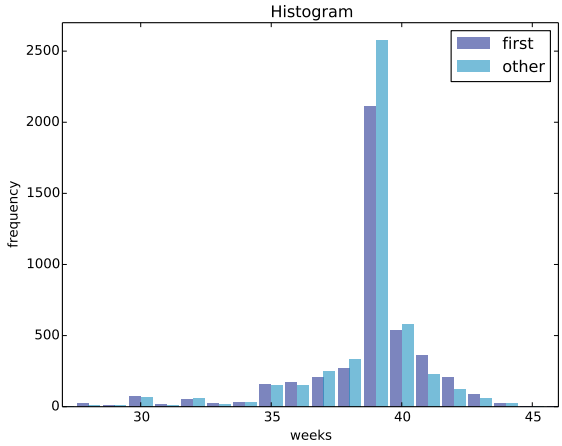

Figure 2.5: Histogram of pregnancy lengths.

`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 adjusted the axis to show only data between 27 and 46 weeks. Figure 2.5 shows the result.

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.

2.7 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 that 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; in other words

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

The words "mean" and "average" are sometimes used interchangeably, but I will maintain 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.

2.8 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, which is why it is denoted with exponent 2. 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 is 7.3, which is hard to interpret, especially since the units are weeks^2, or "square weeks," whatever that means. Variance is useful in some calculations we'll see later, but it is not a good summary statistic.

2.9 Effect size

An **effect size** is a summary statistic intended 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.

For example, the 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 that 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", which is a weighted sum of the standard deviations within groups. Here's what it looks like in Python:

```
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. To put that in perspective, the difference in height between men and women is about 1.7 standard deviations (see `https://en.wikipedia.org/wiki/Effect_size`).

2.10 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.

2.11 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* (`http://straightdope.com`), 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 you should try.

A solution to this exercise is in `chap02ex_soln.ipynb`

For the following exercises, create a file named `chap02ex.py`. You can find a solution in `chap02ex_soln.py`.

Exercise 2.3 The mode of a distribution is the most frequent value (see `http://wikipedia.org/wiki/Mode_(statistics)`). 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 Use 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 ($d = 0.028$)?

2.12 Glossary

- distribution: A summary of 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.
- Gaussian distribution: An idealization of a bell-shaped distribution; also known as a normal 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 the most average 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

3.1 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.py`. 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])
```

The result is normalized so the total probability is 1.

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

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
```

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 for PMFs can be confusing, so here it is: I use Pmf for the name of the class, `pmf` as a variable name, and "PMF" to refer to the mathematical concept of a probability mass function.

3.2 Plotting PMFs

There are two common ways to plot Pmfs:

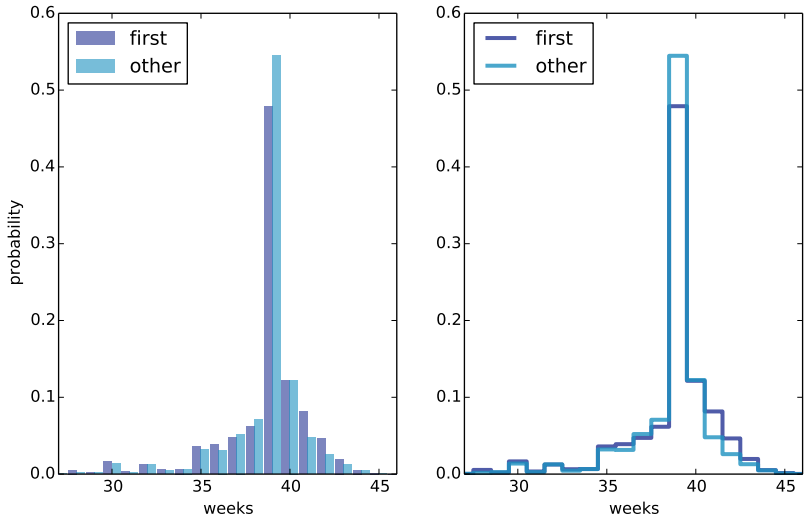

Figure 3.1: PMF of pregnancy lengths for first babies and others, using bar graphs and step functions.

- 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.

Figure 3.1 shows PMFs of pregnancy length for first babies and others using bar graphs (left) and step functions (right).

By plotting the PMF instead of the histogram, we can compare the two distributions without distortion caused 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 a late (weeks 41 and 42).

Whether you plot PMFs as bar graphs or step functions, I'll leave that up to you. The code that generates this figure is in `probability.py`. Note: `pyplot` provides a function called `hist` that takes a sequence of values, computes the histogram and plots it. Since I use `Hist` objects, I usually don't use `pyplot.hist`.

The code that generates Figure 3.1 is in `probability.py`:

```
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])
```

`PrePlot` takes optional parameters `rows` and `cols` to make a grid of figures, in this case 1 row of 2 figures. The first figure (on the left) displays the Pmfs using `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 `Pmf`. 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.

3.3 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 your 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 to 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. This figure

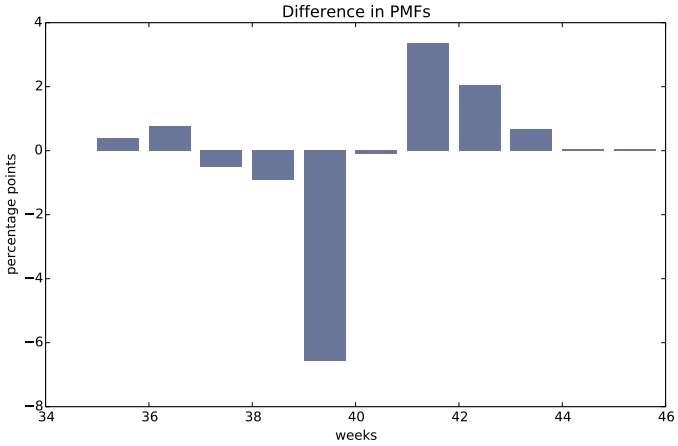

Figure 3.2: Difference, in percentage points, by week.

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.

3.4 The class size paradox

Before we go on, I want to look at an example that demonstrates one kind of computation you can do with Pmf objects: I call this example the "class size paradox."

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 was bigger. Let's see how much bigger.

First, I'll 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)
```

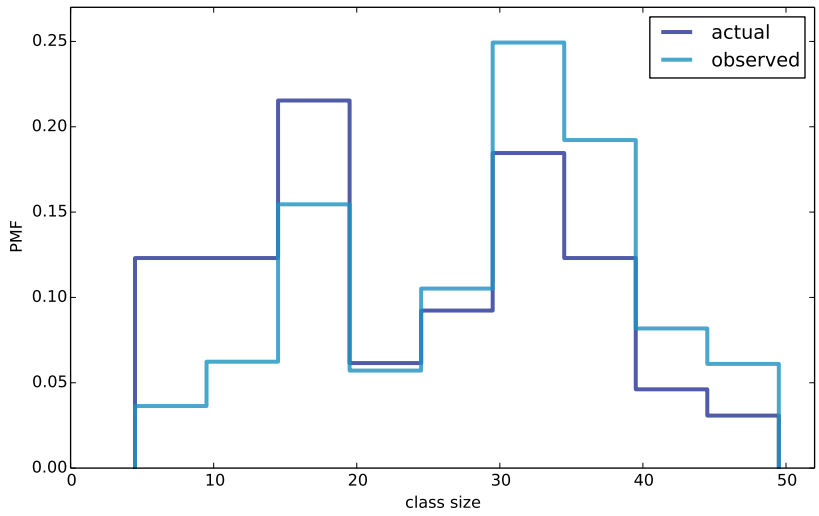

Figure 3.3: Distribution of class sizes, actual and as observed by students.

```
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.

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 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.

The code for this example is in `chap03.py`. It confirms that if we start with the actual distribution, apply `BiasPmf` and apply `UnbiasPmf` to the result, we get the original distribution back.

3.5 DataFrame indexing

In Section 1.4 we read a pandas DataFrame and used it to select and modify data columns. Now let's look at row selection. To start, I'll 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)
>>> pandas.DataFrame(array)
          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']
>>> pandas.DataFrame(array, columns=columns)
          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']
>>> pandas.DataFrame(array, columns=columns, index=index)
          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, and `iloc` can take a list of integer positions. 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.

3.6 Exercises

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` You can find a solution in `chap03ex_soln.ipynb`.

For the following exercises, create a file named `chap03ex.py`. You can find a solution in `chap03ex_soln.py`.

Exercise 3.2 In Section 2.7 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 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 started running my leg, 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. To see why, think about the extremes. If another runner is going at the same speed as me, neither of us will overtake the other. If someone is going so fast that they cover the entire course while I am running, they are certain to overtake me.

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 `http://thinkstats2.com/relay_soln.py`

3.7 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.

Chapter 4

Cumulative distribution functions

4.1 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 Gaussian 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.

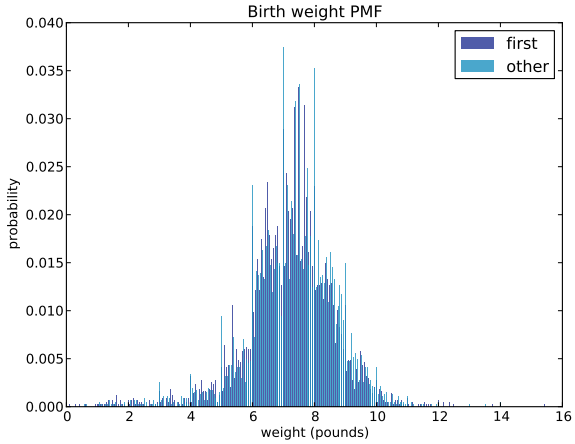

Figure 4.1: PMF of birth weights. This figure shows a limitation of PMFs: they are hard to compare.

4.2 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 scores 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
```

The code in this section is in `chap04.py`. 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 very 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 score and computes its percentile rank in a set of scores; `Percentile` takes a percentile rank and computes the corresponding score.

4.3 Conditional distributions

A **conditional distribution** is the distribution of a subset of the data, selected according to some condition.

For example, if you are above average in weight, but way above average in height, then you might be relatively light for your height. Here's how you could make that claim more precise.

1. Select a cohort of people who are the same height as you (within some range).

2. Find the CDF of weight for those people.

3. Find the percentile rank of your weight in that distribution.

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; the results are available from `http://coolrunning.com/results/10/ma/Apr25_27thAn_set1.shtml`. 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 is 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 "male 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 in the next 10 years I should expect to slow down by 3–4 minutes.

I maintain a friendly rivalry with a student who is in the F2039 division. How fast does she have to run her next 10K to "beat" me in terms of percentile ranks?

4.4 Cumulative distribution functions

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 in a distribution.

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 sample, `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 should look familiar; it is almost identical to `PercentileRank`, except that the result is in a probability in the range 0–1 rather than a percentile rank in the range 0–100.

As an example, suppose a sample has 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.

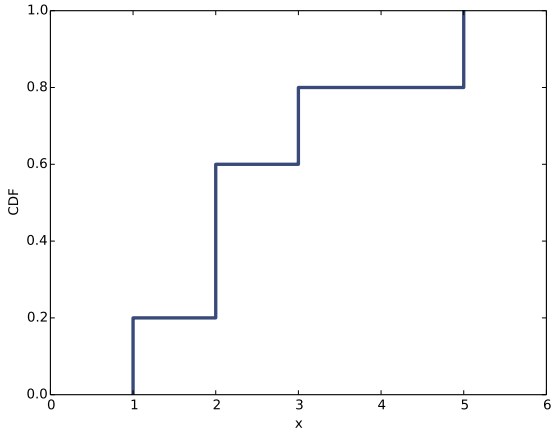

Figure 4.2: Example of a CDF.

4.5 Representing CDFs

`thinkstats2` provides a class named `Cdf` that represents CDFs. Cdfs are implemented with two sorted lists: `xs`, which contains the values, and `ps`, which contains the probabilities. The fundamental methods `Cdf` provides are:

- `Prob(x)`: Given a value `x`, computes the probability $p = \text{CDF}(x)$.
- `Value(p)`: Given a probability `p`, computes the corresponding value, `x`; that is, the **inverse CDF** of `p`.

Because `xs` and `ps` are sorted, these operations use the bisection algorithm, which is efficient; the run time is proportional to the logarithm of the number of values; see `http://wikipedia.org/wiki/Time_complexity`.

The Cdf constructor can take as an argument a list of values, a pandas Series, a Hist, Pmf, or another Cdf.

`thinkplot.py` provides functions named `Cdf` and `Cdfs` that plot Cdfs as lines. For example, here's the code to make and plot the CDF of pregnancy length for all live births.

```
live, firsts, others = first.MakeFrames()
cdf = thinkstats2.Cdf(live.prglngth, label='prglngth')
thinkplot.Cdf(cdf)
thinkplot.Show(xlabel='weeks', ylabel='CDF')
```

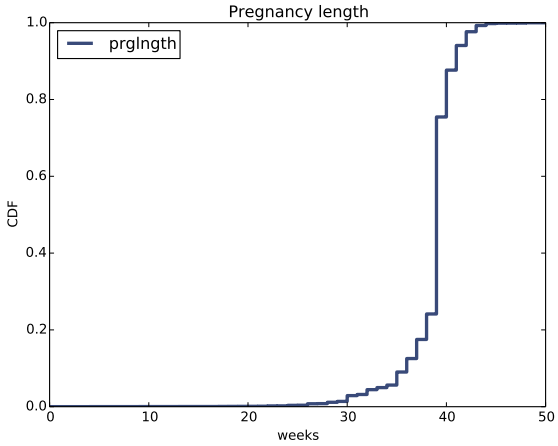

Figure 4.3: CDF of birth weights.

This code is in `cumulative.py`. 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 very few values below 30 weeks, so the CDF in this range is flat.

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.

4.6 Comparing CDFs

CDFs are especially useful for comparing distributions. For example, here is the code that plots the CDF of birth weight 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. 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.

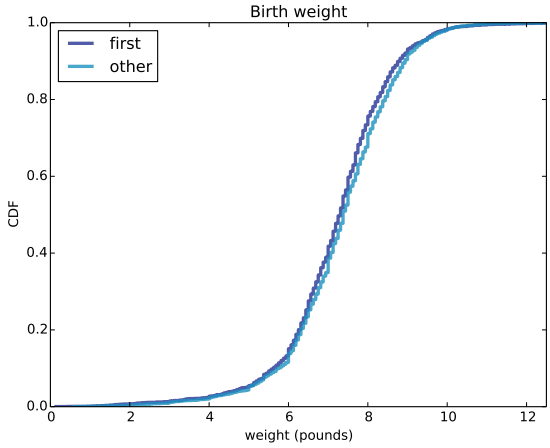

Figure 4.4: CDF of birth weights for first babies and others.

4.7 Percentile-based statistics

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

- `PercentileRank(x)`: Given a value `x`, computes its percentile rank, 100CDF(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 the 20th, 40th, 60th and 80th percentiles. Other distributions are divided into ten "deciles". Statistics like these that represent equally-spaced points in a CDF are called **quantiles**. For more, see `https://en.wikipedia.org/wiki/Quantile`.

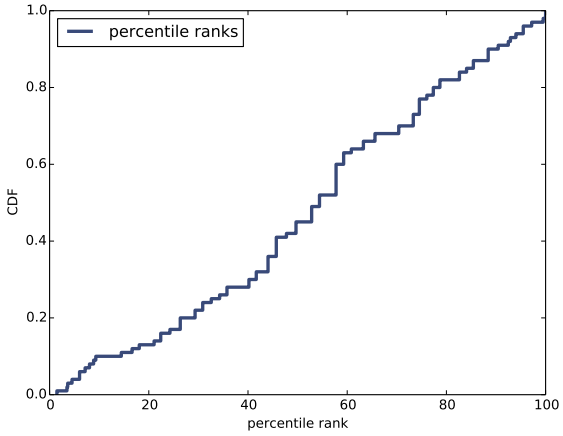

Figure 4.5: CDF of percentile ranks for a random sample of birth weights.

4.8 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 the code to compute it:

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

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

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

`cdf` is the distribution of all birthweights. `sample` is a random sample of 100 birthweights, chosen with replacement; that is, the same value could be chosen more than once.

`ranks` is a list of percentile ranks for the 100 values in the sample. `rank_cdf` is the CDF of those ranks. Figure 4.5 shows the result. The CDF is approximately a straight line, which means that the distribution is uniform.

That result might be non-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% are 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.

If `Sample` works correctly, the distribution of the sample should match the original distribution. Here's how we can check:

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

    sample = cdf.Sample(1000)
    sample_cdf = thinkstats2.Cdf(sample, label='sample')
    thinkplot.Cdfs([cdf, sample_cdf])
    thinkplot.Show(xlabel='weight (pounds)',
                   ylabel='CDF')
```

Again, `cdf` is the distribution of birth weights for live births. `sample` is a random sample drawn from `cdf`, and `sample_cdf` is its Cdf.

This code is in `cumulative.py`; you can run it to confirm that `Cdf.Sample` works.

4.9 Exercises

For the following exercises, you can start with `chap04ex.ipynb`. My solution is in `chap04ex_soln.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?

You can read about the uniform distribution at `http://wikipedia.org/wiki/Uniform_distribution_(discrete)`.

4.10 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.
- conditional distribution: The distribution of a subset of the data, selected according to some condition.
- 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 probability, p, to the corresponding value.
- median: The 50th percentile, often used as a measure of central tendency.
- interquartile range: A measure of spread, the difference between the 75th and 25th percentiles.
- quantile: A sequence of values that correspond to equally-spaced percentile ranks; for example, quartiles are the 25th, 50th and 75th percentiles.

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 an analytic function (as opposed to a step 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 Section .

5.1 The exponential distribution

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

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

The parameter, λ, determines the shape of the distribution. Figure 5.1 shows what this CDF looks like with $\lambda = 0.5, 1, 2$.

In the real world, exponential distributions come up when we look at a series of events and measure the times between events, called **interarrival**

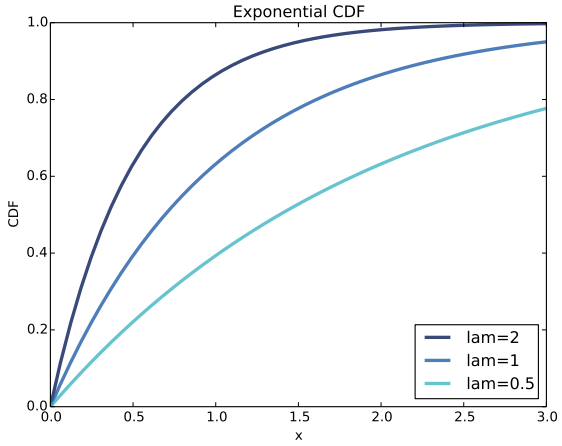

Figure 5.1: CDFs of exponential distributions with various parameters.

times. If the events are equally likely to occur at any time, the distribution of interarrival times tends to look like an exponential distribution.

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 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 shows the CDF (left). It seems to have the general shape of an exponential distribution, but how can we tell?

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

[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).

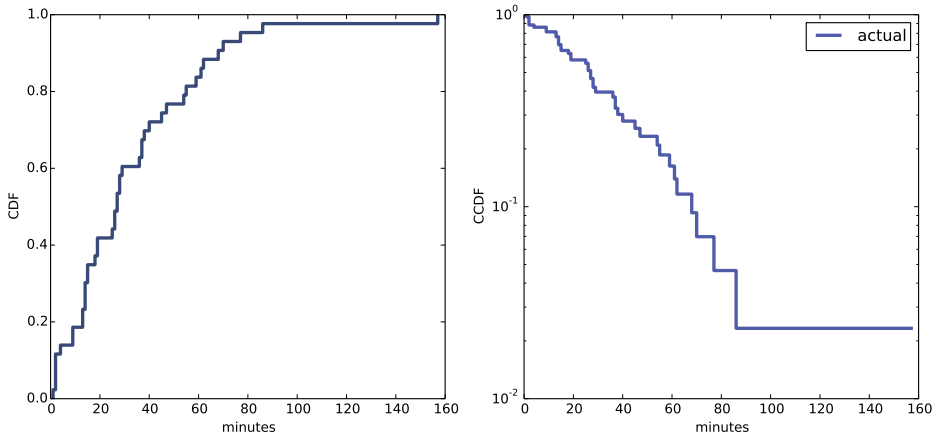

Figure 5.2: CDF of interarrival times (left) and CCDF on a log-y scale (right).

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 shows the result (right). 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. In that case 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

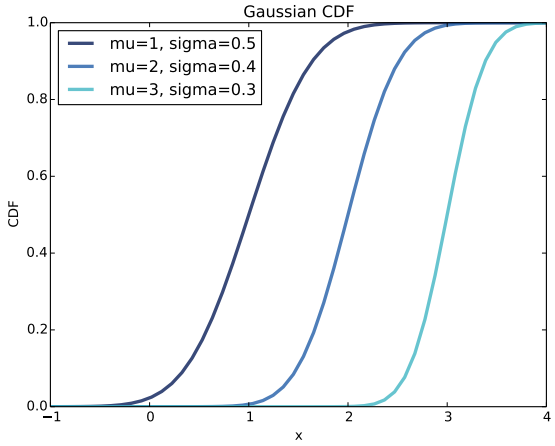

Figure 5.3: CDF of normal distributions with a range of parameters.

born in 24 hours, so the rate is $\lambda = 1.83$ births per hour. The mean of an exponential distribution is $1/\lambda$, so the mean time between births is 0.55 hours.

5.2 The normal distribution

The **normal distribution**, also called Gaussian, is commonly used because it describes so many phenomena, at least approximately. It turns out that there is a good reason for its ubiquity, which we will get to in Section 14.4.

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.

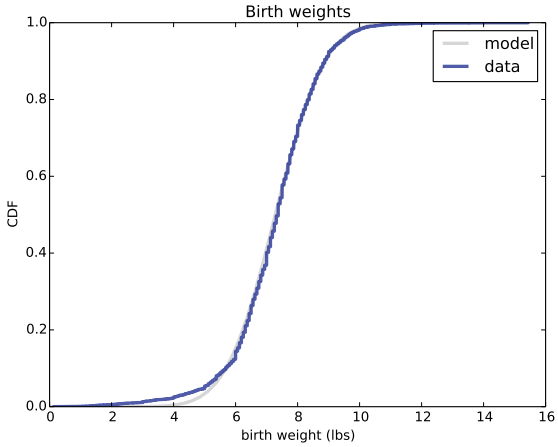

Figure 5.4: CDF of birth weights with a normal model.

`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 `GaussianCdf`, which takes parameters `mu` and `sigma` and evaluates the CDF at `x`:

```
def EvalGaussianCdf(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 we can summarize the distribution with the parameters $\mu = 7.28$ and $\sigma = 1.24$, and the resulting error (difference between the model and the data) is small.

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 interested in studying 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.

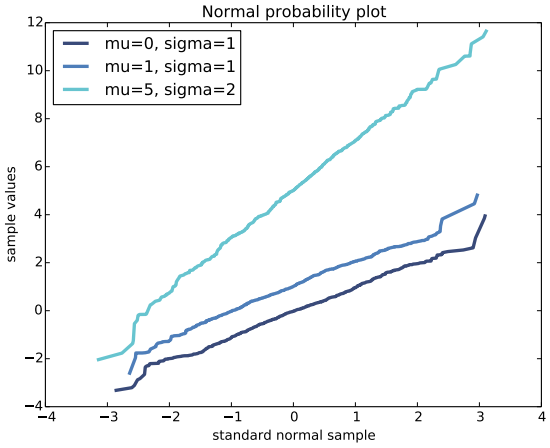

Figure 5.5: Normal probability plot for random samples from normal distributions.

5.3 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 of 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. If you are interested in the hard way, you can read about it at `https://en.wikipedia.org/wiki/Normal_probability_plot`. 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 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.

```
mean = weights.mean()
std = weights.std()

xs = [-4, 4]
xs, ys = thinkstats2.FitLine(xs, inter=mean, slope=std)
thinkplot.Plot(xs, ys, 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.

`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 deviation 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.

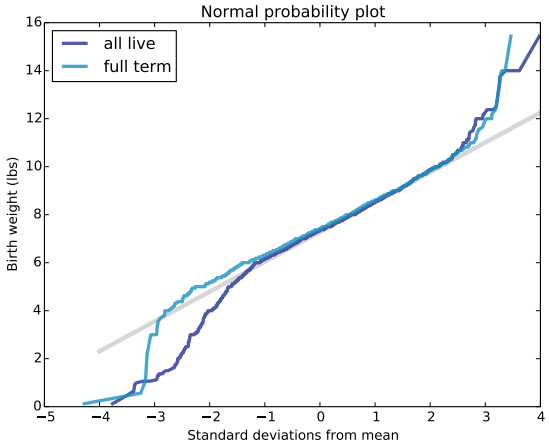

Figure 5.6: Normal probability plot of birth weights.

5.4 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.

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

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 `http://wikipedia.org/wiki/Log-normal_distribution`).

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]

[2]I was tipped off to this possibility by a comment (without citation) at `http://mathworld.wolfram.com/LogNormalDistribution.html`. 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. Online at `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1636707`.

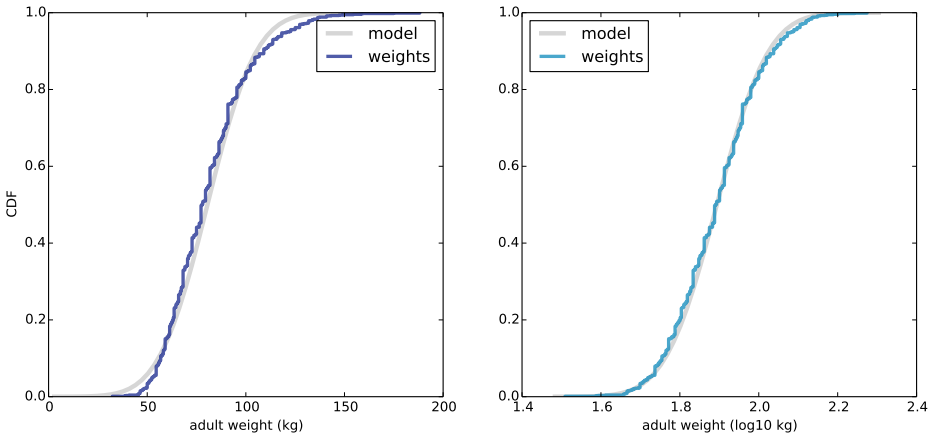

Figure 5.7: CDF of adult weights on a linear scale (left) and log scale (right).

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] 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 analyses the data.

Figure 5.7 shows the distribution of adult weights on a linear scale, with a normal model, and 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 5.8 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 would conclude that the lognormal distribution is a good model for this data.

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

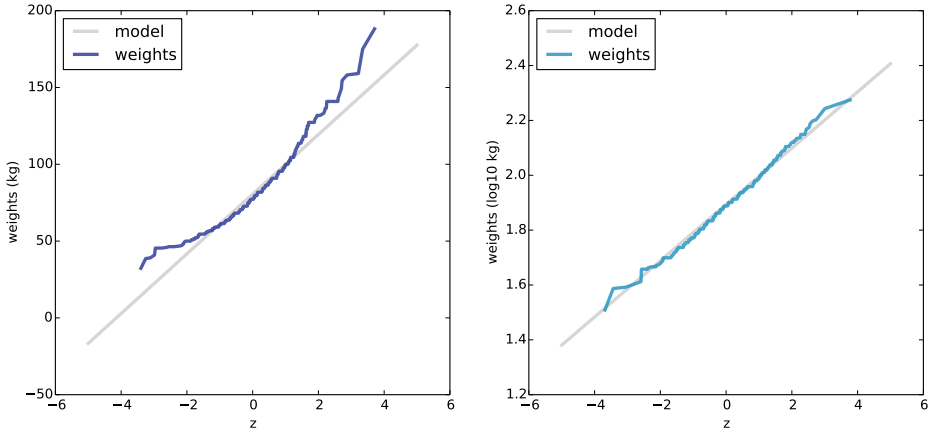

Figure 5.8: Normal probability plots for adult weight on a linear scale (left) and log scale (right).

5.5 The Pareto distribution

The **Pareto distribution** is named after the economist Vilfredo Pareto, who used it to describe the distribution of wealth (see `http://wikipedia.org/wiki/Pareto_distribution`). 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, forest fires and earthquakes.

The CDF of the Pareto distribution is:

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

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:

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

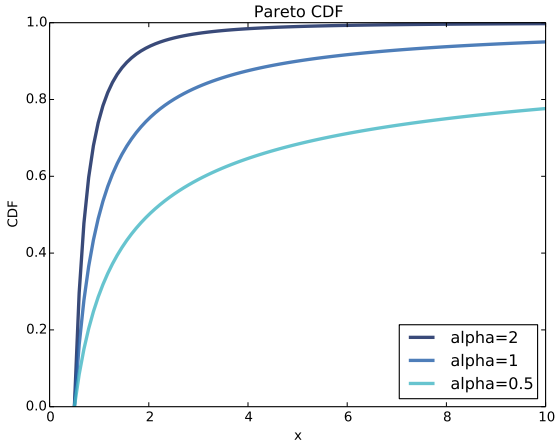

Figure 5.9: CDFs of Pareto distributions with different parameters.

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 in the United States. The U.S. Census Bureau publishes data on the population of every incorporated city and town in the United States.

I downloaded their data from `http://www.census.gov/popest/data/cities/totals/2012/SUB-EST2012-3.html`; 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 data file and plots the distribution of populations.

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.

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

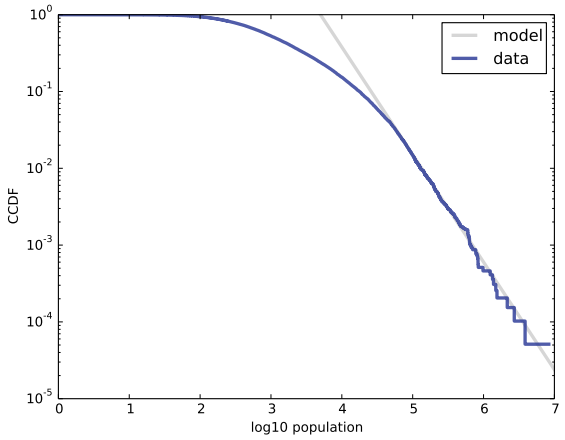

Figure 5.10: CCDFs of city and town populations, on a log-log scale.

model is a better fit for the other 99%. Which model is appropriate depends on which part of the distribution you is relevant.

5.6 Generating random numbers

Analytic CDFs can be used to generate random numbers with a given distribution function, $p = \text{CDF}(x)$. 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 from 0 to 1, 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`.

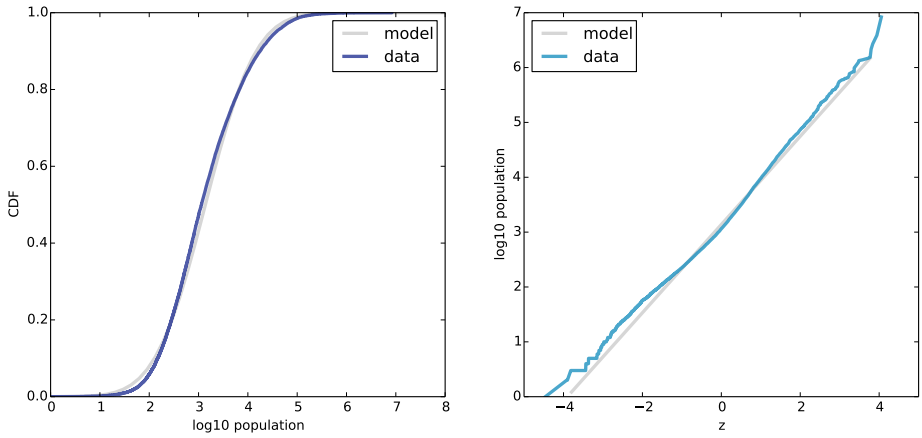

Figure 5.11: CDF of city and town populations on a log-x scale (left), and normal probability plot of log-transformed populations (right).

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. Most implementations of `random.random` can return 0 but not 1, so $1 - p$ can be 1 but not 0, so that works out well.

5.7 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: see `http://wikipedia.org/wiki/Preferential_attachment`.).

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.

5.8 Exercises

For the following exercises, you can start with `chap05ex.ipynb`. My solution is in `chap05ex_soln.ipynb`.

Exercise 5.1 In the BRFSS (see Section 5.4), 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 between 5'10" and 6'1" (see `http://bluemancasting.com`). What percentage of the U.S. male population is in this range? Hint: use `scipy.stats.norm.cdf`.

Exercise 5.2 To get a feel for the Pareto distribution, imagine what the world would be like if the distribution of human height were Pareto. Choosing 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.

What is the mean human height in Pareto world? What is the mean of this sample? 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 (see `http://wikipedia.org/wiki/Weibull_distribution`). 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 Section 5.1 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. A solution to this problem is in `chap05ex_soln.py`.

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 different distributions.

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 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 `http://www.census.gov/hhes/www/cpstables/032013/hhinc/toc.htm`. 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`.

5.9 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 $1 - \mathrm{CDF}(x)$.
- 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.
- model: A useful simplification. Analytic distributions are often good models of more complex empirical distributions.

Chapter 6

Probability density functions

The code for this chapter is in `density.py`. For information about downloading and working with this code, see Section .

6.1 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, `xs` and `ps`.
- `MakePmf`, which uses `Render` to evaluate the 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 `GaussianPdf` that evaluates the Gaussian density function.

```
class GaussianPdf(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 GaussianPdf object contains the parameters `mu` and `sigma`. `Density` uses `scipy.stats.norm`, which is an object that represents a Gaussian distribution and provides `cdf` and `pdf`, among other methods (see Section 5.2).

The following example creates a GaussianPdf with the mean and variance of adult female heights, in cm, from the BRFSS (see Section 5.4). 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)
```

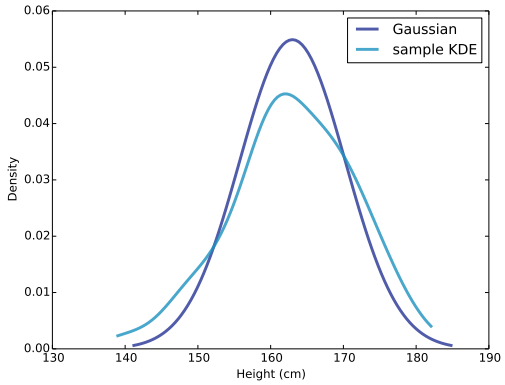

Figure 6.1: A Gaussian PDF that models adult female height in the U.S., and the kernel density estimate of a sample with $n = 1000$.

```
>>> pdf = thinkstats2.GaussianPdf(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='Gaussian')
>>> 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 see in the next section.

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`.

6.2 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 `http://en.wikipedia.org/wiki/Kernel_density_estimation`.

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, calls gaussian_kde.evaluate, and returns the resulting density.

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

```
>>> sample = [random.gauss(mean, std) for i in range(100)]
>>> sample_pdf = thinkstats2.EstimatedPdf(sample)
>>> thinkplot.Pdf(pdf, label='sample KDE')
```

sample is a list of 100 random heights. sample_pdf is a Pdf object that containts 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 Gaussian density function and a KDE based on a sample of 1000 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 smooth, you can use KDE to interpolate the density for values that don't appear in the sample.

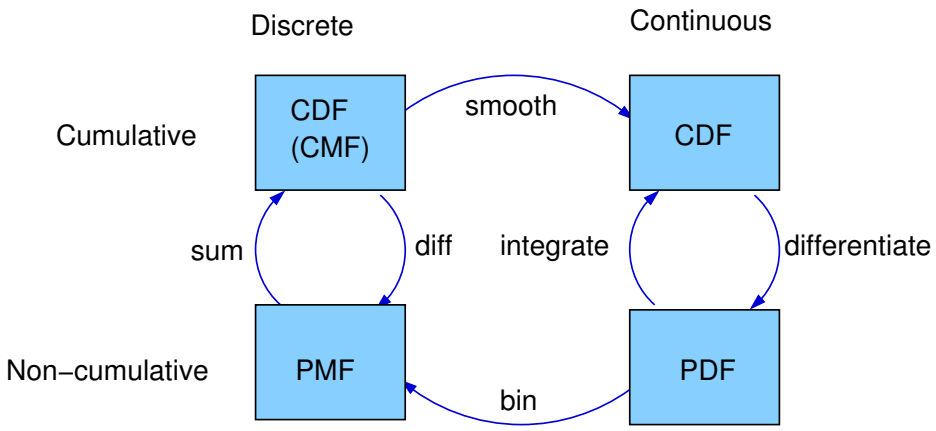

Figure 6.2: A framework that relates representations of distribution functions.

Simulation : If your sample is an input to a simulation, it might be appropriate to smooth the sample distribution using KDE. That might allow the simulation to explore more possible outcomes, rather than replicating the observed data.

6.3 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.

We started with PMFs, which represent the probabilities for a discrete set of values. To get from a PMF to a CDF, you can compute a cumulative sum. To be more strictly correct, a discrete CDF should be called a cumulative mass function (CMF), but as far as I can tell no one uses that term. To get from a CDF to a PMF, you can compute differences in cumulative probabilities.

Similarly, a PDF is the derivative of a continuous CDF; or, equivalently, a CDF is the integral of a PDF. But 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.

6.4 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.

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

```
    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]
```

Because `Hist` operators and methods are based on dictionaries, you should expect getting and setting items to be constant time operations; that is, their run time does not increase as the Hist gets bigger.

`_DictWrapper` also provides special methods to support getting and setting items, so instead of using `Incr` you could write:

```
>>> hist = thinkstats2.Hist()
>>> hist['a'] += 1
```

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

6.5 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 said to be normalized.

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

```
    def Normalize(self, fraction=1.0):
        total = self.Total()
        if total == 0.0:
            raise ValueError('Normalize: 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 and error.

`Hist` and `Pmf` have the same constructor. It can take as an argument a Python `dict`, a 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.

6.6 Cdf implementation

A CDF is a map from values to their cumulative probabilities, so it would be reasonable to implement `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 mapping from cumulative probability to value. So the implementaion I chose is two sorted lists. That way I can use bisection to do a forward or inverse mapping 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.

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

```
    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
```

And here is the implementation of `Value`, which takes a cumulative probability and returns the corresponding value:

```
    def Value(self, p):
        if p < 0 or p > 1:
            raise ValueError('Probability 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`:

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

`np.roll` shift 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.

6.7 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):
    xbar = RawMoment(xs, 1)
    return sum((x - xbar)**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 place a mass along a number line at each value, x_i, and then spin the number line around the mean, the moment of inertia of the spinning masses is the variance of the values. If you are not familiar with moment of intertia, see `http://en.wikipedia.org/wiki/Moment_of_inertia`. .

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. For the second moment, we solve the problem by computing the square root of variance, known as the standard deviation, which is in the same units as x_i.

6.8 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 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.

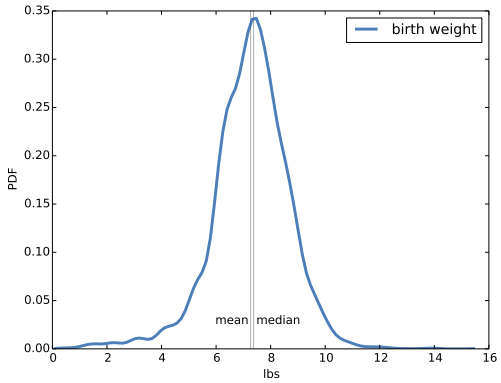

Figure 6.3: Estimated PDF of birthweight data from the NSFG.

Pearson's median skewness coefficient is an alternative measure of skewness that explicitly captures the relationship 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 birthweights 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')
```

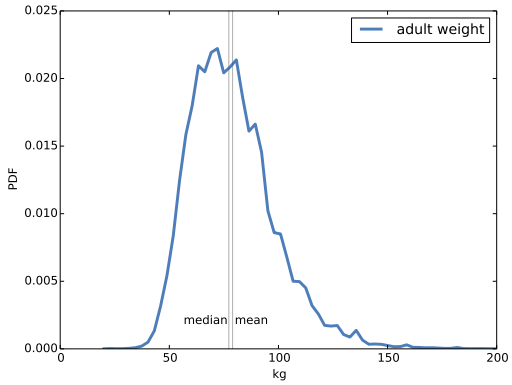

Figure 6.4: Estimated PDF of adult weight data from the BRFSS.

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.

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.

6.9 Exercises

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 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 `http://www.census.gov/hhes/www/cpstables/032013/hhinc/toc.htm`. 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 "$250,0000 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?

A solution to this exercise is in `hinc2_soln.py`.

6.10 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 characteristic of a distribution; intuitively, it is a measure of how asymmetric the 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.
- 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 people. 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 Section .

7.1 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 Section 5.4).

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
```

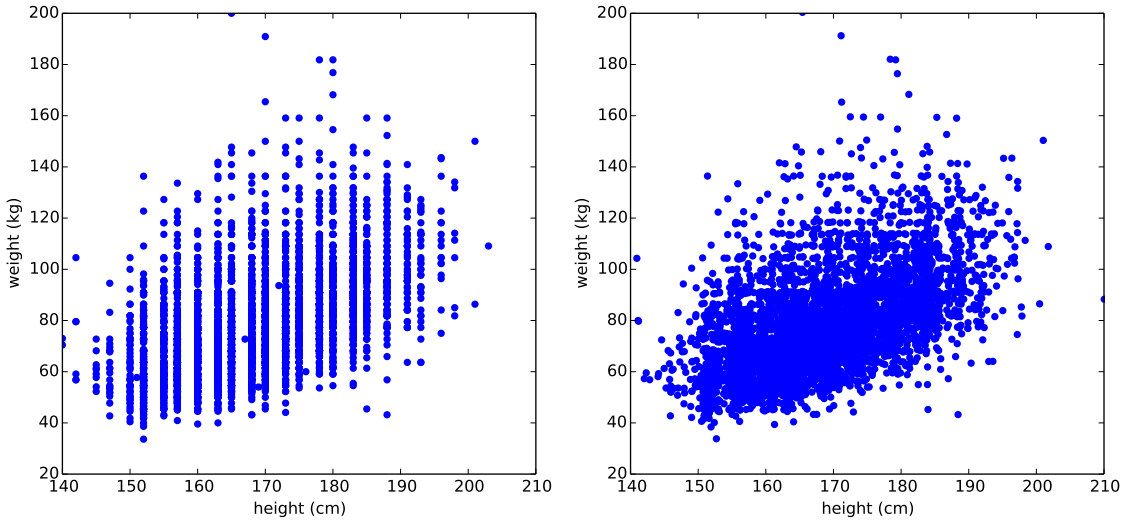

Figure 7.1: Scatter plots of weight versus height for the respondents in the BRFSS, unjittered (left), jittered (right).

`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 one.

`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.

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.

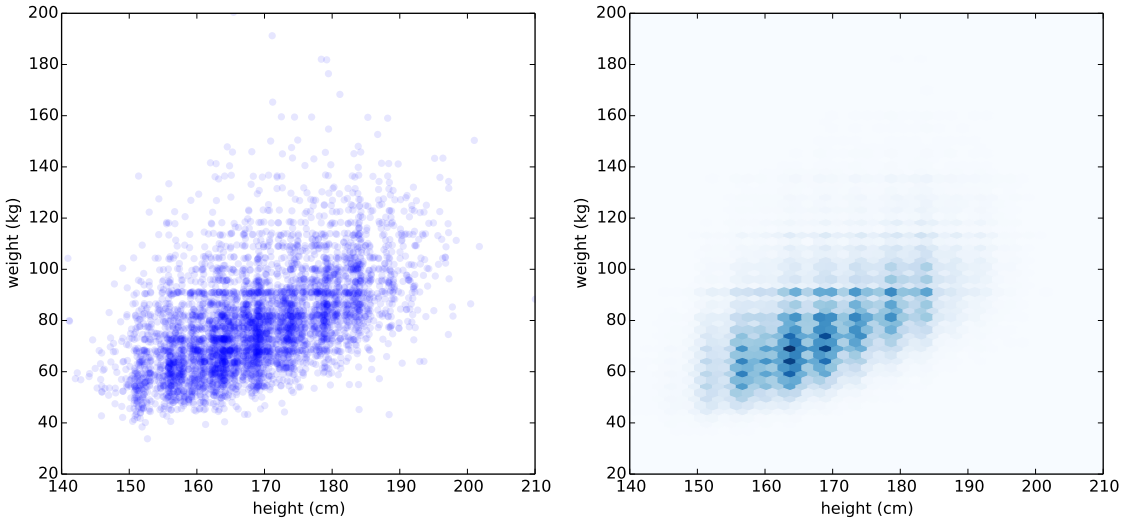

Figure 7.2: Scatter plot with jittering and transparency (left), hexbin plot (right).

```
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 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.

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, but in time and in the size of the file it generates. A drawback is that it makes the outliers invisible.

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

7.2 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 use one variable to bin the data and plot percentiles of the other variable.

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 columns listed in `subset`.

`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

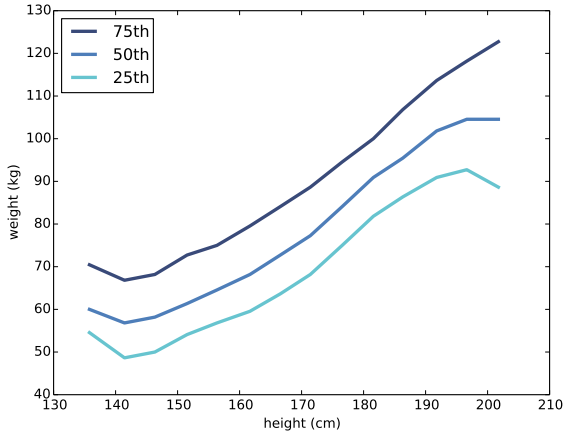

Figure 7.3: Percentiles of weight for a range of height bins.

lowest bin are mapped to index 0. Values above the highest bin are mapped to `len(bins)`. So `indices` is one element longer than `bins`.

`groupby` is a pandas method that divides a DataFrame into groups. The result is an object that iterates over the groups. So, for example, we can print the number of rows in each group:

```
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][1:-1]
    cdfs = [thinkstats2.Cdf(group.wtkg2) for i, group in groups][1:-1]
```

The slice operators omit the first and last groups, which contain the values that fall outside the range of bins. 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 linear. This range includes more than 99% of the data, so we don't have to worry too much about the extremes.

7.3 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:

1. Transform all values to **standard scores**. This leads to the "Pearson product-moment correlation coefficient."

2. Transform all values to their percentile ranks. This leads to the "Spearman rank correlation coefficient."

If X is a series of 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 that case it is more robust to use percentile ranks. If we compute a new variable, R, so that r_i is the percentile rank of x_i, the distribution of R is uniform between 0 and 100, regardless of the distribution of X.

7.4 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 othogonal, and negative if they point in opposite directions. `thinkstats2` uses `np.dot` to implement `Cov` efficiently:

```
def Cov(xs, ys, meanx=None, meany=None):
    if meanx is None:
        meanx = np.mean(xs)
    if meany is None:
        meany = np.mean(ys)

    cov = np.dot(np.asarray(xs)-meanx, np.asarray(ys)-meany) / len(xs)
    return cov
```

By default `Cov` computes deviations from the sample means, or you can provide known values. 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.

7.5 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. In Section 10.5 I will say more about how to interpret correlations.

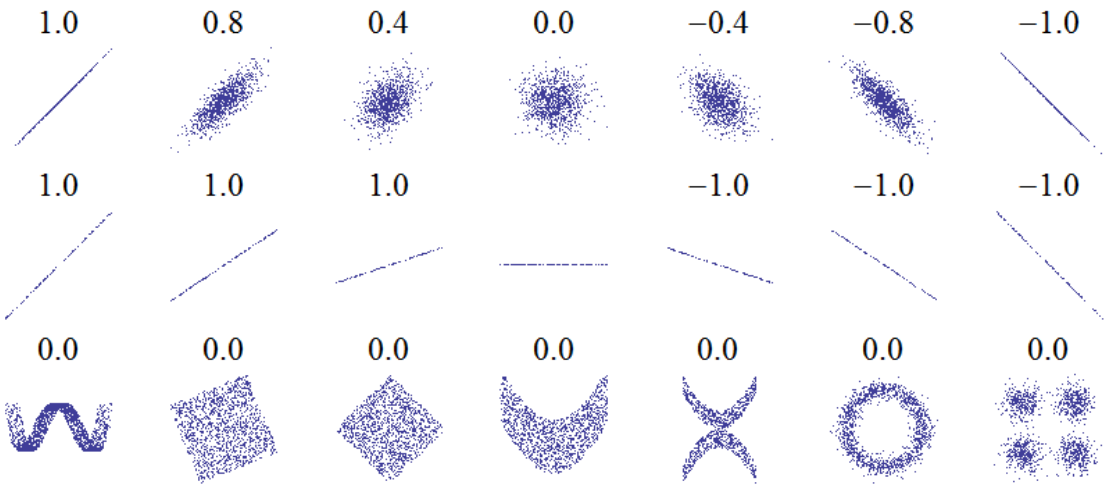

Figure 7.4: Examples of datasets with a range of correlations.

7.6 Non-linear 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 `http://wikipedia.org/wiki/Correlation_and_dependence`. It shows scatter plots and correlation coefficients for several carefully-constructed datasets.

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.

7.7 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.

Anscombe's quartet demonstrates this effect; it contains four data sets with the same correlation. One is a linear relation with random noise, one is a non-linear relation, one is a perfect relation with an outlier, and one has no relation except an artifact caused by an outlier. You can read more about it at `http://wikipedia.org/wiki/Anscombe's_quartet`.

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

Or I could 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, a little higher than the Pearson correlation, 0.51. There are several possible reasons for the difference, including:

- If the relationship is nonlinear, Pearson's correlation tends to underestimate the strength of the relationship, and
- Pearson's correlation can affected (in either direction) if one of the distributions is skewed or contains outliers. Spearman's rank correlation is immune to these effects.

In the BRFSS example, we know that the distribution of weights is roughly lognormal, so we could eliminate the effect of skewness by computing Pearson's correlation between height and the log of weight.

```
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.

7.8 Correlation and causation

If variables A and B are related, 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: `http://wikipedia.org/wiki/Correlation_does_not_imply_causation`.

So what can you do to provide evidence of causation?

1. 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.

2. 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 variable 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 the foundation of science-based medicine (see `http://wikipedia.org/wiki/Randomized_controlled_trial`).

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. You can read more about this topic at `http://wikipedia.org/wiki/Natural_experiment`.

In some cases it is possible to infer causal relationships using **regression analysis**, which is the topic of Chapter 11.

7.9 Exercises

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?

A solution to this exercise is in `chap07ex_soln.py`.

7.10 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 it expressed in terms of 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 subject 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 interventions.
- 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 Section .

8.1 The estimation game

Let's play a game. I'll 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, 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.

8.2 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:

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

For an explanation of why S^2 is biased, and a proof that S^2_{n-1} is unbiased, see `http://wikipedia.org/wiki/Bias_of_an_estimator`.

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^2_{n-1}, 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^2_{n-1}:

```
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^2_{n-1} if you provide the argument `ddof=1`, which stands for "delta degrees of freedom." I won't explain that now, but you can read about it at `http://en.wikipedia.org/wiki/Degrees_of_freedom_(statistics)`.

`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^2_{n-1}, the mean error was 0.014, about 10 times smaller. As `m` increases, we expect the mean error for S^2_{n-1} 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.

If 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.

8.3 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, $\bar{x} = 90$, is an unbiased estimator of μ, and in the long run it minimizes MSE. So if you reported 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.

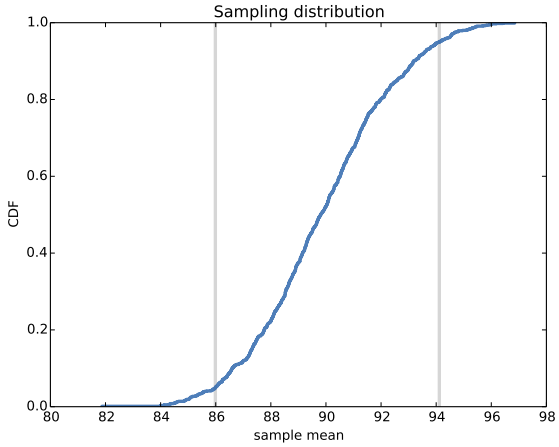

Figure 8.1: Sampling distribution of $\bar{x}$, with confidence interval.

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 1000 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.

There are two common ways to summarize the sampling distribution:

- **Standard error** (SE) is 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.
- A **confidence interval** (CI) is 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.

8.4 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 number with other women.

If factors like income, employment, and household size are related to weight (and it is plausible that they are), the results of your survey will 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 will 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.

8.5 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[1]. 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 ran this experiment with $\lambda = 2$, I found 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 minimized MSE, but at least it seems better than L_m.

[1] See http://wikipedia.org/wiki/Exponential_distribution#Maximum_likelihood.

Sadly, it seems that both estimators are biased. For L the mean error is 0.33, 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.

8.6 Exercises

For the following exercises, you might want to start with `estimation.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 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 you draw a sample with size $n = 10$ from a population with 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 1 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`?

8.7 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 errors 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 Section .

9.1 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:

[1] For more about Bayesian inference, see the sequel to this book, *Think Bayes*.

- 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 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 is 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 enough, we conclude that the null hypothesis is unlikely to be true.

9.2 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 implementations for `TestStatistic`, `RunModel`, and optionally `MakeModel`.

`__init__` takes the data in whatever form is appropriate. It calls `RunModel`, 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 stored 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 big difference if the coin is actually fair:

```
class CoinTest(thinkstats2.HypothesisTest):

    def TestStatistic(self, data):
```

[2]Adapted from MacKay, *Information Theory, Inference, and Learning Algorithms*, 2003.

```
        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 integers: the number of heads and tails. The test statistic is the absolute difference between them, so `self.actual` is 30.

`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.

Instead 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 would conclude that the data do not provide strong evidence that the coin is biased or not.

9.3 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
```

The parameter, `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 data from 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

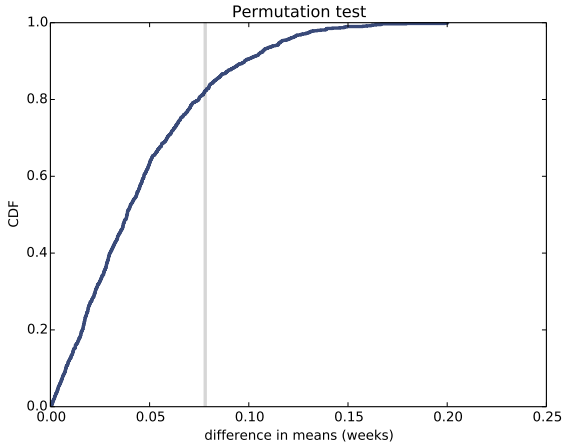

Figure 9.1: CDF of difference in mean pregnancy length under the null hypothesis.

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 yielded 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.

9.4 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.

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 Section 3.3, 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.

9.5 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'll 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.

9.6 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 a difference as big as that 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.

9.7 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, 0.12. If we take the 5% threshold seriously, we would consider this effect statistically significant. But considering the two tests togther, 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.

9.8 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 signficant. But in Section 3.3, 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, so it is a bit more complex. Here's the code that makes and runs the model:

```
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):
        """Run the model of the null hypothesis.

        returns: simulated data
        """
        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 pre-computed 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 1000 iteration, 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.

9.9 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 question:

- 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 the distribution T and its CDF CDF_T.
- Each time we run an experiment, we get a test statistic, t, which is drawn from T. Then we compute a p-value, which is the probability that a random value from 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 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.

9.10 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.

9.11 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[3]. 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 more 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 with 0.12 lbs in the original dataset).
- The correlation between birthweight 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.

9.12 Exercises

A solution to these exercises is in `chap09ex_soln.py`.

[3]See `https://en.wikipedia.org/wiki/Holm-Bonferroni_method`

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 sample size decreases? What is the smallest sample size that yields a positive test?

Exercise 9.2 In Section 9.3, 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 the resampling, but one of the simplest is to draw a sample, with replacement, from the observed values, as in Section 9.10.

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?

9.13 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.
- 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-square test: A test that uses the chi-square 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 Section .

10.1 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 non-random 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 `https://en.wikipedia.org/wiki/Linear_regression`.
- 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.

10.2 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
```

To see how this is derived, see http://wikipedia.org/wiki/Numerical_methods_for_linear_least_squares.

LeastSquares takes sequences xs and ys and returns the estimated parameters inter and slope.

thinkstats2 also provides FitLine, which takes the estimated parameters 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 value 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 expected 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.

10.3 Residuals

Another useful test is to plot the residuals. thinkstats2 provides a function that computes residuals:

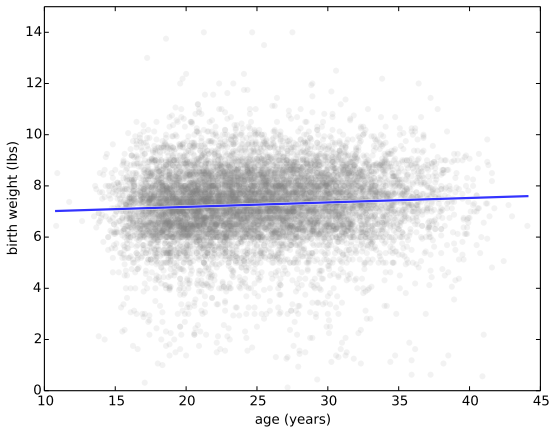

Figure 10.1: Scatter plot of birth weight and mother's age with a linear fit.

```
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 Section 7.2. 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.

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.

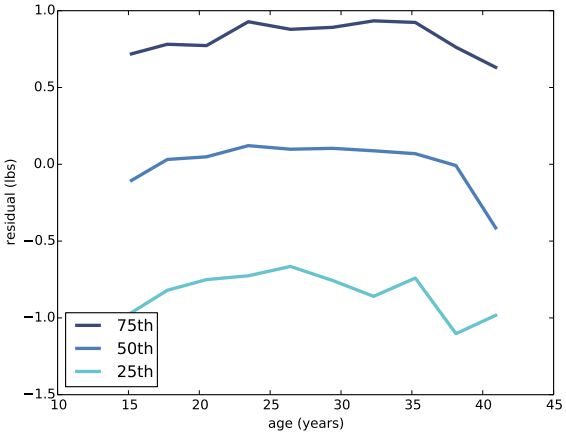

Figure 10.2: Residuals of the linear fit.

10.4 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 discussed in Chapter 8, sampling bias is caused by non-representative 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.SampleRows(live, len(live), replace=True)
        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 `SampleRows` to resample the observed pregnancies, then uses the simulated sample to estimate the parameters. It returns sequences of estimated intercepts and estimated slopes.

I summarize the sampling distributions by printing the standard error and confidence interval:

```
def Summarize(values, actual):
    std = thinkstats2.Std(values, mu=actual)
    cdf = thinkstats2.Cdf(values)
    ci = cdf.Percentile(5), cdf.Percentile(95)
    print('SE, CI', std, ci)
```

For the intercept, the estimated value 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, **options):
    size = len(slopes), len(xs)
    array = np.zeros(size)

    for i, (inter, slope) in enumerate(zip(inters, slopes)):
        fxs, fys = thinkstats2.FitLine(xs, inter, slope)
        array[i,] = fys

    xs = np.sort(xs)
    array = np.sort(array, axis=0)

    low = array[5,]
    high = array[95,]
    thinkplot.FillBetween(xs, low, high, **options)
```

`xs` is the sequence of mother's age. `inters` and `slopes` are the estimates parameters generated by `SamplingDistributions`.

`PlotConfidenceIntervals` generates a fitted line for each pairs of `inter` and `slope` and stores the results in an array with one row per simulated exper-

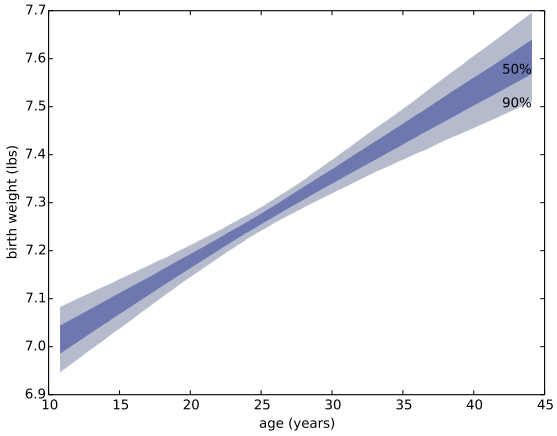

Figure 10.3: 50% and 90% confidence intervals showing variability in the fitted line due to sampling error of `inter` and `slope`.

iment. Then it sorts each column in the array, and selects the row with indices 5 and 95, which contain the 5th and 95th percentiles (this is a simplified version of the code for the case where there are 101 rows).

`FillBetween` draws a polygon that fills the space between two lines. Figure 10.3 shows the 50% and 90% confidence intervals. 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.

10.5 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":

```
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 less than half of 1% of variance in birth weight.

There is a simple relationship between the coefficient of determination and Pearson's correlation coefficient: $R^2 = \rho^2$. 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 so `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.

10.6 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 Section 9.5, when we tested the correlation between mother's age and birthweight. 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.

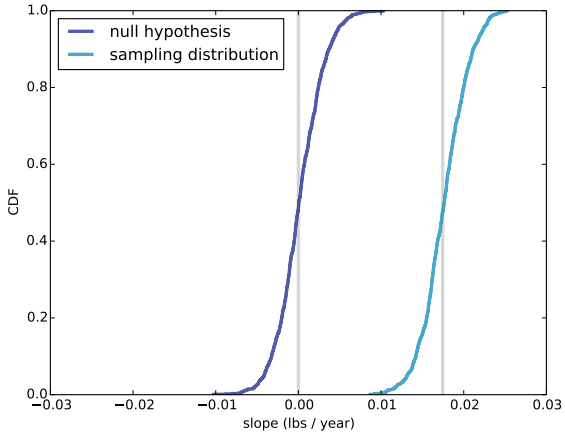

Figure 10.4: The sampling distribution of the estimated slope and the distribution of slopes generated under the null hypothesis. The vertical lines are at 0 and the observed slope, 0.017 lbs/year.

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 Section 10.4. 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 Section 10.4, 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 Section 14.4.

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 distributions 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 $p < 0.001$.

10.7 Weighted resampling

So far we have treated the NSFG data as if it were a representative sample, but as I mentioned in Section 1.2, 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 confidence intervals. As an example, I will estimate mean birth weight with and without sample weights.

We've already seen `SampleRows`, which chooses random rows from a DataFrame. `thinkstats2` also provide `ResampleRows`, which handles the case when the new sample is the same size as the original:

```
def ResampleRows(df):
    return SampleRows(df, len(df), replace=True)
```

Now we need to do the same thing using sample 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 = [thinkstats2.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.

10.8 Exercises

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?

A solution to this exercise is in `chap10ex_soln.ipynb`

10.9 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

Multiple regression

A linear least squares fit with [1]

11.1 StatsModels

11.2 Multiple regression

I won't cover those techniques here, but there are also simple ways to control for spurious relationships. For example, in the NSFG, we saw that first babies tend to be lighter than others (see Section 4.6). But birth weight is also correlated with the mother's age, and mothers of first babies tend to be younger than mothers of other babies.

So it may be that first babies are lighter because their mothers are younger. To control for the effect of age, we could divide the mothers into age groups and compare birth weights for first babies and others in each age group.

If the difference between first babies and others is the same in each age group as it was in the pooled data, we conclude that the difference is not related to age. If there is no difference, we conclude that the effect is entirely due to age. Or, if the difference is smaller, we can quantify how much of the effect is due to age.

Exercise 11.1 The NSFG data includes a variable named `agepreg` that records the age of the mother at the time of birth. Make a scatter plot of

[1] See `http://wikipedia.org/wiki/Simple_linear_regression`.

mother's age and baby's weight for each live birth. Can you see a relationship?

Compute a linear least-squares fit for these variables. What are the units of the estimated parameters $\hat{\text{inter}}$ and $\hat{\text{slope}}$? How would you summarize these results in a sentence or two?

Compute the average age for mothers of first babies and the average age of other mothers. Based on the difference in ages between the groups, how much difference do you expect in the mean birth weights? What fraction of the actual difference in birth weights is explained by the difference in ages?

You can download a solution to this problem from `http://thinkstats2.com/agemodel.py`. If you are curious about multivariate regression, you can run `http://thinkstats2.com/age_lm.py` which shows how to use the R statistical computing package from Python. But that's a whole other book.

11.3 Logistic regression

Chapter 12

Time series analysis

Chapter 13

Survival analysis

Chapter 14

Operations on distributions

14.1 Random Variables

A **random variable** represents a process that generates a random number. Random variables are usually written with a capital letter, like X. You can think of a random variable as a random number generator.

For example, here is a definition for a class that represents random variables:

```
class RandomVariable(object):
    """Parent class for all random variables."""
```

And here is a random variable with an exponential distribution:

```
class Exponential(RandomVariable):
    def __init__(self, lam):
        self.lam = lam

    def Generate(self):
        return random.expovariate(self.lam)
```

The `__init__` method takes the parameter, λ, and stores it as an attribute. The `Generate` method returns a random value from the exponential distribution with that parameter.

Each time you invoke `Generate`, you get a different value. The value you get is called a **random variate**, which is why many function names in the `random` module include the word "variate."

If I were just generating exponential variates, I would not bother to define a new class; I would use `random.expovariate`. But for other distributions it might be useful to use RandomVariable objects. For example, the Erlang distribution is an analytic distribution with parameters λ and k (see `http://wikipedia.org/wiki/Erlang_distribution`).

One way to generate values from an Erlang distribution is to add k values from an exponential distribution with the same λ. Here's an implementation:

```
class Erlang(RandomVariable):
    def __init__(self, lam, k):
        self.lam = lam
        self.k = k
        self.expo = Exponential(lam)

    def Generate(self):
        total = 0
        for i in range(self.k):
            total += self.expo.Generate()
        return total
```

The `__init__` method creates an Exponential object with the given parameter; then `Generate` uses it. In general, `__init__` can take any set of parameters and the `Generate` function can implement any random process.

14.2 Functions of random variables

Suppose we have two random variables, X and Y, represented by RandomVariable objects. How can we compute the distribution of the sum $Z = X + Y$?

One option is to write a RandomVariable object that generates the sum:

```
class Sum(RandomVariable):
    def __init__(X, Y):
        self.X = X
        self.Y = Y

    def Generate():
        return X.Generate() + Y.Generate()
```

Given any RandomVariables, X and Y, we can create a Sum object that represents Z. Then we can use a sample from Z to approximate CDF_Z.

This approach is simple and versatile, but not very efficient; we have to generate a large sample to estimate CDF_Z accurately, and even then it is not exact.

Exercise 14.1 If you are given Pmf objects, you can compute the distribution of the sum by enumerating all pairs of values:

```
for x in pmf_x.Values():
    for y in pmf_y.Values():
        z = x + y
```

Write a function that takes PMF_X and PMF_Y and returns a new Pmf that represents the distribution of the sum $Z = X + Y$.

Write a similar function that computes the PMF of $Z = max(X, Y)$.

14.3 Why normal?

I said earlier that normal distributions are amenable to analysis, but I didn't say why. One reason is that they are closed under linear transformation and convolution. To explain what that means, it will help to introduce some notation.

If the distribution of a random variable, 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 $X \sim \mathcal{N}(\mu, \sigma^2)$,

$$X' \sim \mathcal{N}(a\mu + b, a^2\sigma^2)$$

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

$$Z \sim \mathcal{N}(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2)$$

The other distributions we have looked at do not have these properties.

14.4 Central limit theorem

So far we have seen:

- If we add values drawn from normal distributions, the distribution of the sum is normal.
- If we add values drawn from other distributions, the sum does not generally have one of the analytic distributions we have seen.

But it turns out that if we add up a large number of values from almost any distribution, the distribution of the sum converges to normal.

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 is called the **Central Limit Theorem**. It is one of the most useful tools for statistical analysis, but it comes with caveats:

- The values have to be drawn independently.
- 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 number of values you need before you see convergence depends on the skewness of the distribution. Sums from an exponential distribution converge for small sample sizes. Sums from a lognormal distribution require much larger sizes.

The Central Limit Theorem explains, at least in part, the prevalence of normal distributions in the natural world. Most characteristics of animals and other life forms are affected by a large number of 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.

Exercise 14.2 If I draw a sample, $x_1..x_n$, independently from a distribution with finite mean μ and variance σ^2, what is the distribution of the sample mean:

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

As n increases, what happens to the variance of the sample mean? Hint: review Section 14.3.

Exercise 14.3 Choose a distribution (one of exponential, lognormal or Pareto) and choose values for the parameter(s). Generate samples with sizes 2, 4, 8, etc., and compute the distribution of their sums. Use a normal probability plot to see if the distribution is approximately normal. How many terms do you have to add to see convergence?

Exercise 14.4 Instead of the distribution of sums, compute the distribution of products; what happens as the number of terms increases? Hint: look at the distribution of the log of the products.

14.5 Exercises

Exercise 14.5 Write a definition for a class that represents a random variable with a Gumbel distribution (see `http://wikipedia.org/wiki/Gumbel_distribution`).

Exercise 14.6 Suppose I draw two values from a distribution; what is the distribution of the larger value? Write a RandomVariable class called `Max` that generates values from $Z = max(X, Y)$.

As the number of values increases, the distribution of the maximum converges on one of the extreme value distributions; see `http://wikipedia.org/wiki/Gumbel_distribution`.

Exercise 14.7 If $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$ and $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$, what is the distribution of Z = aX + bY?

Exercise 14.8 Let's see what happens when we add values from other distributions. Choose a pair of distributions (any two of exponential, normal, lognormal, and Pareto) and choose parameters that make their mean and variance similar.

Generate random numbers from these distributions and compute the distribution of their sums. Use the tests from Chapter 5 to see if the sum can be modeled by an analytic distribution.

Index