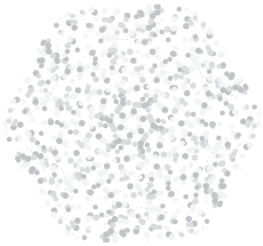
1



STATISTICS IN THE MODERN DAY

> Retake the falling snow: each drifting flake
> Shapeless and slow, unsteady and opaque,
> A dull dark white against the day's pale white
> And abstract larches in the neutral light.
>
> —Nabokov (1962, lines 13–16)

Statistical analysis has two goals, which directly conflict. The first is to find patterns in static: given the infinite number of variables that one could observe, how can one discover the relations and patterns that make human sense? The second goal is a fight against *apophenia*, the human tendency to invent patterns in random static. Given that someone has found a pattern regarding a handful of variables, how can one verify that it is not just the product of a lucky draw or an overactive imagination?

Or, consider the complementary dichotomy of objective versus subjective. The objective side is often called *probability*; e.g., given the assumptions of the Central Limit Theorem, its conclusion is true with mathematical certainty. The subjective side is often called *statistics*; e.g., our claim that observed quantity A is a linear function of observed quantity B may be very useful, but Nature has no interest in it.

This book is about writing down subjective models based on our human understanding of how the world works, but which are heavily advised by objective information, including both mathematical theorems and observed data.[1]

[1] Of course, human-gathered data is never perfectly objective, but we all try our best to make it so.

The typical scheme begins by proposing a model of the world, then estimating the parameters of the model using the observed data, and then evaluating the fit of the model. This scheme includes both a descriptive step (describing a pattern) and an inferential step (testing whether there are indications that the pattern is valid). It begins with a subjective model, but is heavily advised by objective data.

Figure 1.1 shows a model in flowchart form. First, the descriptive step: data and parameters are fed into a function—which may be as simple as *a is correlated to b*, or may be a complex set of interrelations—and the function spits out some output. Then comes the testing step: evaluating the output based on some criterion, typically regarding how well it matches some portion of the data. Our goal is to find those parameters that produce output that best meets our evaluation criterion.

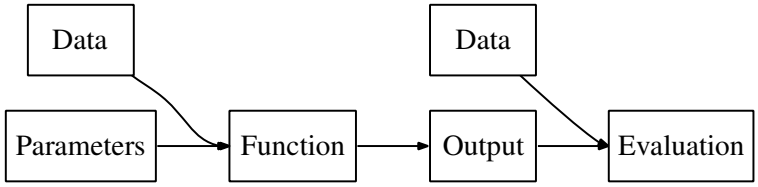


Figure 1.1 A flowchart for distribution fitting, linear regression, maximum likelihood methods, multilevel modeling, simulation (including agent-based modeling), data mining, non-parametric modeling, and various other methods. [Online source for the diagram: `models.dot`.]

The Ordinary Least Squares (OLS) model is a popular and familiar example, pictured in Figure 1.2. [If it is not familiar to you, we will cover it in Chapter 8.] Let $\mathbf{X}$ indicate the independent data, $\boldsymbol{\beta}$ the parameters, and $\mathbf{y}$ the dependent data. Then the function box consists of the simple equation $\mathbf{y}_{\text{out}} = \mathbf{X}\boldsymbol{\beta}$, and the evaluation step seeks to minimize squared error, $(\mathbf{y} - \mathbf{y}_{\text{out}})^2$.

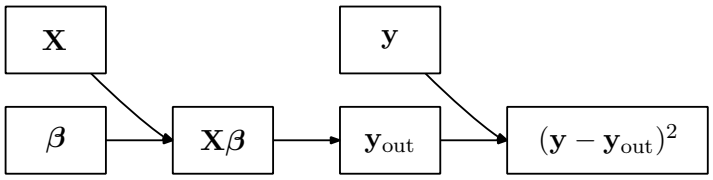


Figure 1.2 The OLS model: a special case of Figure 1.1.

For a simulation, the function box may be a complex flowchart in which variables are combined non-linearly with parameters, then feed back upon each other in unpredictable ways. The final step would evaluate how well the simulation output corresponds to the real-world phenomenon to be explained.

The key computational problem of statistical modeling is to find the parameters at

the beginning of the flowchart that will output the best evaluation at the end. That is, for a given function and evaluation in Figure 1.1, we seek a routine to take in data and produce the optimal parameters, as in Figure 1.3. In the OLS model above, there is a simple, one-equation solution to the problem: $\boldsymbol{\beta}_{\text{best}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$. But for more complex models, such as simulations or many multilevel models, we must strategically try different sets of parameters to hunt for the best ones.

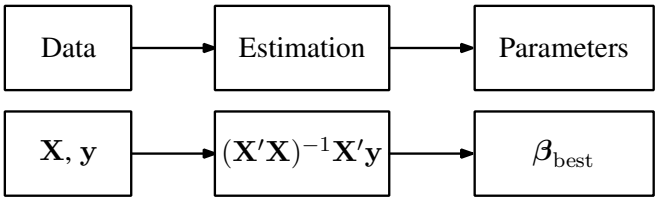


Figure 1.3 Top: the parameters which are the input for the model in Figure 1.1 are the output for the estimation routine.
Bottom: the estimation of the OLS model is a simple equation.

And that's the whole book: develop models whose parameters and tests may discover and verify interesting patterns in the data. But the setup is incredibly versatile, and with different function specifications, the setup takes many forms. Among a few minor asides, this book will cover the following topics, all of which are variants of Figure 1.1:

- Probability: how well-known distributions can be used to model data
- Projections: summarizing many-dimensional data in two or three dimensions
- Estimating linear models such as OLS
- Classical hypothesis testing: using the Central Limit Theorem (CLT) to ferret out apophenia
- Designing multilevel models, where one model's output is the input to a parent model
- Maximum likelihood estimation
- Hypothesis testing using likelihood ratio tests
- Monte Carlo methods for describing parameters
- "Nonparametric" modeling (which comfortably fits into the parametric form here), such as smoothing data distributions
- Bootstrapping to describe parameters and test hypotheses

THE SNOWFLAKE PROBLEM, OR A BRIEF HISTORY OF STATISTICAL COMPUTING

The simplest models in the above list have only one or two parameters, like a Binomial(n, p) distribution which is built from n identical draws, each of which is a success with probability p [see Chapter 7]. But draws in the real world are rarely identical—no two snowflakes are exactly alike. It would be nice if an outcome variable, like annual income, were determined entirely by one variable (like education), but we know that a few dozen more enter into the picture (like age, race, marital status, geographical location, et cetera).

The problem is to design a model that accommodates that sort of complexity, in a manner that allows us to actually compute results. Before computers were common, the best we could do was analysis of variance methods (ANOVA), which ascribed variation to a few potential causes [see Sections 7.1.3 and 9.4].

The first computational milestone, circa the early 1970s, arrived when civilian computers had the power to easily invert matrices, a process that is necessary for most linear models. The linear models such as ordinary least squares then became dominant [see Chapter 8].

The second milestone, circa the mid 1990s, arrived when desktop computing power was sufficient to easily gather enough local information to pin down the global optimum of a complex function—perhaps thousands or millions of evaluations of the function. The functions that these methods can handle are much more general than the linear models: you can now write and optimize models with millions of interacting agents or functions consisting of the sum of a thousand sub-distributions [see Chapter 10].

The ironic result of such computational power is that it allows us to return to the simple models like the Binomial distribution. But instead of specifying a fixed n and p for the entire population, every observation could take on a value of n that is a function of the individual's age, race, et cetera, and a value of p that is a different function of age, race, et cetera [see Section 8.4].

The models in Part II are listed more-or-less in order of complexity. The infinitely quotable Albert Einstein advised, "make everything as simple as possible, but not simpler." The Central Limit Theorem tells us that errors often are Normally distributed, and it is often the case that the dependent variable is basically a linear or log-linear function of several variables. If such descriptions do no violence to the reality from which the data were culled, then OLS is the method to use, and using more general techniques will not be any more persuasive. But if these assumptions do not apply, we no longer need to assume linearity to overcome the snowflake problem.

THE PIPELINE A statistical analysis is a guided series of transformations of the data from its raw form as originally written down to a simple summary regarding a question of interest.

The flow above, in the statistics textbook tradition, picked up halfway through the analysis: it assumes a data set that is in the correct form. But the full pipeline goes from the original messy data set to a final estimation of a statistical model. It is built from functions that each incrementally transform the data in some manner, like removing missing data, selecting a subset of the data, or summarizing it into a single statistic like a mean or variance.

Thus, you can think of this book as a catalog of pipe sections and filters, plus a discussion of how to fit elements together to form a stream from raw data to final publishable output. As well as the pipe sections listed above, such as the ordinary least squares or maximum likelihood procedures, the book also covers several techniques for directly transforming data, computing statistics, and welding all these sections into a full program:

- Structuring programs using modular functions and the *stack* of *frames*
- Programming tools like the debugger and profiler
- Methods for reliability testing functions and making them more robust
- Databases, and how to get them to produce data in the format you need
- Talking to external programs, like graphics packages that will generate visualizations of your data
- Finding and using pre-existing functions to quickly estimate the parameters of a model from data.
- Optimization routines: how they work and how to use them
- Monte Carlo methods: getting a picture of a model via millions of random draws

To make things still more concrete, almost all of the sample code in this book is available from the book's Web site, linked from `http://press.princeton.edu/titles/8706.html`. This means that you can learn by running and modifying the examples, or you can cut, paste, and modify the sample code to get your own analyses running more quickly. The programs are listed and given a complete discussion on the pages of this book, so you can read it on the bus or at the beach, but you are very much encouraged to read through this book while sitting at your computer, where you can run the sample code, see what happens given different settings, and otherwise explore.

Figure 1.4 gives a typical pipeline from raw data to final paper. It works at a number of different *layers of abstraction*: some segments involve manipulating individual numbers, some segments take low-level numerical manipulation as given and op-

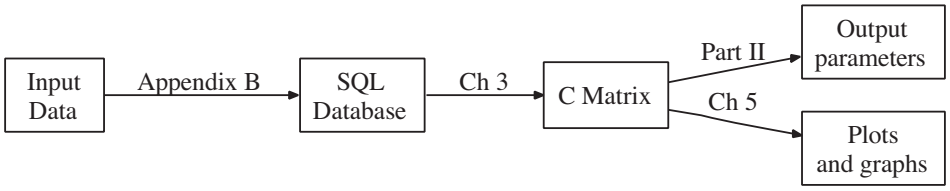


Figure 1.4 Filtering from input data to outputs. [Online source: `datafiltering.dot`]

erate on database tables or matrices, and some segments take matrix operations as given and run higher-level hypothesis tests.

The lowest level Chapter 2 presents a tutorial on the C programming language itself. The work here is at the lowest level of abstraction, covering nothing more difficult than adding columns of numbers. The chapter also discusses how C facilitates the development and use of *libraries*: sets of functions written by past programmers that provide the tools to do work at higher and higher levels of abstraction (and thus ignore details at lower levels).[2]

For a number of reasons to be discussed below, the book relies on the C programming language for most of the pipe-fitting, but if there is a certain section that you find useful (the appendices and the chapter on databases comes to mind) then there is nothing keeping you from welding that pipe section to others using another programming language or system.

Dealing with large data sets Computers today are able to crunch numbers a hundred times faster they did a decade ago—but the data sets they have to crunch are a thousand times larger. Geneticists routinely pull 550,000 genetic markers each from a hundred or a thousand patients. The US Census Bureau's 1% sample covers almost 3 million people. Thus, the next layer of abstraction provides specialized tools for dealing with data sets: databases and a query language for organizing data. Chapter 3 presents a new syntax for talking to a database, Structured Query Language (SQL). You will find that many types of data manipulation and filtering that are difficult in traditional languages or stats packages are trivial—even pleasant—via SQL.

[2]Why does the book omit a linear algebra tutorial but include an extensive C tutorial? Primarily because the use of linear algebra has not changed much this century, while the use of C has evolved as more libraries have become available. If you were writing C code in the early 1980s, you were using only the standard library and thus writing at a very low level. In the present day, the process of writing code is more about joining together libraries than writing from scratch. I felt that existing C tutorials and books focused too heavily on the process of writing from scratch, perpetuating the myth that C is appropriate only for low-level bit shifting. The discussion of C here introduces tools like package managers, the debugger, and the `make` utility as early as possible, so you can start calling existing libraries as quickly and easily as possible.

As Huber (2000, p 619) explains: "Large real-life problems always require a combination of database management and data analysis. . . . Neither database management systems nor traditional statistical packages are up to the task." The solution is to build a pipeline, as per Figure 1.4, that includes both database management and statistical analysis sections. Much of graceful data handling is in knowing where along the pipeline to place a filtering operation. The database is the appropriate place to filter out bad data, join together data from multiple sources, and aggregate data into group means and sums. C matrices are appropriate for filtering operations like those from earlier that took in data, applied a function like $(\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, and then measured $(\mathbf{y}_{\text{out}} - \mathbf{y})^2$.

Because your data probably did not come pre-loaded into a database, Appendix B discusses text manipulation techniques, so when the database expects your data set to use commas but your data is separated by erratic tabs, you will be able to quickly surmount the problem and move on to analysis.

Computation The GNU Scientific Library works at the numerical computation layer of abstraction. It includes tools for all of the procedures commonly used in statistics, such as linear algebra operations, looking up the value of F, t, χ^2 distributions, and finding maxima of likelihood functions. Chapter 4 presents some basics for data-oriented use of the GSL.

The Apophenia library, primarily covered in Chapter 4, builds upon these other layers of abstraction to provide functions at the level of data analysis, model fitting, and hypothesis testing.

Pretty pictures Good pictures can be essential to good research. They often reveal patterns in data that look like mere static when that data is presented as a table of numbers, and are an effective means of communicating with peers and persuading grantmakers. Consistent with the rest of this book, Chapter 5 will cover the use of Gnuplot and Graphviz, two packages that are freely available for the computer you are using right now. Both are entirely automatable, so once you have a graph or plot you like, you can have your C programs autogenerate it or manipulate it in amusing ways, or can send your program to your colleague in Madras and he will have no problem reproducing and modifying your plots.[3] Once you have the basics down, animation and real-time graphics for simulations are easy.

[3]Following a suggestion by Thomson (2001), I have chosen the gender of representative agents in this book by flipping a coin.

WHY C? You may be surprised to see a book about modern statistical computing based on a language composed in 1972. Why use C instead of a specialized language or package like SAS, Stata, SPSS, S-Plus, SAGE, SIENA, SUDAAN, SYSTAT, SST, SHAZAM, J, K, GAUSS, GAMS, GLIM, GENSTAT, GRETL, EViews, Egret, EQS, PcGive, MatLab, Minitab, Mupad, Maple, Mplus, Maxima, MLn, Mathematica, WinBUGS, TSP, HLM, R, RATS, LISREL, Lisp-Stat, LIMDEP, BMDP, Octave, Orange, OxMetrics, Weka, or Yorick? This may be the only book to advocate statistical computing with a general computing language, so I will take some time to give you a better idea of why modern numerical analysis is best done in an old language.

One of the side effects of a programming language being stable for so long is that a mythology builds around it. Sometimes the mythology is outdated or false: I have seen professional computer programmers and writers claim that simple structures like linked lists always need to be written from scratch in C (see Section 6.2 for proof otherwise), that it takes ten to a hundred times as long to write a program in C than in a more recently-written language like R, or that because people have used C to write device drivers or other low-level work, it can not be used for high-level work.[4] This section is partly intended to dispel such myths.

Is C a hard language? C *was* a hard language. With nothing but a basic 80s-era compiler, you could easily make many hard-to-catch mistakes. But programmers have had a few decades to identify those pitfalls and build tools to catch them. Modern compilers warn you of these issues, and debuggers let you interact with your program as it runs to catch more quirks. C's reputation as a hard language means the tools around it have evolved to make it an easy language.

Computational speed—really Using a stats package sure beats inverting matrices by hand, but as computation goes, many stats packages are still relatively slow, and that slowness can make otherwise useful statistical methods infeasible.

R and Apophenia use the same C code for doing the Fisher exact test, so it makes a good basis for a timing test.[5] Listings 1.5 and 1.6 show programs in C and R (respectively) that will run a Fisher exact test five million times on the same data set. You can see that the C program is a bit more verbose: the steps taken in lines 3–8 of the C code and lines 1–6 of the R code are identical, but those lines are

[4] Out of courtesy, citations are omitted. This section makes frequent comparisons to R partly because it is a salient and common stats package, and partly because I know it well, having used it on a daily basis for several years.

[5] That is, if you download the source code for R's `fisher.test` function, you will find a set of procedures written in C. Save for a few minor modifications, the code underlying the `apop_test_fisher_exact` function is line-for-line identical.

```
1  #include <apop.h>
2  int main(){
3      int i, test_ct = 5e6;
4      double data[] = { 30, 86,
5                        24, 38 };
6      apop_data *testdata = apop_line_to_data(data,0,2,2);
7      for (i = 0; i< test_ct; i++)
8          apop_test_fisher_exact(testdata);
9  }
```

Listing 1.5 C code to time a Fisher exact test. It runs the same test five million times. Online source: `timefisher.c`.

```
1  test_ct <- 5e6
2  data <- c( 30, 86,
3             24, 38 )
4  testdata<- matrix(data, nrow=2)
5  for (i in 1:test_ct){
6      fisher.test(testdata)
7  }
```

Listing 1.6 R code to do the same test as Listing 1.5. Online source: `Rtimefisher`.

longer in C, and the C program has some preliminary code that the R script does not have.

On my laptop, Listing 1.5 runs in under three minutes, while Listing 1.6 does the same work in 89 minutes—about thirty times as long. So the investment of a little more verbosity and a few extra stars and semicolons returns a thirty-fold speed gain.[6] Nor is this an isolated test case: I can't count how many times people have told me stories about an analysis or simulation that took days or weeks in a stats package but ran in minutes after they rewrote it in C.

Even for moderately-sized data sets, real computing speed opens up new possibilities, because we can drop the (typically false) assumptions needed for closed-form solutions in favor of maximum likelihood or Monte Carlo methods. The Monte Carlo examples in Section 11.2 were produced using over a billion draws from t distributions; if your stats package can't produce a few hundred thousand draws per second (some can't), such work will be unfeasibly slow.[7]

[6] These timings are actually based on a modified version of `fisher.test` that omits some additional R-side calculations. If you had to put a Fisher test in a `for` loop without first editing R's code, the R-to-C speed ratio would be between fifty and a hundred.

[7] If you can produce random draws from t distributions as a batch (`draws <- rt(5e6, df)`), then R takes a mere 3.5 times as long as comparable C code. But if you need to produce them individually (`for (i in 1:5e6) {draw <- rt(1, df)}`), then R takes about fifteen times as long as comparable C code. On my laptop, R in

Simplicity C is a super-simple language. Its syntax has no special tricks for polymorphic operators, abstract classes, virtual inheritance, lexical scoping, lambda expressions, or other such arcana, meaning that you have less to learn. Those features are certainly helpful in their place, but without them C has already proven to be sufficient for writing some impressive programs, like the Mac and Linux operating systems and most of the stats packages listed above.

Simplicity affords stability—C is among the oldest programming languages in common use today[8]—and stability brings its own benefits. First, you are reasonably assured that you will be able to verify and modify your work five or even ten years from now. Since C was written in 1972, countless stats packages have come and gone, while others are still around but have made so many changes in syntax that they are effectively new languages. Either way, those who try to follow the trends have on their hard drives dozens of scripts that they can't run anymore. Meanwhile, correctly written C programs from the 1970s will compile and run on new PCs.

Second, people have had a few decades to write good libraries, and libraries that build upon those libraries. It is not the syntax of a language that allows you to easily handle complex structures and tasks, but the vocabulary, which in the case of C is continually being expanded by new function libraries. With a statistics library on hand, the C code in Listing 1.5 and the R code in Listing 1.6 work at the same high level of abstraction.

Alternatively, if you need more precision, you can use C's low-level bit-twiddling to shunt individual elements of data. There is nothing more embarrassing than a presenter who answers a question about an anomaly in the data or analysis with 'Stata didn't have a function to correct that.' [Yes, I have heard this in a real live presentation by a real live researcher.] But since C's higher-level and lower-level libraries are equally accessible, you can work at the level of laziness or precision called for in any given situation.

Interacting with C scripts Many of the stats packages listed above provide a pleasing interface that let you run regressions with just a few mouse-clicks. Such systems are certainly useful for certain settings, such as asking a few quick questions of a new data set. But an un-replicable analysis based on clicking an arbitrary sequence of on-screen buttons is as useful as no analysis at all. In the context of building a repeatable script that takes the data as far as possible along the pipeline from raw format to final published output, developing

batch mode produced draws at a rate $\approx 424,000/\text{sec}$, while C produced draws at a rate $\approx 1,470,000/\text{sec}$.

[8]However, it is not the oldest, an honor that goes to FORTRAN. This is noteworthy because some claim that C is in common use today merely because of inertia, path dependency, et cetera. But C displaced a number of other languages such as ALGOL and PL/I which had more inertia behind them, by making clear improvements over the incumbents.

`script.do` for an interpreter and developing `program.c` for a compiler become about equivalent—especially since compilation on a modern computer takes on the order of 0.0 seconds.

With a debugger, the distance is even smaller, because you can jump around your C code, change intermediate values, and otherwise interact with your program the way you would with a stats package. Graphical interfaces for stats packages and for C debuggers tend to have a similar design.

But C is ugly! C is by no means the best language for all possible purposes. Different systems have specialized syntaxes for communicating with other programs, handling text, building Web pages, or producing certain graphics. But for data analysis, C is very effective. It has its syntactic flaws: you will forget to append semicolons to every line, and will be frustrated that `3/2==1` while `3/2.==1.5`. But then, Perl also requires semicolons after every line, and `3/2` is one in Perl, Python, and Ruby too. Type declarations are one more detail to remember, but the alternatives have their own warts: Perl basically requires that you declare the type of your variable (`@`, `$`, or `#`) with every use, and R will guess the type you meant to use, but will often guess wrong, such as thinking that a one-element list like `{14}` is really just an integer. C's `printf` statements look terribly confusing at first, but the authors of Ruby and Python, striving for the most programmer-friendly syntax possible, chose to use C's `printf` syntax over many alternatives that are easier on the eyes but harder to use.

In short, C does not do very well when measured by initial ease-of-use. But there is a logic to its mess of stars and braces, and over the course of decades, C has proven to be very well suited for designing pipelines for data analysis, linking together libraries from disparate sources, and describing detailed or computation-intensive models.

TYPOGRAPHY Here are some notes on the typographic conventions used by this book.

※ *Seeing the forest for the trees* On the one hand, a good textbook should be a narrative that plots a definite course through a field. On the other hand, most fields have countless interesting and useful digressions and side-paths. Sections marked with a ※ cover details that may be skipped on a first reading. They are not necessarily advanced in the sense of being somehow more difficult than unmarked text, but they may be distractions to the main narrative.

$\mathbb{Q}_{1.1}$ Questions and exercises are marked like this paragraph. The exercises are not thought experiments. It happens to all of us that we think we understand something until we sit down to actually do it, when a host of hairy details turn up. Especially at the outset, the exercises are relatively simple tasks that let you face the hairy details before your own real-world complications enter the situation. Exercises in the later chapters are more involved and require writing or modifying longer segments of code.

Notation

$\mathbf{X}$: boldface, capital letters are matrices. With few exceptions, data matrices in this book are organized so that the rows are each a single observation, and each column is a variable.

$\mathbf{x}$: lowercase boldface indicates a vector. Vectors are generally a column of numbers, and their transpose, $\mathbf{x}'$, is a row. $\mathbf{y}$ is typically a vector of dependent variables (the exception being when we just need two generic data vectors, in which case one will be $\mathbf{x}$ and one $\mathbf{y}$).

x: A lowercase variable, not bold, is a scalar, i.e., a single real number.

$\mathbf{X}'$ is the transpose of the matrix $\mathbf{X}$. Some authors notate this as $\mathbf{X^T}$.

$\underline{\mathbf{X}}$ is the data matrix $\mathbf{X}$ with the mean of each column subtracted, meaning that each column of $\underline{\mathbf{X}}$ has mean zero. If $\mathbf{X}$ has a column of ones (as per most regression techniques), then the constant column is left unmodified in $\underline{\mathbf{X}}$.

n: the number of observations in the data set under discussion, which is typically the number of rows in $\mathbf{X}$. When there is ambiguity, n will be subscripted.

$\mathbf{I}$: The *identity matrix*. A square matrix with ones along its diagonal and zeros everywhere else.

β: Greek letters indicate parameters to be estimated; if boldface, they are a vector of parameters. The most common letter is β, but others may slip in, such as...

σ, μ: the standard deviation and the mean. The variance is σ^2.

$\hat{\sigma}, \hat{\beta}$: a carat over a parameter indicates an empirical estimate of the parameter derived from data. Typically read as, e.g., *sigma hat*, *beta hat*.

$\boldsymbol{\epsilon} \sim \mathcal{N}(0, 1)$: Read this as *epsilon is distributed as a Normal distribution with parameters 0 and 1.*

$P(\cdot)$: A probability density function.

$LL(\cdot)$: The log likelihood function, $\ln(P(\cdot))$.

$S(\cdot)$: The Score, which is the vector of derivatives of $LL(\cdot)$.

$\mathbb{I}(\cdot)$: The information matrix, which is the matrix of second derivatives of $LL(\cdot)$.

$E(\cdot)$: The expected value, aka the mean, of the input.

$P(x|\beta)$: The probability of x given that β is true.

$P(x, \beta)|_x$: The probability density function, holding x fixed. Mathematically, this is simply $P(x, \beta)$, but in the given situation it should be thought of as a function only of β.[9]

$E_x(f(x, \beta))$: Read as *the expectation over* x of the given function, which will take a form like $\int_{\forall x} f(x, \beta)P(x)dx$. Because the integral is over all x, $E_x(f(x, \beta))$ is not itself a function of x.

`teletype typeface` indicates text that can be typed directly into a text file and understood as a valid shell script, C commands, SQL queries, et cetera.

`cat` *`sample_file`*: Slanted teletype text indicates a placeholder for text you will insert—a variable name rather than text to be read literally. You could read the code here as, 'let *`sample_file`* be the name of a file on your hard drive. Then type `cat` *`sample_file`* at the command prompt'.

$a \equiv b$: Read as 'a is equivalent to b' or 'a is defined as b'.

$a \propto b$: Read as 'a is proportional to b'.

2.3e6: Engineers often write scientific notation using so-called *exponential* or *E notation*, such as $2.3 \times 10^6 \equiv$ `2.3e6`. Many computing languages (including C, SQL, and Gnuplot) recognize E-notated numbers.

[9] Others use a different notation. For example, Efron & Hinkley (1978, p 458): "The log likelihood function $l_\theta(x)\ldots$ is the log of the density function, thought of as a function of θ." See page 329 for more on the philosophical considerations underlying the choice of notation.

Every section ends with a summary of the main points, set like this paragraph. There is much to be said for the strategy of flipping ahead to the summary at the end of the section before reading the section itself.
The summary for the introduction:

$\sum$

- This book will discuss methods of estimating and testing the parameters of a model with data.
- It will also cover the means of writing for a computer, including techniques to manage data, plot data sets, manipulate matrices, estimate statistical models, and test claims about their parameters.

Credits Thanks to the following people, who added higher quality and richness to the book:

- Anjeanette Agro for graphic design suggestions.
- Amber Baum for extensive testing and critique.
- The Brookings Institution's Center on Social and Economic Dynamics, including Rob Axtell, Josh Epstein, Carol Graham, Emily Groves, Ross Hammond, Jon Parker, Matthew Raifman, and Peyton Young.
- Dorothy Gambrel, author of *Cat and Girl*, for the Lonely Planet data.
- Rob Goodspeed and the National Center for Smart Growth Research and Education at the University of Maryland, for the Washington Metro data.
- Derrick Higgins for comments, critique, and the Perl commands on page 414.
- Lucy Day Hobor and Vickie Kearn for editorial assistance and making working with Princeton University Press a pleasant experience.
- Guy Klemens, for a wide range of support on all fronts.
- Anne Laumann for the tattoo data set (Laumann & Derick, 2006).
- Abigail Rudman for her deft librarianship.