통계적 사고

파이썬과 R을 이용한 탐색적 자료 분석

Version 2.0.25

통계적 사고

파이썬과 R을 이용한 탐색적 자료 분석

Version 2.0.25

번역 및 저자: 이광춘
원저자: Allen B. Downey

xwMOOC http://www.xwmooc.net

서문

탐색적 자료 분석(EDA, Exploratory Data Analysis)에 대한 소개서로 작성되었다.

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

- Importing and cleaning: Whatever format the data is in, it usually takes some time and effort to read the data, clean and transform it, and check that everything made it through the translation process intact.
- Single variable explorations: I usually start by examining one variable at a time, finding out what the variables mean, looking at distributions of the values, and choosing appropriate summary statistics.
- Pair-wise explorations: To identify possible relationships between variables, I look at tables and scatter plots, and compute correlations and linear fits.
- Multivariate analysis: If there are apparent relationships between variables, I use multiple regression to add control variables and investigate more complex relationships.
- Estimation and hypothesis testing: When reporting statistical results, it is important to answer three questions: How big is the effect? How

much variability should we expect if we run the same measurement again? Is it possible that the apparent effect is due to chance?

- Visualization: During exploration, visualization is an important tool for finding possible relationships and effects. Then if an apparent effect holds up to scrutiny, visualization is an effective way to communicate results.

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

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

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

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

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

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

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

제 1 절 How I wrote this book

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

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

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

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

제 2 절 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 under Git's control is called a **repository**. GitHub is a hosting service that provides storage for Git repositories and a convenient web interface.

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

- You can create a copy of my repository on GitHub by pressing the Fork button. If you don't already have a GitHub account, you'll need to create one. After forking, you'll have your own repository on GitHub that you can use to keep track of code you write while working on this book. Then you can clone the repo, which means that you make a copy of the files on your computer.

- Or you could clone my repository. You don't need a GitHub account to do this, but you won't be able to write your changes back to GitHub.

- If you don't want to use Git at all, you can download the files in a Zip file using the button in the lower-right corner of the GitHub page.

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

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

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

- 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/`;
- StatsModels for regression and other statistical analysis, `http://statsmodels.sourceforge.net/`; and
- 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. If you have trouble installing them, I strongly recommend using Anaconda or one of the other Python distributions that include these packages.

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

Most exercises use Python scripts, but some also use the IPython notebook. If you have not used IPython notebook before, I suggest you start with the documentation at `http://ipython.org/ipython-doc/stable/notebook/notebook.html`.

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

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

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

—

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

Contributor List

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

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

- Lisa Downey and June Downey read an early draft and made many corrections and suggestions.
- Steven Zhang found several errors.
- Andy Pethan and Molly Farison helped debug some of the solutions, and Molly spotted several typos.
- Andrew Heine found an error in my error function.
- Dr. Nikolas Akerblom knows how big a Hyracotherium is.
- Alex Morrow clarified one of the code examples.
- Jonathan Street caught an error in the nick of time.
- Gábor Lipták found a typo in the book and the relay race solution.
- Many thanks to Kevin Smith and Tim Arnold for their work on plasTeX, which I used to convert this book to DocBook.
- George Caplan sent several suggestions for improving clarity.
- Julian Ceipek found an error and a number of typos.
- Stijn Debrouwere, Leo Marihart III, Jonathan Hammler, and Kent Johnson found errors in the first print edition.
- Dan Kearney found a typo.
- Jeff Pickhardt found a broken link and a typo.

- Jörg Beyer found typos in the book and made many corrections in the docstrings of the accompanying code.
- Tommie Gannert sent a patch file with a number of corrections.
- Alexander Gryzlov suggested a clarification in an exercise.
- Martin Veillette reported an error in one of the formulas for Pearson's correlation.
- Christoph Lendenmann submitted several errata.
- Haitao Ma noticed a typo and and sent me a note.
- Michael Kearney sent me many excellent suggestions.
- Alex Birch made a number of helpful suggestions.
- Lindsey Vanderlyn, Griffin Tschurwald, and Ben Small read an early version of this book and found many errors.
- John Roth, Carol Willing, and Carol Novitsky performed technical reviews of the book. They found many errors and made many helpful suggestions.
- Rohit Deshpande found a typesetting error.
- David Palmer sent many helpful suggestions and corrections.
- Erik Kulyk found many typos.
- Nir Soffer sent several excellent pull requests for both the book and the supporting code.
- Joanne Pratt found a number that was off by a factor of 10.

차 례

제 1 장

탐색적 자료 분석

이 책의 주요 논지는 실용적인 방법과 결합된 데이터가 질문에 대답하고, 불확실성하에서 의사결정을 안내하는 것이다.

한가지 사례로, 집사람과 함께 첫번째 아이를 기대할 때 전해들은 질문에 모디브를 얻어 한가지 사례 연구를 제시한다: 첫째 애기는 늦게 낳는 경향이 있을까요?

만약 이 질문을 구글에 검색하면, 상당한 토론글을 볼 수 있다. 몇몇 사람은 사실이라고하고, 다른 사람은 미신이라고 하고, 다른 사람은 첫째 얘들이 일찍 나온다고 애둘러 말하곤 한다.

많은 토론글에서, 사람들은 자신의 주장을 뒷받침하기 위해서 데이터를 제공한다. 다음과 같은 많은 사례를 찾아볼 수 있다.

> "최근에 첫째 아이를 출산한 내 친구 두명은 모두 자연분만 혹은 제왕절개하기 전에 예정일에서 거의 2주나 지났다."
>
> "첫째는 2주 늦게 나왔고 이제 생각하기에 둘째는 2주 빨리 나올것 같다.!!"
>
> "제 생각에는 사실이 아니라고 생각하는데 왜냐하면 언니가 엄마의 첫째인데 다른 많은 사촌과 마찬가지로 빨리 나왔기 때문이다."

이와 같은 보고를 **일화적 증거(anecdotal evidence)**라고 부르는데, 이유는 논문으로 출판되지 않고 대체로 개인적인 데이터에 기반하고 있기 때문이다. 일상적인 대화에서, 일화와 관련해서 잘못된 것은 없다. 그래서 인용한 사람을 콕 집어서 뭐라고 할 의도는 없다.

하지만, 좀더 설득적인 증거와, 좀더 신빙성있는 답을 원할지도 모른다. 이런 기준으로 본다면, 일화적 증거는 대체로 성공적이지 못하다. 왜냐하면:

- 적은 관측치(Small number of observations): 만약 첫째 아기에 대해서 임신 기간이 좀더 길다면, 아마도 차이는 자연적인 변동과 비교하여 적을 것이다. 이 경우에, 차이가 존재한다는 것을 확실히 하기 위해서는 많은 임신 사례를 비교해야할 것이다.

- 선택 편의(Selection bias): 임신기간에 관한 토론에 참가한 사람들은 첫째 아이가 늦게 태어났기 때문에 관심이 있을 수 있다. 이 경우에 데이터를 선택하는 과정이 결과를 왜곡할 수도 있다.

- 확증 편의(Confirmation bias): 주장을 믿는 사람들은 주장을 확증해주는 사례에 좀더 기여할 듯 하다. 주장에 의구심을 갖는 사람들은 반례를 좀더 들것 같다.

- 부정확(Inaccuracy): 일화는 종종 개인 이야기로, 기억이 부정확하고, 잘못 표현되며, 부정확하게 반복된다.

그렇다면, 어떻게 더 잘 할 수 있을까요?

제 1 절 통계적 접근방법

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.

제 2 절 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." See `http://cdc.gov/nchs/nsfg.htm`.

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. Ideally surveys would collect data from every member of the population, but that's seldom possible. Instead we collect data from a subset of the population called a **sample**. The people who participate in a survey are called **respondents**.

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

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

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

When working with this kind of data, it is important to be familiar with the **codebook**, which documents the design of the study, the survey questions, and the encoding of the responses. The codebook and user's guide for the NSFG data are available from `http://www.cdc.gov/nchs/nsfg/nsfg_cycle6.htm`

제 3 절 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 2.

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`; it is a gzip-compressed data file in plain text (ASCII), with fixed width columns. Each line in the file is a **record** that contains data about one pregnancy.

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

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

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

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

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

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

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

제 4 절 DataFrames

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

제 5 절 Variables

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

- `caseid` is the integer ID of the respondent.
- `prglngth` is the integer duration of the pregnancy in weeks.
- `outcome` is an integer code for the outcome of the pregnancy. The code 1 indicates a live birth.
- `pregordr` is a pregnancy serial number; for example, the code for a respondent's first pregnancy is 1, for the second pregnancy is 2, and so on.
- `birthord` is a serial number for live births; the code for a respondent's first child is 1, and so on. For outcomes other than live 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.

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

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

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

제 6 절 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 generate bogus results, like a 99-pound baby. The `replace` method replaces these values with `np.nan`, a special floating-point value that represents "not a number." The `inplace` flag tells `replace` to modify the existing Series rather than create a new one.

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

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

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

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

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

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

Not dot notation, like this:

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

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

제 7 절 Validation

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

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

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

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

```
>>> df.outcome.value_counts().sort_index()
1    9148
2    1862
3     120
```

```
4    1921
5     190
6     352
```

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

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

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

And here are the value counts:

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

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

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

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

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

제 8 절 Interpretation

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

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

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

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

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

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

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

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

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

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

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

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

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

제 9 절 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`.

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

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

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

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

Exercise 1.2 Create a file named `chap01ex.py` and write code that reads the respondent file, `2002FemResp.dat.gz`. You might want to start with a copy of `nsfg.py` and modify it.

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

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

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

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

Exercise 1.3 The best way to learn about statistics is to work on a project you are interested in. 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!

제 10 절 Glossary

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

- **population**: A group we are interested in studying. "Population" often refers to a group of people, but the term is used for other subjects, too.
- **cross-sectional study**: A study that collects data about a population at a particular point in time.
- **cycle**: In a repeated cross-sectional study, each repetition of the study is called a cycle.
- **longitudinal study**: A study that follows a population over time, collecting data from the same group repeatedly.
- **record**: In a dataset, a collection of information about a single person or other subject.
- **respondent**: A person who responds to a survey.
- **sample**: The subset of a population used to collect data.
- **representative**: A sample is representative if every member of the population has the same chance of being in the sample.
- **oversampling**: The technique of increasing the representation of a sub-population in order to avoid errors due to small sample sizes.
- **raw data**: Values collected and recorded with little or no checking, calculation or interpretation.
- **recode**: A value that is generated by calculation and other logic applied to raw data.
- **data cleaning**: Processes that include validating data, identifying errors, translating between data types and representations, etc.

제 2 장

Distributions

제 1 절 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.

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

제 2 절 Representing histograms

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

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

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

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

The bracket operator does the same thing:

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

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

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

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

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

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

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

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

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

제 3 절 Plotting histograms

For this book I wrote a module called `thinkplot.py` that provides functions for plotting Hists and other objects defined in `thinkstats2.py`. It is based

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

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

on `pyplot`, which is part of the `matplotlib` package. See Section 2 for information about installing `matplotlib`.

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

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

You can read the documentation for `thinkplot` at `http://greenteapress.com/thinkstats2/thinkplot.html`.

제 4 절 NSFG variables

Now let's get back to the data from the NSFG. The code in this chapter is in `first.py`. For information about downloading and working with this code, see Section 2.

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 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 use these variables to demonstrate some features of histograms.

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

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

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

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

그림 2.4: Histogram of pregnancy length in weeks.

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

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

Figure 2.1 shows the result. The most common value, called the **mode**, is 7 pounds. The distribution is approximately bell-shaped, which is the shape of the **normal** distribution, also called a **Gaussian** distribution. But unlike a true normal distribution, this distribution is asymmetric; it has a **tail** that extends farther to the left than to the right.

Figure 2.2 shows the histogram of `birthwgt_oz`, which is the ounces part of birth weight. In theory we expect this distribution to be **uniform**; that is, all values should have the same frequency. In fact, 0 is more common than the other values, and 1 and 15 are less common, probably because respondents round off birth weights that are close to an integer value.

Figure 2.3 shows the histogram of `agepreg`, the mother's age at the end of pregnancy. The mode is 21 years. The distribution is very roughly bell-shaped, but in this case the tail extends farther to the right than left; most mothers are in their 20s, fewer in their 30s.

Figure 2.4 shows the histogram of `prglngth`, the length of the pregnancy in weeks. By far the most common value is 39 weeks. The left tail is longer than the right; early babies are common, but pregnancies seldom go past 43 weeks, and doctors often intervene if they do.

제 5 절 Outliers

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

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

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

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

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

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

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

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

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

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

제 6 절 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]
```

그림 2.5: Histogram of pregnancy lengths.

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

Then I plotted their histograms on the same axis:

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

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

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

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

Finally, I adjust the axis to show only data between 27 and 46 weeks. Figure 2.5 shows the result.

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.

제 7 절 Summarizing distributions

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

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

Some of the characteristics we might want to report are:

- central tendency: Do the values tend to cluster around a particular point?
- modes: Is there more than one cluster?
- spread: How much variability is there in the values?
- tails: How quickly do the probabilities drop off as we move away from the modes?
- outliers: Are there extreme values far from the modes?

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

If you have a sample of `n` values, x_i, the mean, $\bar{x}$, is the sum of the values divided by the number of values; in other words

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

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

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

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

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

제 8 절 Variance

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

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

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

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

If you have prior experience, you might have seen a formula for variance with $n-1$ in the denominator, rather than `n`. This statistic is used to estimate the variance in a population using a sample. We will come back to this in Chapter 5.

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

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

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

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

제 9 절 Effect size

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

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

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

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

$$d = \frac{\bar{x_1} - \bar{x_2}}{s}$$

where $\bar{x_1}$ and $\bar{x_2}$ are the means of the groups and s is the "pooled standard deviation". Here's the Python code that computes Cohen's *d*:

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

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

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

In this example, the difference in means is 0.029 standard deviations, which is small. To put that in perspective, the difference in height between men and women is about 1.7 standard deviations (see `https://en.wikipedia.org/wiki/Effect_size`).

제 10 절 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.

제 11 절 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. Follow the instructions and fill in the answers.

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

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

Exercise 2.3 The mode of a distribution is the most frequent value; see `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 Using the variable `totalwgt_lb`, investigate whether first babies are lighter or heavier than others. Compute Cohen's d to quantify the difference between the groups. How does it compare to the difference in pregnancy length?

제 12 절 Glossary

- distribution: The values that appear in a sample and the frequency of each.
- histogram: A mapping from values to frequencies, or a graph that shows this mapping.
- frequency: The number of times a value appears in a sample.
- mode: The most frequent value in a sample, or one of the most frequent values.
- normal distribution: An idealization of a bell-shaped distribution; also known as a Gaussian distribution.
- uniform distribution: A distribution in which all values have the same frequency.
- tail: The part of a distribution at the high and low extremes.
- central tendency: A characteristic of a sample or population; intuitively, it is an average or typical value.
- outlier: A value far from the central tendency.
- spread: A measure of how spread out the values in a distribution are.
- summary statistic: A statistic that quantifies some aspect of a distribution, like central tendency or spread.
- variance: A summary statistic often used to quantify spread.
- standard deviation: The square root of variance, also used as a measure of spread.
- effect size: A summary statistic intended to quantify the size of an effect like a difference between groups.
- clinically significant: A result, like a difference between groups, that is relevant in practice.

제 3 장

Probability mass functions

The code for this chapter is in `probability.py`. For information about downloading and working with this code, see Section 2.

제 1 절 Pmfs

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

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

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

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

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

The Pmf is normalized so total probability is 1.

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

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

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

The bracket operator is equivalent:

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

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

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

Or you can multiply a probability by a factor:

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

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

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

To renormalize, call `Normalize`:

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

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

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

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

제 2 절 Plotting PMFs

`thinkplot` provides two ways to plot Pmfs:

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

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

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

By plotting the PMF instead of the histogram, we can compare the two distributions without being mislead 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).

Here's the code that generates Figure 3.1:

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

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

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

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

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

제 3 절 Other visualizations

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

In the NSFG data, the biggest differences in the distributions are near the mode. So it makes sense to zoom in on that part of the graph, and 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 as a bar chart. This figure makes the pattern clearer: first babies are less likely to be born in week 39, and somewhat more likely to be born in weeks 41 and 42.

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

제 4 절 The class size paradox

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

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 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.

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

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

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

    new_pmf.Normalize()
    return new_pmf
```

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

Now we can plot the actual and observed distributions:

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

Figure 3.3 shows the result. In the biased distribution there are fewer small classes and more large ones. The mean of the biased distribution is 29.1, almost 25% higher than the actual mean.

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

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

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

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

    new_pmf.Normalize()
    return new_pmf
```

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

제 5 절 DataFrame indexing

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

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

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

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

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

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

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

```
>>> df['A']
a   -0.143510
```

```
b   -1.489647
c   -0.074350
d   -1.369968
Name: A, dtype: float64
```

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

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

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

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

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

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

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

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

Or by integer position:

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

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

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

제 6 절 Exercises

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

Exercise 3.1 Something like the class size paradox appears if you survey children and ask how many children are in their family. Families with many children are more likely to appear in your sample, and families with no children have no chance to be in the sample.

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

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

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

Exercise 3.2 In Section 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 of a bias similar to the effect of class size. The race was unusual in two ways: it used a staggered start, so teams started at different times; also, many teams included runners at different levels of ability.

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

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

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

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

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

제 7 절 Glossary

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

- probability: A frequency expressed as a fraction of the sample size.
- normalization: The process of dividing a frequency by a sample size to get a probability.
- index: In a pandas DataFrame, the index is a special column that contains the row labels.

제 4 장

Cumulative distribution functions

The code for this chapter is in `cumulative.py`. For information about downloading and working with this code, see Section 2.

제 1 절 The limits of PMFs

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

For example, we might be interested in the distribution of birth weights. In the NSFG data, the variable `totalwgt_lb` records weight at birth in pounds. Figure 4.1 shows the PMF of these values for first babies and others.

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

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

These problems can be mitigated by binning the data; that is, dividing the range of values into non-overlapping intervals and counting the number of

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

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.

제 2 절 Percentiles

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

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

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

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

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

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

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

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

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

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

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

제 3 절 CDFs

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

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

Here's what that looks like as a function that takes a sequence, `sample`, and a value, `x`:

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

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

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

그림 4.2: Example of a CDF.

그림 4.3: CDF of pregnancy length.

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

$$CDF(0) = 0$$

$$CDF(1) = 0.2$$
$$CDF(2) = 0.6$$
$$CDF(3) = 0.8$$
$$CDF(4) = 0.8$$
$$CDF(5) = 1$$

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

Figure 4.2 is a graphical representation of this CDF. The CDF of a sample is a step function.

제 4 절 Representing CDFs

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

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

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

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

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

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

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

Figure 4.3 shows the result. One way to read a CDF is to look up percentiles. For example, it looks like about 10% of pregnancies are shorter than 36 weeks, and about 90% are shorter than 41 weeks. The CDF also provides a visual representation of the shape of the distribution. Common values appear as steep or vertical sections of the CDF; in this example, the mode at 39 weeks is apparent. There are few values below 30 weeks, so the CDF in this range is flat.

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.

제 5 절 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. Compared to Figure 4.1, this figure makes the shape of the distributions, and the differences between them, much clearer. We can see that first babies are slightly lighter throughout the distribution, with a larger discrepancy above the mean.

제 6 절 Percentile-based statistics

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

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

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

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

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

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

More generally, percentiles are often used to summarize the shape of a distribution. For example, the distribution of income is often reported in "quintiles"; that is, it is split at the 20th, 40th, 60th and 80th percentiles. Other distributions are divided into 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`.

제 7 절 Random numbers

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

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

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

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

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

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

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

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

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

Figure 4.5 shows the result. The CDF is approximately a straight line, which means that the distribution is uniform.

That outcome 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% is below the 20th percentile, and so on, exactly as we should expect.

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

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

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

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

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

제 8 절 Comparing percentile ranks

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

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

제 9 절 Exercises

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

Exercise 4.1 How much did you weigh at birth? If you don't know, call your mother or someone else who knows. Using the NSFG data (all live births), compute the distribution of birth weights and use it to find your percentile rank. If you were a first baby, find your percentile rank in the distribution for first babies. Otherwise use the distribution for others. If you are in the 90th percentile or higher, call your mother back and apologize.

Exercise 4.2 The numbers generated by `random.random` are supposed to be uniform between 0 and 1; that is, every value in the range should have the same probability.

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

제 10 절 Glossary

- percentile rank: The percentage of values in a distribution that are less than or equal to a given value.
- percentile: The value associated with a given percentile rank.
- cumulative distribution function (CDF): A function that maps from values to their cumulative probabilities. CDF(x) is the fraction of the sample less than or equal to x.
- inverse CDF: A function that maps from a cumulative probability, p, to the corresponding value.
- median: The 50th percentile, often used as a measure of central tendency.
- interquartile range: The difference between the 75th and 25th percentiles, used as a measure of spread.
- quantile: A sequence of values that correspond to equally spaced percentile ranks; for example, the quartiles of a distribution are the 25th, 50th and 75th percentiles.
- replacement: A property of a sampling process. "With replacement" means that the same value can be chosen more than once; "without replacement" means that once a value is chosen, it is removed from the population.

제 5 장

Estimation

The code for this chapter is in `estimation.py`. For information about downloading and working with this code, see Section 2.

제 1 절 The estimation game

Let's play a game. I think of a distribution, and you have to guess what it is. I'll give you two hints: it's a normal distribution, and here's a random sample drawn from it:

```
[-0.441, 1.774, -0.101, -1.138, 2.975, -2.138]
```

What do you think is the mean parameter, μ, of this distribution?

One choice is to use the sample mean, $\bar{x}$, as an estimate of μ. In this example, $\bar{x}$ is 0.155, so it would be reasonable to guess $\mu = 0.155$. This process is called **estimation**, and the statistic we used (the sample mean) is called an **estimator**.

Using the sample mean to estimate μ is so obvious that it is hard to imagine a reasonable alternative. But suppose we change the game by introducing outliers.

I'm thinking of a distribution. It's a normal distribution, and here's a sample that was collected by an unreliable surveyor who occasionally puts the decimal point in the wrong place.

```
[-0.441, 1.774, -0.101, -1.138, 2.975, -213.8]
```

Now what's your estimate of μ? If you use the sample mean, your guess is -35.12. Is that the best choice? What are the alternatives?

One option is to identify and discard outliers, 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.

제 2 절 Guess the variance

I'm thinking of a distribution. It's a normal distribution, and here's a (familiar) sample:

```
[-0.441, 1.774, -0.101, -1.138, 2.975, -2.138]
```

What do you think is the variance, σ^2, of my distribution? Again, the obvious choice is to use the sample variance, S^2, as an estimator.

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

For large samples, S^2 is an adequate estimator, but for small samples it tends to be too low. Because of this unfortunate property, it is called a **biased** estimator. An estimator is **unbiased** if the expected total (or mean) error, after many iterations of the estimation game, is 0.

Fortunately, there is another simple statistic that is an unbiased estimator of σ^2:

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

For an explanation of why S^2 is biased, and a proof that S_{n-1}^2 is unbiased, see `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_{n-1}^2, and the symbol S^2 is used for either or both.

Here is a function that simulates the estimation game and tests the performance of S^2 and S_{n-1}^2:

```
def Estimate2(n=7, m=1000):
    mu = 0
    sigma = 1

    estimates1 = []
    estimates2 = []
    for _ in range(m):
        xs = [random.gauss(mu, sigma) for i in range(n)]
        biased = np.var(xs)
        unbiased = np.var(xs, ddof=1)
        estimates1.append(biased)
        estimates2.append(unbiased)

    print('mean error biased', MeanError(estimates1, sigma**2))
    print('mean error unbiased', MeanError(estimates2, sigma**2))
```

Again, `n` is the sample size and `m` is the number of times we play the game. `np.var` computes S^2 by default and S_{n-1}^2 if you provide the argument `ddof=1`, which stands for "delta degrees of freedom." I won't explain that term, but you can read about it at `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.

After you choose an estimator with appropriate properties, and use it to generate an estimate, the next step is to characterize the uncertainty of the estimate, which is the topic of the next section.

제 3 절 Sampling distributions

Suppose you are a scientist studying gorillas in a wildlife preserve. You want to know the average weight of the adult female gorillas in the preserve. To weigh them, you have to tranquilize them, which is dangerous, expensive, and possibly harmful to the gorillas. But if it is important to obtain this information, it might be acceptable to weigh a sample of 9 gorillas. Let's assume that the population of the preserve is well known, so we can choose a representative sample of adult females. We could use the sample mean, $\bar{x}$, to estimate the unknown population mean, μ.

Having weighed 9 female gorillas, you might find $\bar{x} = 90$ kg and sample standard deviation, $S = 7.5$ kg. The sample mean is an unbiased estimator of μ, and in the long run it minimizes MSE. So if you report a single estimate that summarizes the results, you would report 90 kg.

But how confident should you be in this estimate? If you only weigh $n = 9$ gorillas out of a much larger population, you might be unlucky and choose the 9 heaviest gorillas (or the 9 lightest ones) just by chance. Variation in the estimate caused by random selection is called **sampling error**.

To quantify sampling error, we can simulate the sampling process with hypothetical values of μ and σ, and see how much $\bar{x}$ varies.

그림 5.1: 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.Cdf(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 5.1. 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.

제 4 절 Sampling bias

Suppose that instead of the weight of gorillas in a nature preserve, you want to know the average weight of women in the city where you live. It is unlikely that you would be allowed to choose a representative sample of women and weigh them.

A simple alternative would be "telephone sampling;" that is, you could choose random numbers from the phone book, call and ask to speak to an adult woman, and ask how much she weighs.

Telephone sampling has obvious limitations. For example, the sample is limited to people whose telephone numbers are listed, so it eliminates people without phones (who might be poorer than average) and people with unlisted numbers (who might be richer). Also, if you call home telephones during the day, you are less likely to sample people with jobs. And if you only sample the person who answers the phone, you are less likely to sample people who share a phone line.

If factors like income, employment, and household size are related to weight—and it is plausible that they are—the results of your survey would be affected one way or another. This problem is called **sampling bias** because it is a property of the sampling process.

This sampling process is also vulnerable to self-selection, which is a kind of sampling bias. Some people will refuse to answer the question, and if the tendency to refuse is related to weight, that would affect the results.

Finally, if you ask people how much they weigh, rather than weighing them, the results might not be accurate. Even helpful respondents might round up or down if they are uncomfortable with their actual weight. And not all respondents are helpful. These inaccuracies are examples of **measurement error**.

When you report an estimated quantity, it is useful to report standard error, or a confidence interval, or both, in order to quantify sampling error. But it is also important to remember that sampling error is only one source of error, and often it is not the biggest.

제 5 절 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 (see `http://wikipedia.org/wiki/Exponential_distribution#Maximum_likelihood`). So if you want to maximize your chance of guessing λ exactly, L is the way to go.

But we know that $\bar{x}$ is not robust in the presence of outliers, so we expect L to have the same problem.

We can choose an alternative based on the sample median. The median of an exponential distribution is $\ln(2)/\lambda$, so working backwards again, we can define an estimator

$$L_m = \ln(2)/m$$

where m is the sample median.

To test the performance of these estimators, we can simulate the sampling process:

```
def Estimate3(n=7, m=1000):
    lam = 2

    means = []
    medians = []
    for _ in range(m):
        xs = np.random.exponential(1.0/lam, n)
        L = 1 / np.mean(xs)
        Lm = math.log(2) / thinkstats2.Median(xs)
        means.append(L)
        medians.append(Lm)

    print('rmse L', RMSE(means, lam))
    print('rmse Lm', RMSE(medians, lam))
    print('mean error L', MeanError(means, lam))
    print('mean error Lm', MeanError(medians, lam))
```

When I run this experiment with $\lambda = 2$, the RMSE of L is 1.1. For the median-based estimator L_m, RMSE is 1.8. We can't tell from this experiment whether L minimizes MSE, but at least it seems better than L_m.

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

제 6 절 Exercises

For the following exercises, you might want to start with a copy of `estimation.py`. Solutions are in `chap08soln.py`

Exercise 5.1 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 5.2 Suppose you draw a sample with size $n = 10$ from an exponential distribution with $\lambda = 2$. Simulate this experiment 1000 times and plot the sampling distribution of the estimate L. Compute the standard error of the estimate and the 90% confidence interval.

Repeat the experiment with a few different values of n and make a plot of standard error versus n.

Exercise 5.3 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`?

제 7 절 Glossary

- estimation: The process of inferring the parameters of a distribution from a sample.
- estimator: A statistic used to estimate a parameter.
- mean squared error (MSE): A measure of estimation error.
- root mean squared error (RMSE): The square root of MSE, a more meaningful representation of typical error magnitude.

- maximum likelihood estimator (MLE): An estimator that computes the point estimate most likely to be correct.
- bias (of an estimator): The tendency of an estimator to be above or below the actual value of the parameter, when averaged over repeated experiments.
- sampling error: Error in an estimate due to the limited size of the sample and variation due to chance.
- sampling bias: Error in an estimate due to a sampling process that is not representative of the population.
- measurement error: Error in an estimate due to inaccuracy collecting or recording data.
- sampling distribution: The distribution of a statistic if an experiment is repeated many times.
- standard error: The RMSE of an estimate, which quantifies variability due to sampling error (but not other sources of error).
- confidence interval: An interval that represents the expected range of an estimator if an experiment is repeated many times.

제 6 장

Regression

The linear least squares fit in the previous chapter is an example of **regression**, which is the more general problem of fitting any kind of model to any kind of data. This use of the term "regression" is a historical accident; it is only indirectly related to the original meaning of the word.

The goal of regression analysis is to describe the relationship between one set of variables, called the **dependent variables**, and another set of variables, called independent or **explanatory variables**.

In the previous chapter we used mother's age as an explanatory variable to predict birth weight as a dependent variable. When there is only one dependent and one explanatory variable, that's **simple regression**. In this chapter, we move on to **multiple regression**, with more than one explanatory variable. If there is more than one dependent variable, that's multivariate regression.

If the relationship between the dependent and explanatory variable is linear, that's **linear regression**. For example, if the dependent variable is y and the explanatory variables are x_1 and x_2, we would write the following linear regression model:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

where β_0 is the intercept, β_1 is the parameter associated with x_1, β_2 is the parameter associated with x_2, and ε is the residual due to random variation or other unknown factors.

Given a sequence of values for y and sequences for x_1 and x_2, we can find the parameters, β_0, β_1, and β_2, that minimize the sum of ε^2. This process is called **ordinary least squares**. The computation is similar to thinkstats2.LeastSquare, but generalized to deal with more than one explanatory variable. You can find the details at https://en.wikipedia.org/wiki/Ordinary_least_squares

The code for this chapter is in regression.py. For information about downloading and working with this code, see Section 2.

제 1 절 StatsModels

In the previous chapter I presented thinkstats2.LeastSquares, an implementation of simple linear regression intended to be easy to read. For multiple regression we'll switch to StatsModels, a Python package that provides several forms of regression and other analyses. If you are using Anaconda, you already have StatsModels; otherwise you might have to install it.

As an example, I'll run the model from the previous chapter with StatsModels:

```
import statsmodels.formula.api as smf

live, firsts, others = first.MakeFrames()
formula = 'totalwgt_lb ~ agepreg'
model = smf.ols(formula, data=live)
results = model.fit()
```

statsmodels provides two interfaces (APIs); the "formula" API uses strings to identify the dependent and explanatory variables. It uses a syntax called patsy; in this example, the ~ operator separates the dependent variable on the left from the explanatory variables on the right.

smf.ols takes the formula string and the DataFrame, live, and returns an OLS object that represents the model. The name ols stands for "ordinary least squares."

The fit method fits the model to the data and returns a RegressionResults object that contains the results.

The results are also available as attributes. params is a Series that maps from variable names to their parameters, so we can get the intercept and slope like this:

```
inter = results.params['Intercept']
slope = results.params['agepreg']
```

The estimated parameters are 6.83 and 0.0175, the same as from `LeastSquares`.

`pvalues` is a Series that maps from variable names to the associated p-values, so we can check whether the estimated slope is statistically significant:

```
    slope_pvalue = results.pvalues['agepreg']
```

The p-value associated with `agepreg` is `5.7e-11`, which is less than 0.001, as expected.

`results.rsquared` contains R^2, which is 0.0047. `results` also provides `f_pvalue`, which is the p-value associated with the model as a whole, similar to testing whether R^2 is statistically significant.

And `results` provides `resid`, a sequence of residuals, and `fittedvalues`, a sequence of fitted values corresponding to `agepreg`.

The results object provides `summary()`, which represents the results in a readable format.

```
    print(results.summary())
```

But it prints a lot of information that is not relevant (yet), so I use a simpler function called `SummarizeResults`. Here are the results of this model:

```
Intercept       6.83    (0)
agepreg         0.0175  (5.72e-11)
R^2 0.004738
Std(ys) 1.408
Std(res) 1.405
```

`Std(ys)` is the standard deviation of the dependent variable, which is the RMSE if you have to guess birth weights without the benefit of any explanatory variables. `Std(res)` is the standard deviation of the residuals, which is the RMSE if your guesses are informed by the mother's age. As we have already seen, knowing the mother's age provides no substantial improvement to the predictions.

제 2 절 Multiple regression

In Section 5 we saw that first babies tend to be lighter than others, and this effect is statistically significant. But it is a strange result because there is no obvious mechanism that would cause first babies to be lighter. So we might wonder whether this relationship is **spurious**.

In fact, there is a possible explanation for this effect. We have seen that birth weight depends on mother's age, and we might expect that mothers of first babies are younger than others.

With a few calculations we can check whether this explanation is plausible. Then we'll use multiple regression to investigate more carefully. First, let's see how big the difference in weight is:

```
diff_weight = firsts.totalwgt_lb.mean() - others.totalwgt_lb.mean()
```

First babies are 0.125 lbs lighter, or 2 ounces. And the difference in ages:

```
diff_age = firsts.agepreg.mean() - others.agepreg.mean()
```

The mothers of first babies are 3.59 years younger. Running the linear model again, we get the change in birth weight as a function of age:

```
results = smf.ols('totalwgt_lb ~ agepreg', data=live).fit()
slope = results.params['agepreg']
```

The slope is 0.175 pounds per year. If we multiply the slope by the difference in ages, we get the expected difference in birth weight for first babies and others, due to mother's age:

```
slope * diff_age
```

The result is 0.063, just about half of the observed difference. So we conclude, tentatively, that the observed difference in birth weight can be partly explained by the difference in mother's age.

Using multiple regression, we can explore these relationships more systematically.

```
    live['isfirst'] = live.birthord == 1
    formula = 'totalwgt_lb ~ isfirst'
    results = smf.ols(formula, data=live).fit()
```

The first line creates a new column named `isfirst` that is True for first babies and false otherwise. Then we fit a model using `isfirst` as an explanatory variable.

Here are the results:

```
Intercept         7.33   (0)
isfirst[T.True]  -0.125  (2.55e-05)
R^2 0.00196
```

Because `isfirst` is a boolean, `ols` treats it as a **categorical variable**, which means that the values fall into categories, like True and False, and should not be treated as numbers. The estimated parameter is the effect on birth weight when `isfirst` is true, so the result, -0.125 lbs, is the difference in birth weight between first babies and others.

The slope and the intercept are statistically significant, which means that they were unlikely to occur by chance, but the the R^2 value for this model is small, which means that `isfirst` doesn't account for a substantial part of the variation in birth weight.

The results are similar with `agepreg`:

```
Intercept       6.83    (0)
agepreg         0.0175  (5.72e-11)
R^2 0.004738
```

Again, the parameters are statistically significant, but R^2 is low.

These models confirm results we have already seen. But now we can fit a single model that includes both variables. With the formula `totalwgt_lb ~ isfirst + agepreg`, we get:

```
Intercept         6.91    (0)
isfirst[T.True] -0.0698  (0.0253)
agepreg           0.0154  (3.93e-08)
R^2 0.005289
```

In the combined model, the parameter for `isfirst` is smaller by about half, which means that part of the apparent effect of `isfirst` is actually accounted for by `agepreg`. And the p-value for `isfirst` is about 2.5%, which is on the border of statistical significance.

R^2 for this model is a little higher, which indicates that the two variables together account for more variation in birth weight than either alone (but not by much).

제 3 절 Nonlinear relationships

Remembering that the contribution of `agepreg` might be nonlinear, we might consider adding a variable to capture more of this relationship. One option is to create a column, `agepreg2`, that contains the squares of the ages:

```
    live['agepreg2'] = live.agepreg**2
    formula = 'totalwgt_lb ~ isfirst + agepreg + agepreg2'
```

Now by estimating parameters for `agepreg` and `agepreg2`, we are effectively fitting a parabola:

```
Intercept         5.69     (1.38e-86)
isfirst[T.True] -0.0504    (0.109)
agepreg           0.112    (3.23e-07)
agepreg2        -0.00185   (8.8e-06)
R^2 0.007462
```

The parameter of agepreg2 is negative, so the parabola curves downward, which is consistent with the shape of the lines in Figure **??**.

The quadratic model of agepreg accounts for more of the variability in birth weight; the parameter for isfirst is smaller in this model, and no longer statistically significant.

Using computed variables like agepreg2 is a common way to fit polynomials and other functions to data. This process is still considered linear regression, because the dependent variable is a linear function of the explanatory variables, regardless of whether some variables are nonlinear functions of others.

The following table summarizes the results of these regressions:

	isfirst	agepreg	agepreg2	R^2
Model 1	-0.125 *	–	–	0.002
Model 2	–	0.0175 *	–	0.0047
Model 3	-0.0698 (0.025)	0.0154 *	–	0.0053
Model 4	-0.0504 (0.11)	0.112 *	-0.00185 *	0.0075

The columns in this table are the explanatory variables and the coefficient of determination, R^2. Each entry is an estimated parameter and either a p-value in parentheses or an asterisk to indicate a p-value less that 0.001.

We conclude that the apparent difference in birth weight is explained, at least in part, by the difference in mother's age. When we include mother's age in the model, the effect of isfirst gets smaller, and the remaining effect might be due to chance.

In this example, mother's age acts as a **control variable**; including agepreg in the model "controls for" the difference in age between first-time mothers and others, making it possible to isolate the effect (if any) of isfirst.

제 4 절 Data mining

So far we have used regression models for explanation; for example, in the previous section we discovered that an apparent difference in birth weight is actually due to a difference in mother's age. But the R^2 values of those models is very low, which means that they have little predictive power. In this section we'll try to do better.

Suppose one of your co-workers is expecting a baby and there is an office pool to guess the baby's birth weight (if you are not familiar with betting pools, see `https://en.wikipedia.org/wiki/Betting_pool`).

Now suppose that you *really* want to win the pool. What could you do to improve your chances? Well, the NSFG dataset includes 244 variables about each pregnancy and another 3087 variables about each respondent. Maybe some of those variables have predictive power. To find out which ones are most useful, why not try them all?

Testing the variables in the pregnancy table is easy, but in order to use the variables in the respondent table, we have to match up each pregnancy with a respondent. In theory we could iterate through the rows of the pregnancy table, use the `caseid` to find the corresponding respondent, and copy the values from the correspondent table into the pregnancy table. But that would be slow.

A better option is to recognize this process as a **join** operation as defined in SQL and other relational database languages (see `https://en.wikipedia.org/wiki/Join_(SQL)`). Join is implemented as a DataFrame method, so we can perform the operation like this:

```
    live = live[live.prglngth>30]
    resp = chap01soln.ReadFemResp()
    resp.index = resp.caseid
    join = live.join(resp, on='caseid', rsuffix='_r')
```

The first line selects records for pregnancies longer than 30 weeks, assuming that the office pool is formed several weeks before the due date.

The next line reads the respondent file. The result is a DataFrame with integer indices; in order to look up respondents efficiently, I replace `resp.index` with `resp.caseid`.

The `join` method is invoked on `live`, which is considered the "left" table, and passed `resp`, which is the "right" table. The keyword argument `on` indicates the variable used to match up rows from the two tables.

In this example some column names appear in both tables, so we have to provide `rsuffix`, which is a string that will be appended to the names of overlapping columns from the right table. For example, both tables have a column named race that encodes the race of the respondent. The result of the join contains two columns named race and race_r.

The pandas implementation is fast. Joining the NSFG tables takes less than a second on an ordinary desktop computer. Now we can start testing variables.

```
t = []
for name in join.columns:
    try:
        if join[name].var() < 1e-7:
            continue

        formula = 'totalwgt_lb ~ agepreg + ' + name
        model = smf.ols(formula, data=join)
        if model.nobs < len(join)/2:
            continue

        results = model.fit()
    except (ValueError, TypeError):
        continue

    t.append((results.rsquared, name))
```

For each variable we construct a model, compute R^2, and append the results to a list. The models all include agepreg, since we already know that it has some predictive power.

I check that each explanatory variable has some variability; otherwise the results of the regression are unreliable. I also check the number of observations for each model. Variables that contain a large number of nans are not good candidates for prediction.

For most of these variables, we haven't done any cleaning. Some of them are encoded in ways that don't work very well for linear regression. As a result, we might overlook some variables that would be useful if they were cleaned properly. But maybe we will find some good candidates.

제 5 절 Prediction

The next step is to sort the results and select the variables that yield the highest values of R^2.

```
    t.sort(reverse=True)
    for mse, name in t[:30]:
        print(name, mse)
```

The first variable on the list is `totalwgt_lb`, followed by `birthwgt_lb`. Obviously, we can't use birth weight to predict birth weight.

Similarly `prglngth` has useful predictive power, but for the office pool we assume pregnancy length (and the related variables) are not known yet.

The first useful predictive variable is `babysex` which indicates whether the baby is male or female. In the NSFG dataset, boys are about 0.3 lbs heavier. So, assuming that the sex of the baby is known, we can use it for prediction.

Next is `race`, which indicates whether the respondent is white, black, or other. As an explanatory variable, race can be problematic. In datasets like the NSFG, race is correlated with many other variables, including income and other socioeconomic factors. In a regression model, race acts as a **proxy variable**, so apparent correlations with race are often caused, at least in part, by other factors.

The next variable on the list is `nbrnaliv`, which indicates whether the pregnancy yielded multiple births. Twins and triplets tend to be smaller than other babies, so if we know whether our hypothetical co-worker is expecting twins, that would help.

Next on the list is `paydu`, which indicates whether the respondent owns her home. It is one of several income-related variables that turn out to be predictive. In datasets like the NSFG, income and wealth are correlated with just about everything. In this example, income is related to diet, health, health care, and other factors likely to affect birth weight.

Some of the other variables on the list are things that would not be known until later, like `bfeedwks`, the number of weeks the baby was breast fed. We can't use these variables for prediction, but you might want to speculate on reasons `bfeedwks` might be correlated with birth weight.

Sometimes you start with a theory and use data to test it. Other times you start with data and go looking for possible theories. The second approach, which this section demonstrates, is called **data mining**. An advantage of data mining is that it can discover unexpected patterns. A hazard is that many of the patterns it discovers are either random or spurious.

Having identified potential explanatory variables, I tested a few models and settled on this one:

```
    formula = ('totalwgt_lb ~ agepreg + C(race) + babysex==1 + '
               'nbrnaliv>1 + paydu==1 + totincr')
    results = smf.ols(formula, data=join).fit()
```

This formula uses some syntax we have not seen yet: `C(race)` tells the formula parser (Patsy) to treat race as a categorical variable, even though it is encoded numerically.

The encoding for `babysex` is 1 for male, 2 for female; writing `babysex==1` converts it to boolean, True for male and false for female.

Similarly `nbrnaliv>1` is True for multiple births and `paydu==1` is True for respondents who own their houses.

`totincr` is encoded numerically from 1-14, with each increment representing about $5000 in annual income. So we can treat these values as numerical, expressed in units of $5000.

Here are the results of the model:

```
Intercept               6.63    (0)
C(race)[T.2]            0.357   (5.43e-29)
C(race)[T.3]            0.266   (2.33e-07)
babysex == 1[T.True]    0.295   (5.39e-29)
nbrnaliv > 1[T.True]   -1.38    (5.1e-37)
paydu == 1[T.True]      0.12    (0.000114)
agepreg                 0.00741 (0.0035)
totincr                 0.0122  (0.00188)
```

The estimated parameters for race are larger than I expected, especially since we control for income. The encoding is 1 for black, 2 for white, and 3 for other. Babies of black mothers are lighter than babies of other races by 0.27–0.36 lbs.

As we've already seen, boys are heavier by about 0.3 lbs; twins and other multiplets are lighter by 1.4 lbs.

People who own their homes have heavier babies by about 0.12 lbs, even when we control for income. The parameter for mother's age is smaller than what we saw in Section 2, which suggests that some of the other variables are correlated with age, probably including `paydu` and `totincr`.

All of these variables are statistically significant, some with very low p-values, but R^2 is only 0.06, still quite small. RMSE without using the model is 1.27 lbs; with the model it drops to 1.23. So your chance of winning the pool is not substantially improved. Sorry!

제 6 절 Logistic regression

In the previous examples, some of the explanatory variables were numerical and some categorical (including boolean). But the dependent variable was always numerical.

Linear regression can be generalized to handle other kinds of dependent variables. If the dependent variable is boolean, the generalized model is called **logistic regression**. If the dependent variable is an integer count, it's called **Poisson regression**.

As an example of logistic regression, let's consider a variation on the office pool scenario. Suppose a friend of yours is pregnant and you want to predict whether the baby is a boy or a girl. You could use data from the NSFG to find factors that affect the "sex ratio", which is conventionally defined to be the probability of having a boy.

If you encode the dependent variable numerically, for example 0 for a girl and 1 for a boy, you could apply ordinary least squares, but there would be problems. The linear model might be something like this:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

Where y is the dependent variable, and x_1 and x_2 are explanatory variables. Then we could find the parameters that minimize the residuals.

The problem with this approach is that it produces predictions that are hard to interpret. Given estimated parameters and values for x_1 and x_2, the model might predict $y = 0.5$, but the only meaningful values of y are 0 and 1.

It is tempting to interpret a result like that as a probability; for example, we might say that a respondent with particular values of x_1 and x_2 has a 50% chance of having a boy. But it is also possible for this model to predict $y = 1.1$ or $y = -0.1$, and those are not valid probabilities.

Logistic regression avoids this problem by expressing predictions in terms of **odds** rather than probabilities. If you are not familiar with odds, "odds in favor" of an event is the ratio of the probability it will occur to the probability that it will not.

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

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

```
    o = p / (1-p)
```

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

```
    p = o / (o+1)
```

Logistic regression is based on the following model:

$$\log o = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$$

Where o is the odds in favor of a particular outcome; in the example, o would be the odds of having a boy.

Suppose we have estimated the parameters β_0, β_1, and β_2 (I'll explain how in a minute). And suppose we are given values for x_1 and x_2. We can compute the predicted value of $\log o$, and then convert to a probability:

```
    o = np.exp(log_o)
    p = o / (o+1)
```

So in the office pool scenario we could compute the predictive probability of having a boy. But how do we estimate the parameters?

제 7 절 Estimating parameters

Unlike linear regression, logistic regression does not have a closed form solution, so it is solved by guessing an initial solution and improving it iteratively.

The usual goal is to find the maximum-likelihood estimate (MLE), which is the set of parameters that maximizes the likelihood of the data. For example, suppose we have the following data:

```
>>> y = np.array([0, 1, 0, 1])
>>> x1 = np.array([0, 0, 0, 1])
>>> x2 = np.array([0, 1, 1, 1])
```

And we start with the initial guesses $\beta_0 = -1.5$, $\beta_1 = 2.8$, and $\beta_2 = 1.1$:

```
>>> beta = [-1.5, 2.8, 1.1]
```

Then for each row we can compute `log_o`:

```
>>> log_o = beta[0] + beta[1] * x1 + beta[2] * x2
[-1.5 -0.4 -0.4  2.4]
```

And convert from log odds to probabilities:

```
>>> o = np.exp(log_o)
[  0.223   0.670   0.670  11.02  ]

>>> p = o / (o+1)
[ 0.182  0.401  0.401  0.916 ]
```

Notice that when `log_o` is greater than 0, `o` is greater than 1 and `p` is greater than 0.5.

The likelihood of an outcome is `p` when `y==1` and `1-p` when `y==0`. For example, if we think the probability of a boy is 0.8 and the outcome is a boy, the likelihood is 0.8; if the outcome is a girl, the likelihood is 0.2. We can compute that like this:

```
>>> likes = y * p + (1-y) * (1-p)
[ 0.817  0.401  0.598  0.916 ]
```

The overall likelihood of the data is the product of `likes`:

```
>>> like = np.prod(likes)
0.18
```

For these values of `beta`, the likelihood of the data is 0.18. The goal of logistic regression is to find parameters that maximize this likelihood. To do that, most statistics packages use an iterative solver like Newton's method (see `https://en.wikipedia.org/wiki/Logistic_regression#Model_fitting`).

제 8 절 Implementation

StatsModels provides an implementation of logistic regression called `logit`, named for the function that converts from probability to log odds. To demonstrate its use, I'll look for variables that affect the sex ratio.

Again, I load the NSFG data and select pregnancies longer than 30 weeks:

```
    live, firsts, others = first.MakeFrames()
    df = live[live.prglngth>30]
```

`logit` requires the dependent variable to be binary (rather than boolean), so I create a new column named `boy`, using `astype(int)` to convert to binary integers:

```
    df['boy'] = (df.babysex==1).astype(int)
```

Factors that have been found to affect sex ratio include parents' age, birth order, race, and social status. We can use logistic regression to see if these effects appear in the NSFG data. I'll start with the mother's age:

```
    import statsmodels.formula.api as smf

    model = smf.logit('boy ~ agepreg', data=df)
    results = model.fit()
    SummarizeResults(results)
```

`logit` takes the same arguments as `ols`, a formula in Patsy syntax and a DataFrame. The result is a Logit object that represents the model. It contains attributes called `endog` and `exog` that contain the **endogenous variable**, another name for the dependent variable, and the **exogenous variables**, another name for the explanatory variables. Since they are NumPy arrays, it is sometimes convenient to convert them to DataFrames:

```
    endog = pandas.DataFrame(model.endog, columns=[model.endog_names])
    exog = pandas.DataFrame(model.exog, columns=model.exog_names)
```

The result of `model.fit` is a BinaryResults object, which is similar to the RegressionResults object we got from `ols`. Here is a summary of the results:

```
Intercept   0.00579   (0.953)
agepreg     0.00105   (0.783)
R^2 6.144e-06
```

The parameter of `agepreg` is positive, which suggests that older mothers are more likely to have boys, but the p-value is 0.783, which means that the apparent effect could easily be due to chance.

The coefficient of determination, R^2, does not apply to logistic regression, but there are several alternatives that are used as "pseudo R^2 values." These values can be useful for comparing models. For example, here's a model that includes several factors believed to be associated with sex ratio:

```
    formula = 'boy ~ agepreg + hpagelb + birthord + C(race)'
    model = smf.logit(formula, data=df)
    results = model.fit()
```

Along with mother's age, this model includes father's age at birth (`hpagelb`), birth order (`birthord`), and race as a categorical variable. Here are the results:

```
Intercept      -0.0301     (0.772)
C(race)[T.2]   -0.0224     (0.66)
C(race)[T.3]   -0.000457   (0.996)
agepreg        -0.00267    (0.629)
hpagelb         0.0047     (0.266)
birthord        0.00501    (0.821)
R^2 0.000144
```

None of the estimated parameters are statistically significant. The pseudo-R^2 value is a little higher, but that could be due to chance.

제 9 절 Accuracy

In the office pool scenario, we are most interested in the accuracy of the model: the number of successful predictions, compared with what we would expect by chance.

In the NSFG data, there are more boys than girls, so the baseline strategy is to guess “boy” every time. The accuracy of this strategy is just the fraction of boys:

```
    actual = endog['boy']
    baseline = actual.mean()
```

Since `actual` is encoded in binary integers, the mean is the fraction of boys, which is 0.507.

Here’s how we compute the accuracy of the model:

```
    predict = (results.predict() >= 0.5)
    true_pos = predict * actual
    true_neg = (1 - predict) * (1 - actual)
```

`results.predict` returns a NumPy array of probabilities, which we round off to 0 or 1. Multiplying by `actual` yields 1 if we predict a boy and get it right, 0 otherwise. So, `true_pos` indicates “true positives”.

Similarly, `true_neg` indicates the cases where we guess “girl” and get it right. Accuracy is the fraction of correct guesses:

```
    acc = (sum(true_pos) + sum(true_neg)) / len(actual)
```

The result is 0.512, slightly better than the baseline, 0.507. But, you should not take this result too seriously. We used the same data to build and test the model, so the model may not have predictive power on new data.

Nevertheless, let’s use the model to make a prediction for the office pool. Suppose your friend is 35 years old and white, her husband is 39, and they are expecting their third child:

```
    columns = ['agepreg', 'hpagelb', 'birthord', 'race']
    new = pandas.DataFrame([[35, 39, 3, 2]], columns=columns)
    y = results.predict(new)
```

To invoke `results.predict` for a new case, you have to construct a DataFrame with a column for each variable in the model. The result in this case is 0.52, so you should guess “boy.” But if the model improves your chances of winning, the difference is very small.

제 10 절 Exercises

My solution to these exercises is in `chap11soln.ipynb`.

Exercise 6.1 Suppose one of your co-workers is expecting a baby and you are participating in an office pool to predict the date of birth. Assuming that bets are placed during the 30th week of pregnancy, what variables could you use to make the best prediction? You should limit yourself to variables that are known before the birth, and likely to be available to the people in the pool.

Exercise 6.2 The Trivers-Willard hypothesis suggests that for many mammals the sex ratio depends on "maternal condition"; that is, factors like the mother's age, size, health, and social status. See `https://en.wikipedia.org/wiki/Trivers-Willard_hypothesis`

Some studies have shown this effect among humans, but results are mixed. In this chapter we tested some variables related to these factors, but didn't find any with a statistically significant effect on sex ratio.

As an exercise, use a data mining approach to test the other variables in the pregnancy and respondent files. Can you find any factors with a substantial effect?

Exercise 6.3 If the quantity you want to predict is a count, you can use Poisson regression, which is implemented in StatsModels with a function called `poisson`. It works the same way as `ols` and `logit`. As an exercise, let's use it to predict how many children a woman has born; in the NSFG dataset, this variable is called `numbabes`.

Suppose you meet a woman who is 35 years old, black, and a college graduate whose annual household income exceeds $75,000. How many children would you predict she has born?

Exercise 6.4 If the quantity you want to predict is categorical, you can use multinomial logistic regression, which is implemented in StatsModels with a function called `mnlogit`. As an exercise, let's use it to guess whether a woman is married, cohabitating, widowed, divorced, separated, or never married; in the NSFG dataset, marital status is encoded in a variable called `rmarital`.

Suppose you meet a woman who is 25 years old, white, and a high school graduate whose annual household income is about $45,000. What is the probability that she is married, cohabitating, etc?

제 11 절 Glossary

- regression: One of several related processes for estimating parameters that fit a model to data.
- dependent variables: The variables in a regression model we would like to predict. Also known as endogenous variables.
- explanatory variables: The variables used to predict or explain the dependent variables. Also known as independent, or exogenous, variables.
- simple regression: A regression with only one dependent and one explanatory variable.
- multiple regression: A regression with multiple explanatory variables, but only one dependent variable.
- linear regression: A regression based on a linear model.
- ordinary least squares: A linear regression that estimates parameters by minimizing the squared error of the residuals.
- spurious relationship: A relationship between two variables that is caused by a statistical artifact or a factor, not included in the model, that is related to both variables.
- control variable: A variable included in a regression to eliminate or "control for" a spurious relationship.
- proxy variable: A variable that contributes information to a regression model indirectly because of a relationship with another factor, so it acts as a proxy for that factor.
- categorical variable: A variable that can have one of a discrete set of unordered values.
- join: An operation that combines data from two DataFrames using a key to match up rows in the two frames.
- data mining: An approach to finding relationships between variables by testing a large number of models.
- logistic regression: A form of regression used when the dependent variable is boolean.

- Poisson regression: A form of regression used when the dependent variable is a non-negative integer, usually a count.
- odds: An alternative way of representing a probability, p, as the ratio of the probability and its complement, $p/(1-p)$.