5

Simulation

You will recall from your previous statistics courses that quantifying uncertainty in statistical inference requires us to get at the **sampling distributions** of things like estimators. When the very strong simplifying assumptions of basic statistics courses do not apply[1], there is little hope of being able to write down sampling distributions in closed form. There is equally little help when the estimates are themselves complex objects, like kernel regression curves or even histograms, rather than short, fixed-length parameter vectors. We get around this by using simulation to approximate the sampling distributions we can't calculate.

5.1 What Is a Simulation?

A mathematical model is a mathematical story about how the data could have been made, or **generated**. Simulating the model means following that story, implementing it, step by step, in order to produce something which should look like the data — what's sometimes called **synthetic data**, or **surrogate data**, or a **realization** of the model. In a stochastic model, some of the steps we need to follow involve a random component, and so multiple simulations starting from exactly the same inputs or initial conditions will not give exactly the same outputs or realizations. Rather, the model specifies a distribution over the realizations, and doing many simulations gives us a good approximation to this distribution.

For a trivial example, consider a model with three random variables, $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$, $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$, with $X_1 \perp\!\!\!\perp X_2$, and $X_3 = X_1 + X_2$. Simulating from this model means drawing a random value from the first normal distribution for X_1, drawing a second random value for X_2, and adding them together to get X_3. The marginal distribution of X_3, and the joint distribution of (X_1, X_2, X_3), are implicit in this specification of the model, and we can find them by running the simulation.

In this particular case, we could also find the distribution of X_3, and the joint distribution, by probability calculations of the kind you learned how to do in your basic probability courses. For instance, X_3 is $\mathcal{N}(\mu_1 + \mu_2, \sigma_1^2 + \sigma_2^2)$. These

[1] As discussed *ad nauseam* in Chapter 2, in your linear models class, you learned about the sampling distribution of regression coefficients when the linear model is true, and the noise is Gaussian, independent of the predictor variables, and has constant variance. As an exercise, try to get parallel results when the noise has a t distribution with 10 degrees of freedom.

analytical probability calculations can usually be thought of as just short-cuts for exhaustive simulations.

5.2 How Do We Simulate Stochastic Models?

5.2.1 Chaining Together Random Variables

Stochastic models are usually specified by sets of conditional distributions for one random variable, given some other variable or variables. For instance, a simple linear regression model might have the specification

$$X \sim \mathcal{U}(x_{\min}, x_{\max}) \tag{5.1}$$

$$Y|X \sim \mathcal{N}(\beta_0 + \beta_1 X, \sigma^2) \tag{5.2}$$

If we knew how to generate a random variable from the distributions given on the right-hand sides, we could simulate the whole model by chaining together draws from those conditional distributions. This is in fact the general strategy for simulating any sort of stochastic model, by chaining together random variables.[2]

You might ask why we don't start by generating a random Y, and then generate X by drawing from the $X|Y$ distribution. The basic answer is that you could, but it would generally be messier. (Just try to work out the conditional distribution $X|Y$.) More broadly, in Chapter 20, we'll see how to arrange the variables in complicated probability models in a natural order, so that we start with independent, "exogenous" variables, then first-generation variables which only need to be conditioned on the exogenous variables, then second-generation variables which are conditioned on first-generation ones, and so forth. This is also the natural order for simulation.

The upshot is that we can reduce the problem of simulating to that of generating random variables.

5.2.2 Random Variable Generation

5.2.2.1 Built-in Random Number Generators

R provides random number generators for most of the most common distributions. By convention, the names of these functions all begin with the letter "r", followed by the abbreviation of the functions, and the first argument is always the number of draws to make, followed by the parameters of the distribution. Some examples:

```
rnorm(n, mean = 0, sd = 1)
runif(n, min = 0, max = 1)
rexp(n, rate = 1)
rpois(n, lambda)
rbinom(n, size, prob)
```

[2] In this case, we could in principle first generate Y, and then draw from $Y|X$, but have fun finding those distributions. Especially have fun if, say, X has a t distribution with 10 degrees of freedom. (I keep coming back to that idea, because it's really a very small change from being Gaussian.)

A further convention is that these parameters can be **vectorized**. Rather than giving a single mean and standard deviation (say) for multiple draws from the Gaussian distribution, each draw can have its own:

```
rnorm(10, mean = 1:10, sd = 1/sqrt(1:10))
```

That instance is rather trivial, but the exact same principle would be at work here:

```
rnorm(nrow(x), mean = predict(regression.model, newdata = x), sd = predict(volatility.model,
    newdata = x))
```

where `regression.model` and `volatility.model` are previously-defined parts of the model which tell us about conditional expectations and conditional variances.

Of course, none of this explains how R actually draws from any of these distributions; it's all at the level of a black box, which is to say black magic. Because ignorance is evil, and, even worse, *unhelpful* when we need to go beyond the standard distributions, it's worth opening the black box just a bit. We'll look at using transformations between distributions, and, in particular, transforming uniform distributions into others (§5.2.2.3). Appendix M explains some more advanced methods, and looks at the issue of how to get uniformly-distributed random numbers in the first place.

5.2.2.2 Transformations

If we can generate a random variable Z with some distribution, and $V = g(Z)$, then we can generate V. So one thing which gets a lot of attention is writing random variables as transformations of one another — ideally as transformations of easy-to-generate variables.

Example: from standard to customized Gaussians

Suppose we can generate random numbers from the standard Gaussian distribution $Z \sim \mathcal{N}(0,1)$. Then we can generate from $\mathcal{N}(\mu, \sigma^2)$ as $\sigma Z + \mu$. We can generate χ^2 random variables with 1 degree of freedom as Z^2. We can generate χ^2 random variables with d degrees of freedom by summing d independent copies of Z^2.

In particular, if we can generate random numbers uniformly distributed between 0 and 1, we can use this to generate anything which is a transformation of a uniform distribution. How far does that extend?

5.2.2.3 Quantile Method

Suppose that we know the **quantile function** Q_Z for the random variable Z we want, so that $Q_Z(0.5)$ is the median of X, $Q_Z(0.9)$ is the 90th percentile, and in general $Q_Z(p)$ is bigger than or equal to Z with probability p. Q_Z comes as a pair with the cumulative distribution function F_Z, since

$$Q_Z(F_Z(a)) = a, \; F_Z(Q_Z(p)) = p \tag{5.3}$$

In the **quantile method** (or **inverse distribution transform method**), we generate a uniform random number U and feed it as the argument to Q_Z. Now $Q_Z(U)$ has the distribution function F_Z:

$$\Pr(Q_Z(U) \leq a) = \Pr(F_Z(Q_Z(U)) \leq F_Z(a)) \tag{5.4}$$
$$= \Pr(U \leq F_Z(a)) \tag{5.5}$$
$$= F_Z(a) \tag{5.6}$$

where the last line uses the fact that U is uniform on $[0,1]$, and the first line uses the fact that F_Z is a non-decreasing function, so $b \leq a$ is true if and only if $F_Z(b) \leq F_Z(a)$.

Example. The CDF of the exponential distribution with rate λ is $1 - e^{-\lambda z}$. The quantile function $Q(p)$ is thus $-\frac{\log(1-p)}{\lambda}$. (Notice that this is positive, because $1-p < 1$ and so $\log(1-p) < 0$, and that it has units of $1/\lambda$, which are the units of z, as it should.) Therefore, if U Unif$(0,1)$, then $-\frac{\log(1-U)}{\lambda} \sim \text{Exp}(\lambda)$. This is the method used by `rexp()`.

Example: Power laws

The **Pareto distribution** or **power law** is a two-parameter family, $f(z;\alpha,z_0) = \frac{\alpha-1}{z_0}\left(\frac{z}{z_0}\right)^{-\alpha}$ if $z \geq z_0$, with density 0 otherwise. Integration shows that the cumulative distribution function is $F(z;\alpha,z_0) = 1 - \left(\frac{z}{z_0}\right)^{-\alpha+1}$. The quantile function therefore is $Q(p;\alpha,z_0) = z_0(1-p)^{-\frac{1}{\alpha-1}}$. (Notice that this has the same units as z, as it should.)

Example: Gaussians

The standard Gaussian $\mathcal{N}(0,1)$ does not have a closed form for its quantile function, but there are fast and accurate ways of calculating it numerically (they're what stand behind `qnorm`), so the quantile method can be used. In practice, there are other transformation methods which are even faster, but rely on special tricks.

Since $Q_Z(U)$ has the same distribution function as Z, we can use the quantile method, as long as we can calculate Q_Z. Since Q_Z always exists, in principle this solves the problem. In practice, we need to calculate Q_Z before we can use it, and this may not have a closed form, and numerical approximations may be intractable.[3] In such situations, we turn to more advanced methods, like those described in Appendix M.

5.2.3 Sampling

A complement to drawing from given distributions is to **sample** from a given collection of objects. This is a common task, so R has a function to do it:

[3] In essence, we have to solve the nonlinear equation $F_Z(z) = p$ for z over and over for different p — and that assumes we can easily calculate F_Z.

```
sample(x, size, replace = FALSE, prob = NULL)
```

Here `x` is a vector which contains the objects we're going to sample from. `size` is the number of samples we want to draw from `x`. `replace` says whether the samples are drawn with or without replacement. (If `replace=TRUE`, then `size` can be arbitrarily larger than the length of `x`. If `replace=FALSE`, having a larger `size` doesn't make sense.) Finally, the optional argument `prob` allows for *weighted* sampling; ideally, `prob` is a vector of probabilities as long as `x`, giving the probability of drawing each element of `x`[4].

As a convenience for a common situation, running `sample` with one argument produces a random permutation of the input, i.e.,

```
sample(x)
```

is equivalent to

```
sample(x, size = length(x), replace = FALSE)
```

For example, the code for k-fold cross-validation, Code Example 3, had the lines

```
fold.labels <- sample(rep(1:nfolds, length.out = nrow(data)))
```

Here, `rep` repeats the numbers from 1 to `nfolds` until we have one number for each row of the data frame, say $1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2$ if there were twelve rows. Then `sample` shuffles the order of those numbers randomly. This then would give an assignment of each row of `df` to one (and only one) of five folds.

5.2.3.1 Sampling Rows from Data Frames

When we have multivariate data (which is the usual situation), we typically arrange it into a data-frame, where each row records one unit of observation, with multiple interdependent columns. The natural notion of sampling is then to draw a random sample of the data points, which in that representation amounts to a random sample of the rows. We can implement this simply by sampling row *numbers*. For instance, this command,

```
df[sample(1:nrow(df), size = b), ]
```

will create a new data frame from `b`, by selecting `b` rows from `df` without replacement. It is an easy exercise to figure out how to sample from a data frame *with* replacement, and with unequal probabilities per row.

[4] If the elements of `prob` do not add up to 1, but are positive, they will be normalized by their sum, e.g., setting `prob=c(9,9,1)` will assign probabilities $(\frac{9}{19}, \frac{9}{19}, \frac{1}{19})$ to the three elements of `x`.

5.2.3.2 Multinomials and Multinoullis

If we want to draw one value from a multinomial distribution with probabilities $p = (p_1, p_2, \ldots p_k)$, then we can use sample:

```
sample(1:k, size = 1, prob = p)
```

If we want to simulate a "multinoulli" process[5], i.e., a sequence of independent and identically distributed multinomial random variables, then we can easily do so:

```
rmultinoulli <- function(n, prob) {
    k <- length(prob)
    return(sample(1:k, size = n, replace = TRUE, prob = prob))
}
```

Of course, the labels needn't be the integers $1 : k$ (exercise 5.1).

5.2.3.3 Probabilities of Observation

Often, our models of how the data are generated will break up into two parts. One part is a model of how actual variables are related to each other out in the world. (E.g., we might model how education and racial categories are related to occupation, and occupation is related to income.) The other part is a model of how variables come to be recorded in our data, and the distortions they might undergo in the course of doing so. (E.g., we might model the probability that someone appears in a survey as a function of race and income.) Plausible sampling mechanisms often make the probability of appearing in the data a function of some of the variables. This can then have important consequences when we try to draw inferences about the whole population or process from the sample we happen to have seen (see, e.g., App. K).

```
income <- rnorm(n, mean = predict(income.model, x), sd = sigma)
capture.probabilities <- predict(observation.model, x)
observed.income <- sample(income, size = b, prob = capture.probabilities)
```

5.3 Repeating Simulations

Because simulations are often most useful when they are repeated many times, R has a command to repeat a whole block of code:

```
replicate(n, expr)
```

Here `expr` is some executable "expression" in R, basically something you could type in the terminal, and `n` is the number of times to repeat it.

For instance,

[5] A handy term I learned from Gustavo Lacerda.

```
output <- replicate(1000, rnorm(length(x), beta0 + beta1 * x, sigma))
```

will replicate, 1000 times, sampling from the predictive distribution of a Gaussian linear regression model. Conceptually, this is equivalent to doing something like

```
output <- matrix(0, nrow = 1000, ncol = length(x))
for (i in 1:1000) {
    output[i, ] <- rnorm(length(x), beta0 + beta1 * x, sigma)
}
```

but the `replicate` version has two great advantages. First, it is faster, because R processes it with specially-optimized code. (Loops are especially slow in R.) Second, and *far* more importantly, it is *clearer*: it makes it obvious what is being done, in one line, and leaves the computer to figure out the boring and mundane details of how best to implement it.

5.4 Why Simulate?

There are three major uses for simulation: to understand a model, to check it, and to fit it. We will deal with the first two here, and return to fitting in Chapter 26, after we've looked at dealing with dependence and hidden variables.

5.4.1 Understanding the Model; Monte Carlo

We understand a model by seeing what it predicts about the variables we care about, and the relationships between them. Sometimes those predictions are easy to extract from a mathematical representation of the model, but often they aren't. With a model we can simulate, however, we can just run the model and see what happens.

Our stochastic model gives a distribution for some random variable Z, which in general is a complicated, multivariate object with lots of interdependent components. We may also be interested in some complicated function g of Z, such as, say, the ratio of two components of Z, or even some nonparametric curve fit through the data points. How do we know what the model says about g?

Assuming we can make draws from the distribution of Z, we can find the distribution of any function of it we like, to as much precision as we want. Suppose that $\tilde{Z}_1, \tilde{Z}_2, \ldots \tilde{Z}_b$ are the outputs of b independent runs of the model — b different *replicates* of the model. (The tilde is a reminder that these are just simulations.) We can calculate g on each of them, getting $g(\tilde{Z}_1), g(\tilde{Z}_2), \ldots g(\tilde{Z}_b)$. If averaging makes sense for these values, then

$$\frac{1}{b}\sum_{i=1}^{b} g(\tilde{Z}_i) \xrightarrow[b\to\infty]{} \mathbb{E}\left[g(Z)\right] \tag{5.7}$$

by the law of large numbers. So simulation and averaging lets us get expectation

values. This basic observation is the seed of the **Monte Carlo method**.[6] If our simulations are independent, we can even use the central limit theorem to say that $\frac{1}{b}\sum_{i=1}^{b} g(\tilde{Z}_i)$ has approximately the distribution $\mathcal{N}(\mathbb{E}[g(Z)], \mathbb{V}[g(Z)]/b)$. Of course, if you can get expectation values, you can also get variances. (This is handy if trying to apply the central limit theorem!) You can also get any higher moments — if, for whatever reason, you need the kurtosis, you just have to simulate enough.

You can also pick any set s and get the probability that $g(Z)$ falls into that set:

$$\frac{1}{b}\sum_{i=1}^{b} \mathbf{1}_s(g(\tilde{Z}_i)) \xrightarrow[b\to\infty]{} \Pr(g(Z) \in s) \tag{5.8}$$

The reason this works is of course that $\Pr(g(Z) \in s) = \mathbb{E}[\mathbf{1}_s(g(Z))]$, and we can use the law of large numbers again. So we can get the whole distribution of any complicated function of the model that we want, as soon as we can simulate the model. It is really only a little harder to get the complete sampling distribution than it is to get the expectation value, and the exact same ideas apply.

5.4.2 Checking the Model

An important but under-appreciated use for simulation is to *check* models after they have been fit. If the model is right, after all, it represents the mechanism which generates the data. This means that when we simulate, we run that mechanism, and the surrogate data which comes out of the machine should look like the real data. More exactly, the real data should look like a typical realization of the model. If it does not, then the model's account of the data-generating mechanism is systematically wrong in some way. By carefully choosing the simulations we perform, we can learn a lot about how the model breaks down and how it might need to be improved.[7]

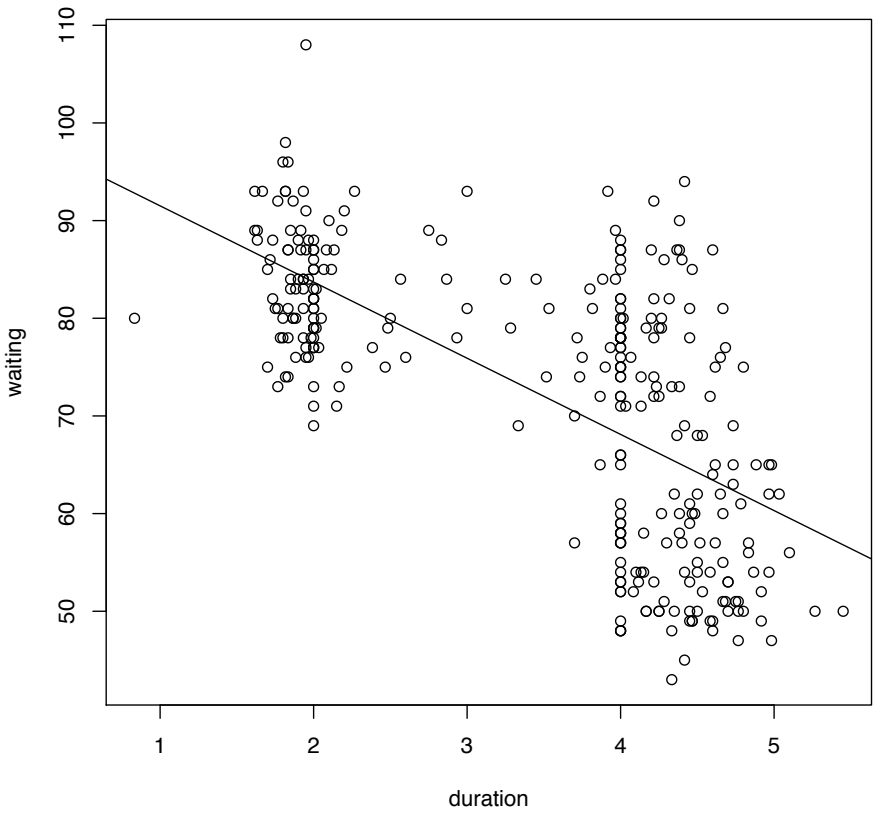
5.4.2.1 "Exploratory" Analysis of Simulations

Often the comparison between simulations and data can be done qualitatively and visually. For example, a classic data set concerns the time between eruptions of the Old Faithful geyser in Yellowstone, and how they relate to the duration of the latest eruption. A common exercise is to fit a regression line to the data by ordinary least squares:

```
library(MASS)
data(geyser)
fit.ols <- lm(waiting ~ duration, data = geyser)
```

[6] The name was coined by the physicists who used the method to do calculations relating to designing the hydrogen bomb; see Metropolis *et al.* (1953). Folklore among physicists says that the method goes back at least to Enrico Fermi in the 1930s, without the cutesy name.

[7] "Might", because sometimes (e.g., §1.4.2) we're better off with a model that makes systematic mistakes, if they're small and getting it right would be a hassle.




```
plot(geyser$duration, geyser$waiting, xlab = "duration", ylab = "waiting")
abline(fit.ols)
```

Figure 5.1 Data for the `geyser` data set, plus the OLS regression line.

Figure 5.1 shows the data, together with the OLS line. It doesn't look that great, but if someone insisted it was a triumph of quantitative vulcanology, how could you show they were wrong?

We'll consider general tests of regression specifications in Chapter 9. For now, let's focus on the way OLS is usually presented as part of a stochastic model for the response conditional on the input, with Gaussian and homoskedastic noise. In this case, the stochastic model is $\texttt{waiting} = \beta_0 + \beta_1 \texttt{duration} + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma^2)$. If we simulate from this probability model, we'll get something we can

```
rgeyser <- function() {
    n <- nrow(geyser)
    sigma <- summary(fit.ols)$sigma
    new.waiting <- rnorm(n, mean = fitted(fit.ols), sd = sigma)
    new.geyser <- data.frame(duration = geyser$duration, waiting = new.waiting)
    return(new.geyser)
}
```

CODE EXAMPLE 6: *Function for generating surrogate data sets from the linear model fit to* `geyser`.

compare to the actual data, to help us assess whether the scatter around that regression line is really bothersome. Since OLS doesn't require us to assume a distribution for the input variable (here, `duration`), the simulation function in Code Example 6 leaves those values alone, but regenerates values of the response (`waiting`) according to the model assumptions.

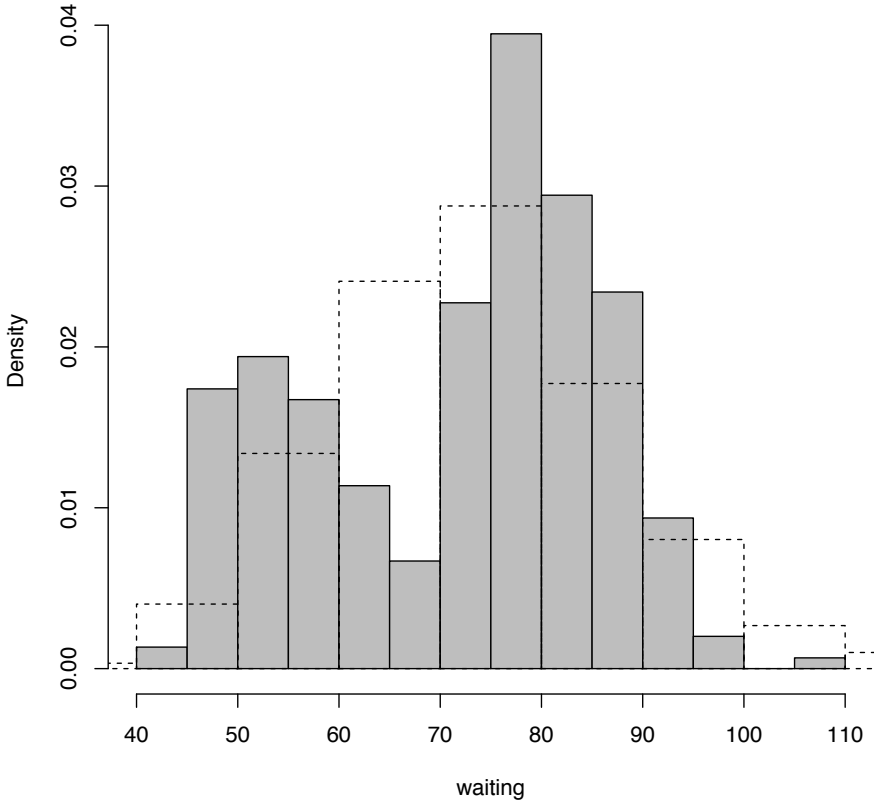
A useful principle for model checking is that if we do some exploratory data analyses of the real data, doing the same analyses to realizations of the model should give roughly the same results (Gelman, 2003; Hunter *et al.*, 2008; Gelman and Shalizi, 2013). This is a test the model fails. Figure 5.2 shows the actual histogram of `waiting`, plus the histogram produced by simulating — reality is clearly bimodal, but the model is unimodal. Similarly, Figure 5.3 shows the real data, the OLS line, and a simulation from the OLS model. It's visually clear that the deviations of the real data from the regression line are both bigger and more patterned than those we get from simulating the model, so something is wrong with the latter.

By itself, just seeing that data doesn't look like a realization of the model isn't super informative, since we'd really like to know *how* the model's broken, and so how to fix it. Further simulations, comparing more detailed analyses of the data to analyses of the simulation output, are often very helpful here. Looking at Figure 5.3, we might suspect that one problem is heteroskedasticity — the variance isn't constant. This suspicion is entirely correct, and will be explored in §10.3.2.

5.4.3 Sensitivity Analysis

Often, the statistical inference we do on the data is predicated on certain assumptions about how the data is generated. We've talked a lot about the Gaussian-noise assumptions that usually accompany linear regression, but there are many others. For instance, if we have missing values for some variables and just ignore incomplete rows, we are implicitly assuming that data are "missing at random", rather than in some systematic way that would carry information about *what* the missing values were (see App. K). Often, these assumptions make our analysis much neater than it otherwise would be, so it would be *convenient* if they were true.

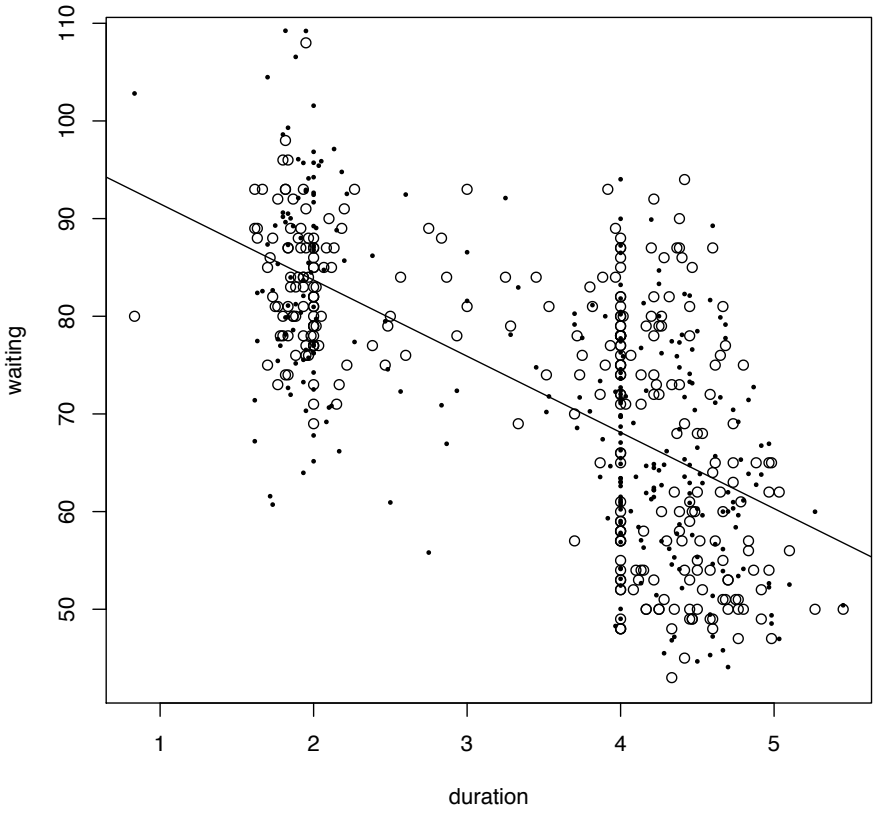
As a wise man said long ago, "The method of 'postulating' what we want has




```
hist(geyser$waiting, freq = FALSE, xlab = "waiting", main = "", sub = "", col = "grey")
lines(hist(rgeyser()$waiting, plot = FALSE), freq = FALSE, lty = "dashed")
```

Figure 5.2 Actual density of the waiting time between eruptions (grey bars, solid lines) and that produced by simulating the OLS model (dashed lines).

many advantages; they are the same as the advantages of theft over honest toil" (Russell, 1920, ch. VII, p. 71). In statistics, honest toil often takes the form of **sensitivity analysis**, of seeing how much our conclusions would change if the assumptions were violated, i.e., of checking how sensitive our inferences are to the assumptions. In principle, this means setting up models where the assumptions are more or less violated, or violated in different ways, analyzing them as though the assumptions held, and seeing how badly wrong we go. Of course, if that




```
plot(geyser$duration, geyser$waiting, xlab = "duration", ylab = "waiting")
abline(fit.ols)
points(rgeyser(), pch = 20, cex = 0.5)
```

Figure 5.3 As in Figure 5.1, plus one realization of simulating the OLS model (small black dots).

was easy to do in closed form, we often wouldn't have needed to make those assumptions in the first place.

On the other hand, it's usually pretty easy to *simulate* a model where the assumption is violated, run our original, assumption-laden analysis on the simulation output, and see what happens. Because it's a simulation, we know the complete truth about the data-generating process, and can assess how far off our inferences are. In favorable circumstances, our inferences don't mess up too much

even when the assumptions we used to motivate the analysis are badly wrong. Sometimes, however, we discover that even tiny violations of our initial assumptions lead to large errors in our inferences. Then we either need to make some compelling case for those assumptions, or be *very* cautious in our inferences.

5.5 Further Reading

Simulation will be used in nearly every subsequent chapter. It is the key to the "bootstrap" technique for quantifying uncertainty (Ch. 6), and the foundation for a whole set of methods for dealing with complex models of dependent data (Ch. 26).

Many texts on scientific programming discuss simulation, including Press *et al.* (1992) and, using R, Jones *et al.* (2009). There are also many more specialized texts on simulation in various applied areas. It must be said that many references on simulation present it as almost completely disconnected from statistics and data analysis, giving the impression that probability models just fall from the sky. Guttorp (1995) is an excellent exception.

For further reading on methods of drawing random variables from a given distribution, on Monte Carlo, and on generating uniform random numbers, see Appendix M. For doing statistical inference by comparing simulations to data, see Chapter 26.

When all (!) you need to do is draw numbers from a probability distribution which isn't one of the ones built in to R, it's worth checking CRAN's "task view" on probability distributions, `https://cran.r-project.org/web/views/Distributions.html`.

For sensitivity analyses, Miller (1998) describes how to use modern optimization methods to actively search for settings in simulation models which break desired behaviors or conclusions. I have not seen this idea applied to sensitivity analyses for statistical models, but it really ought to be.

Exercises

5.1 Modify `rmultinoulli` from §5.2.3.2 so that the values in the output are not the integers from 1 to k, but come from a vector of arbitrary labels.