Chapter 5

The Bootstrap

We are now four chapters into a statistics class and have said basically nothing about uncertainty. This should seem odd, and may even be disturbing if you are very attached to your p-values and saying variables have "significant effects". It is time to remedy this, and talk about how we can quantify uncertainty for complex models. The key technique here is what's called **bootstrapping**, or **the bootstrap**.

5.1 Stochastic Models, Uncertainty, Sampling Distributions

Statistics is the branch of applied mathematics which studies ways of drawing inferences from limited and imperfect data. We want to know how a neuron in a rat's brain responds when one of its whiskers gets tweaked, or how many rats live in Pittsburgh, or how high the water will get under the $16^{mathrmth}$ Street bridge during May, or the typical course of daily temperatures in the city over the year, or the relationship between the number of birds of prey in Schenley Park in the spring and the number of rats the previous fall. We have some data on all of these things. But we know that our data is incomplete, and experience tells us that repeating our experiments or observations, even taking great care to replicate the conditions, gives more or less different answers every time. It is foolish to treat any inference from the data in hand as certain.

If all data sources were totally capricious, there'd be nothing to do beyond piously qualifying every conclusion with "but we could be wrong about this". A mathematical science of statistics is possible because while repeating an experiment gives different results, some kinds of results are more common than others; their relative frequencies are reasonably stable. We thus model the data-generating mechanism through probability distributions and stochastic processes. When and why we can use stochastic models are very deep questions, but ones for another time. If we *can* use them in our problem, quantities like the ones I mentioned above are represented as functions of the stochastic model, i.e., of the underlying probability distribution.

Since a function of a function is a "functional", and these quantities are functions of the true probability distribution function, we'll call these **functionals** or **statistical functionals**[1]. Functionals could be single numbers (like the total rat population), or vectors, or even whole curves (like the expected time-course of temperature over the year, or the regression of hawks now on rats earlier). Statistical inference becomes estimating those functionals, or testing hypotheses about them.

These estimates and other inferences are functions of the data values, which means that they inherit variability from the underlying stochastic process. If we "re-ran the tape" (as the late, great Stephen Jay Gould used to say), we would get different data, with a certain characteristic distribution, and applying a fixed procedure would yield different inferences, again with a certain distribution. Statisticians want to use this distribution to quantify the uncertainty of the inferences. For instance, the standard error is an answer to the question "By how much would our estimate of this functional vary, typically, from one replication of the experiment to another?" (It presumes a particular meaning for "typically vary", as root mean square deviation around the mean.) A confidence region on a parameter, likewise, is the answer to "What are all the values of the parameter which *could* have produced this data with at least some specified probability?", i.e., all the parameter values under which our data are not low-probability outliers. The confidence region is a promise that *either* the true parameter point lies in that region, *or* something very unlikely under any circumstances happened — or that our stochastic model is wrong.

To get things like standard errors or confidence intervals, we need to know the distribution of our estimates around the true values of our functionals. These **sampling distributions** follow, remember, from the distribution of the data, since our estimates are functions of the data. Mathematically the problem is well-defined, but actually *computing* anything is another story. Estimates are typically complicated functions of the data, and mathematically-convenient distributions may all be poor approximations to the data source. Saying anything in closed form about the distribution of estimates can be simply hopeless. The two classical responses of statisticians were to focus on tractable special cases, and to appeal to asymptotics.

Your introductory statistics courses mostly drilled you in the special cases. From one side, limit the kind of estimator we use to those with a simple mathematical form — say, means and other linear functions of the data. From the other, assume that the probability distributions featured in the stochastic model take one of a few forms for which exact calculation *is* possible, analytically or via tabulated special functions. Most such distributions have origin myths: the Gaussian arises from averaging many independent variables of equal size (say, the many genes which contribute to height in humans); the Poisson distribution comes from counting how many of a large number of independent and individually-improbable events have occurred (say, radioactive nuclei decaying in a given second), etc. Squeezed from both ends, the sampling distribution of estimators and other functions of the data becomes exactly calculable in terms of the aforementioned special functions.

That these origin myths invoke various limits is no accident. The great results

[1]Most writers in theoretical statistics just call them "parameters" in a generalized sense, but I will try to restrict that word to actual parameters specifying statistical models, to minimize confusion. I may slip up.

of probability theory — the laws of large numbers, the ergodic theorem, the central limit theorem, etc. — describe limits in which *all* stochastic processes in broad classes of models display the same asymptotic behavior. The central limit theorem, for instance, says that if we average more and more independent random quantities with a common distribution, and that common distribution isn't too pathological, then the average becomes closer and closer to a Gaussian[2] Typically, as in the CLT, the limits involve taking more and more data from the source, so statisticians use the theorems to find the asymptotic, large-sample distributions of their estimates. We have been especially devoted to re-writing our estimates as averages of independent quantities, so that we can use the CLT to get Gaussian asymptotics.

Up through about the 1960s, statistics was split between developing general ideas about how to draw and evaluate inferences with stochastic models, and working out the properties of inferential procedures in tractable special cases (especially the linear-and-Gaussian case), or under asymptotic approximations. This yoked a very broad and abstract theory of inference to very narrow and concrete practical formulas, an uneasy combination often preserved in basic statistics classes.

The arrival of (comparatively) cheap and fast computers made it feasible for scientists and statisticians to record lots of data and to fit models to it, so they did. Sometimes the models were conventional ones, including the special-case assumptions, which often enough turned out to be detectably, and consequentially, wrong. At other times, scientists wanted more complicated or flexible models, some of which had been proposed long before, but now moved from being theoretical curiosities to stuff that could run overnight[3]. In principle, asymptotics might handle either kind of problem, but convergence to the limit could be unacceptably slow, especially for more complex models.

By the 1970s, then, statistics faced the problem of quantifying the uncertainty of inferences without using either implausibly-helpful assumptions or asymptotics; all of the solutions turned out to demand *even more* computation. Here we will examine what may be the most successful solution, Bradley Efron's proposal to combine estimation with simulation, which he gave the less-that-clear but persistent name of "the bootstrap" (Efron, 1979).

5.2 The Bootstrap Principle

Remember (from baby stats.) that the key to dealing with uncertainty in parameters and functionals is the sampling distribution of estimators. Knowing what distribution we'd get for our estimates on repeating the experiment would give us things like standard errors. Efron's insight was that we can *simulate* replication. After all, we have already fitted a model to the data, which is a guess at the mechanism which generated the data. Running that mechanism generates simulated data which, by hypothesis, has the same distribution as the real data. Feeding the simulated data through

[2]The reason is that the non-Gaussian parts of the distribution wash away under averaging, but the average of two Gaussians is another Gaussian.

[3]Kernel regression, kernel density estimation, and nearest neighbors prediction were all proposed in the 1950s, but didn't begin to be widely used until the 1970s, or even the 1980s.

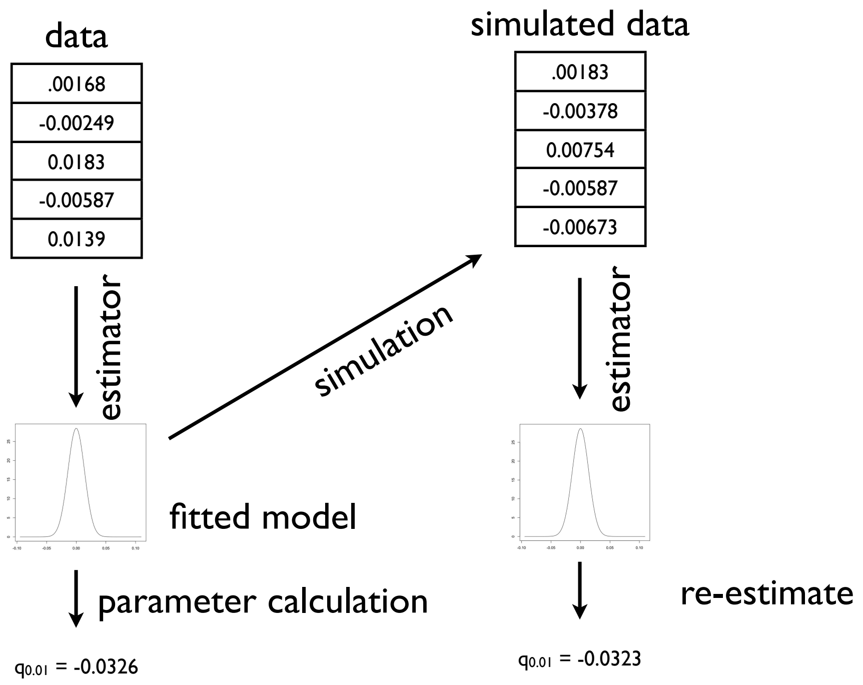


Figure 5.1: Schematic for model-based bootstrapping: simulated values are generated from the fitted model, then treated like the original data, yielding a new estimate of the functional of interest, here called $q_{0.01}$.

our estimator gives us one draw from the sampling distribution; repeating this many times yields the sampling distribution. Since we are using the model to give us its own uncertainty, Efron called this "bootstrapping"; unlike the Baron Munchhausen's plan for getting himself out of a swamp by pulling himself out by his bootstraps, it works.

Figure 5.1 sketches the over-all process: fit a model to data, use the model to calculate the functional, then get the sampling distribution by generating new, synthetic data from the model and repeating the estimation on the simulation output.

To fix notation, we'll say that the original data is x. (In general this is a whole data frame, not a single number.) Our parameter estimate from the data is $\hat{\theta}$. Surrogate data sets simulated from the fitted model will be $\tilde{X}_1, \tilde{X}_2, \ldots \tilde{X}_B$. The corresponding re-estimates of the parameters on the surrogate data are $\tilde{\theta}_1, \tilde{\theta}_2, \ldots \tilde{\theta}_B$. The functional of interest is estimated by the statistic T, with sample value $\hat{t} = T(x)$, and values of the surrogates of $\tilde{t}_1 = T(\tilde{X}_1)$, $\tilde{t}_2 = T(\tilde{X}_2)$, $\ldots \tilde{t}_B = T(\tilde{X}_B)$. (The statistic T may be a direct function of the estimated parameters, and only indirectly a function of x.) Everything which follows applies without modification when the functional of interest *is* the parameter, or some component of the parameter.

In this section, we will assume that the model is correct for *some* value of θ, which we will call θ_0. The true (population or ensemble) values of the functional is likewise

```
rboot <- function(B, statistic, simulator) {
  tboots <- replicate(B, statistic(simulator()))
  return(tboots)
}


bootstrap.se <- function(simulator, statistic, B) {
  tboots <- rboot(B, statistic, simulator)
  se <- sd(tboots)
  return(se)
}
```

Code Example 3: Sketch of code for calculating bootstrap standard errors. The function `rboot` generates `B` bootstrap samples (using the `simulator` function) and calculates the statistic g on them (using `statistic`). `simulator` needs to be a function which returns a surrogate data set in a form suitable for `statistic`. (How would you modify the code to pass arguments to `simulator` and/or `statistic`?) Because every use of bootstrapping is going to need to do this, it makes sense to break it out as a separate function, rather than writing the same code many times (with many chances of getting it wrong). `bootstrap.se` just calls `rboot` and takes a standard deviation.

t_0.

5.2.1 Variances and Standard Errors

The simplest thing to do is to get the variance or standard error:

$$\widehat{\mathrm{Var}}\left[\hat{t}\right] = \mathrm{Var}\left[\tilde{t}\right] \quad (5.1)$$

$$\widehat{\mathrm{se}}(\hat{t}) = \mathrm{sd}(\tilde{t}) \quad (5.2)$$

That is, we approximate the variance of our estimate of t_0 under the true but unknown distribution θ_0 by the variance of re-estimates $\tilde{t}$ on surrogate data from the fitted model $\widehat{\theta}$. Similarly we approximate the true standard error by the standard deviation of the re-estimates. The logic here is that the simulated $\tilde{X}$ has about the same distribution as the real X that our data, x, was drawn from, so applying the same estimation procedure to the surrogate data gives us the sampling distribution. This assumes, of course, that our model is right, and that $\hat{\theta}$ is not too far from θ_0.

Pseudo-code is provided in Code Example 3.

5.2.2 Bias Correction

We can use bootstrapping to correct for a biased estimator. Since the sampling distribution of $\tilde{t}$ is close to that of $\widehat{t}$, and $\widehat{t}$ itself is close to t_0,

$$\mathbf{E}\left[\widehat{t}\right] - t_0 \approx \mathbf{E}\left[\tilde{t}\right] - \widehat{t} \quad (5.3)$$

```
bootstrap.bias <- function(simulator, statistic, B,
  t.hat) {
  tboots <- rboot(B, statistic, simulator)
  bias <- mean(tboots) - t.hat
  return(bias)
}
```

Code Example 4: Sketch of code for bootstrap bias correction. Arguments are as in Code Example 3, except that `t.hat` is the estimate on the original data.

The left hand side is the bias that we want to know, and the right-hand side the was what we can calculate with the bootstrap.

Note, in fact, that Eq. 5.3 remains valid so long as the sampling distribution of $\widehat{t} - t_0$ is close to that of $\tilde{t} - \widehat{t}$. This is a weaker requirement than asking for $\widehat{t}$ and $\tilde{t}$ themselves to have similar distributions, or asking for $\widehat{t}$ to be close to t_0. In statistical theory, a random variable whose distribution does not depend on the parameters is called a **pivot**. (The metaphor is that it stays in one place while the parameters turn around it.) A sufficient (but not necessary) condition for Eq. 5.3 to hold is that $\widehat{t} - t_0$ be a pivot, or approximately pivotal.

5.2.3 Confidence Intervals

A confidence interval is a random interval which contains the truth with high probability (the confidence level). If the confidence interval for g is C, and the confidence level is $1 - \alpha$, then we want

$$\Pr(t_0 \in C) = 1 - \alpha \tag{5.4}$$

no matter what the true value of t_0. When we calculate a confidence interval, our inability to deal with distributions exactly means that the true confidence level, or **coverage** of the interval, is not quite the desired confidence level $1 - \alpha$; the closer it is, the better the approximation, and the more accurate the confidence interval.

Call the upper and lower limits of the confidence interval are C_u and C_l. Moreover, let's say that we want this to be symmetric, with probability $\alpha/2$ on either side of the confidence interval. (This is why we called the confidence level $1 - \alpha$, instead of α.) Then, looking at the lower limit,

$$\begin{aligned} \alpha/2 &= \Pr\left(C_l \geq t_0\right) & (5.5)\\ &= \Pr\left(C_l - \hat{t} \geq t_0 - \hat{t}\right) & (5.6)\\ &= \Pr\left(\hat{t} - C_l \leq \hat{t} - t_0\right) & (5.7) \end{aligned}$$

Likewise, at the upper end,

$$\alpha/2 = \Pr\left(\hat{t} - C_u \geq \hat{t} - t_0\right) \tag{5.8}$$

The point of these manipulations is that bootstrapping gives us the distribution of $\tilde{t} - \hat{t}$, which is approximately the same as the distribution of $\hat{t} - t_0$. Knowing that

```
bootstrap.ci.basic <- function(simulator, statistic, B,
  t.hat, alpha) {
  tboots <- rboot(B,statistic, simulator)
  ci.lower <- 2*t.hat - quantile(tboots,1-alpha/2)
  ci.upper <- 2*t.hat - quantile(tboots,alpha/2)
  return(list(ci.lower=ci.lower,ci.upper=ci.upper))
}
```

Code Example 5: Sketch of code for calculating the basic bootstrap confidence interval. See Code Examples 4 and 3.

distribution, and $\hat{t}$, we can solve for the confidence limits C_l and C_u:

$$\begin{aligned} C_l &= \hat{t} - \left(Q_{\tilde{t}}(1-\alpha/2) - \hat{t}\right) & (5.9) \\ C_u &= \hat{t} - \left(Q_{\tilde{t}}(\alpha/2) - \hat{t}\right) & (5.10) \end{aligned}$$

where $Q_{\tilde{t}}$ is the quantile function of $\tilde{t}$.

This is the **basic bootstrap confidence interval**, or the **pivotal** CI. It is simple and reasonably accurate, and makes a very good default choice for how to find a confidence interval.

Other Bootstrap Confidence Intervals

The basic bootstrap CI relies on the distribution of $\tilde{t} - \hat{t}$ being approximately the same as that of $\hat{t} - t_0$. Even when this is false, however, it can be that the distribution of

$$\tau = \frac{\hat{t} - t_0}{\widehat{\mathrm{se}}(\hat{t})} \tag{5.11}$$

is close to that of

$$\tilde{\tau} = \frac{\tilde{t} - \hat{t}}{\mathrm{se}(\tilde{t})} \tag{5.12}$$

This is like what we calculate in a t-test, and since the t-test was invented by "Student", these are called **studentized** quantities. If τ and $\tilde{\tau}$ have the same distribution, then we can reason as above and get a confidence interval

$$\left(\hat{t} - \widehat{\mathrm{se}}(\hat{t})Q_{\tilde{\tau}}(1-\alpha/2), \hat{t} - \widehat{\mathrm{se}}(\hat{t})Q_{\tilde{\tau}}(\alpha/2)\right) \tag{5.13}$$

This is the same as the basic interval when $\widehat{\mathrm{se}}(\hat{t}) = \mathrm{se}(\tilde{t})$, but different otherwise. To find $\mathrm{se}(\tilde{t})$, we need to actually do a *second* level of bootstrapping, as follows.

1. Fit the model with $\hat{\theta}$, find $\hat{t}$.
2. For $i \in 1:B_1$
    (a) Generate $\tilde{X}_i$ from $\hat{\theta}$

```
    (b) Estimate $\tilde{\theta}_i$, $\tilde{t}_i$
    (c) For $j \in 1 : B_2$
        i. Generate $X^\dagger_{ij}$ from $\tilde{\theta}_i$
        ii. Calculate $t^\dagger_{ij}$
    (d) Set $\tilde{\sigma}_i =$ standard deviation of the $t^\dagger_{ij}$
    (e) Set $\tilde{\tau}_{ij} = \frac{t^\dagger_{ij} - \tilde{t}_i}{\tilde{\sigma}_i}$ for all $j$
3. Set $\widehat{\mathrm{se}}(\hat{t}) =$ standard deviation of the $\tilde{t}_i$
4. Find the $\alpha/2$ and $1 - \alpha/2$ quantiles of the distribution of the $\tilde{\tau}$
5. Plug into Eq. 5.13.
```

The advantage of the studentized intervals is that they are more accurate than the basic ones; the disadvantage is that they are a lot more work! At the other extreme, the **percentile method** simply sets the confidence interval to

$$\left(Q_{\tilde{t}}(\alpha/2), Q_{\tilde{t}}(1-\alpha/2)\right) \tag{5.14}$$

This is definitely easier to calculate, but not as accurate.

All of these methods have many variations, described in the monographs referred to at the end of this chapter.

5.2.4 Hypothesis Testing

For hypothesis tests, we may want to calculate two sets of sampling distributions: the distribution of the test statistic under the null tells us about the size of the test and significance levels, and the distribution under the alternative tells about power and realized power. We can find either with bootstrapping, by simulating from either the null or the alternative. In such cases, the statistic of interest, which I've been calling T, is the test statistic. Code Example 6 illustrates how to find a p-value by simulating under the null hypothesis. The same procedure would work to calculate power, only we'd need to simulate from the alternative hypothesis, and `testhat` would be set to the critical value of T separating acceptance from rejection, not the observed value.

Double bootstrap hypothesis testing

When the hypothesis we are testing involves estimated parameters, we may need to correct for this. Suppose, for instance, that we are doing a goodness-of-fit test. If we estimate our parameters on the data set, we adjust our distribution so that it matches the data. It is thus not surprising if it seems to fit the data well! (Essentially, it's the problem of evaluating performance by looking at in-sample fit, which is more or less where we began the course.)

Some test statistics have distributions which are not affected by estimating parameters, at least not asymptotically. In other cases, one can analytically come up

```
boot.pvalue <- function(test,simulator,B,testhat) {
  testboot <- rboot(B=B, statistic=test, simulator=simulator)
  p <- (sum(test >= testhat)+1)/(B+1)
  return(p)
}
```

Code Example 6: Bootstrap p-value calculation. `testhat` should be the value of the test statistic on the actual data. `test` is a function which takes in a data set and calculates the test statistic, under the presumption that large values indicate departure from the null hypothesis. Note the +1 in the numerator and denominator of the p-value — it would be more straightforward to leave them off, but this is a little more stable when B is comparatively small. (Also, it keeps us from ever reporting a p-value of exactly 0.)

with correction terms. When these routes are blocked, one uses a **double bootstrap**, where a second level of bootstrapping checks how much estimation improves the apparent fit of the model. This is perhaps most easily explained in pseudo-code (Code Example 7).

5.2.5 Parametric Bootstrapping Example: Pareto's Law of Wealth Inequality

The Pareto distribution[4], or power-law distribution, is a popular model for data with "heavy tails", i.e. where the probability density $f(x)$ goes to zero only very slowly as $x \rightarrow \infty$. The probability density is

$$f(x) = \frac{\theta - 1}{x_0} \left(\frac{x}{x_0} \right)^{-\theta} \tag{5.15}$$

where x_0 is the minimum scale of the distribution, and θ is the **scaling exponent**. (EXERCISE: show that x_0 is the mode of the distribution.) The Pareto is highly right-skewed, with the mean being much larger than the median.

If we know x_0, one can show that the maximum likelihood estimator of the exponent θ is

$$\hat{\theta} = 1 + \frac{n}{\sum_{i=1}^{n} \log \frac{x_i}{x_0}} \tag{5.16}$$

and that this is consistent[5], and efficient. Picking x_0 is a harder problem (see Clauset *et al.* 2009) — for the present purposes, pretend that the Oracle tells us. The file `pareto.R`, on the class website, contains a number of functions related to the Pareto distribution, including a function `pareto.fit` for estimating it. (There's an example of its use below.)

[4]Named after Vilfredo Pareto, the highly influential late-19th/early-20th century economist, political scientist, and proto-Fascist.

[5]Because the sample mean of $\log X$ converges, under the law of large numbers

```
doubleboot.pvalue <- function(test,simulator,B1,B2,
  estimator, thetahat, testhat) {
  for (i in 1:B1) {
    xboot <- simulator(theta=thetahat, ...)
    thetaboot <- estimator(xboot)
    testboot[i] <- test(xboot)
    pboot[i] <- boot.pvalue(test,simulator,B2,
      testhat=testboot[i],theta=thetaboot)
  }
  p <- (sum(testboot >= testhat)+1)/(B1+1)
  p.adj <- (sum(pboot <= p)+1)/(B1+1)
}
```

Code Example 7: Code sketch for "double bootstrap" significance testing. The inner or second bootstrap is used to calculate the distribution of nominal bootstrap p-values. For this to work, we need to draw our second-level bootstrap samples from $\tilde{\theta}$, the bootstrap re-estimate, not from $\hat{\theta}$, the data estimate. The code presumes the `simulator` function takes a `theta` argument allowing this.

Pareto came up with this density when he attempted to model the distribution of wealth. Approximately, but quite robustly across countries and time-periods, the upper tail of the distribution of income and wealth follows a power law, with the exponent varying as money is more or less concentrated among the very richest[6]. Figure 5.2 shows the distribution of net worth for the 400 richest Americans in 2003. Taking $x_0 = 9 \times 10^8$ (again, see Clauset *et al.* 2009), the number of individuals in the tail is 302, and the estimated exponent is $\hat{\theta} = 2.34$.

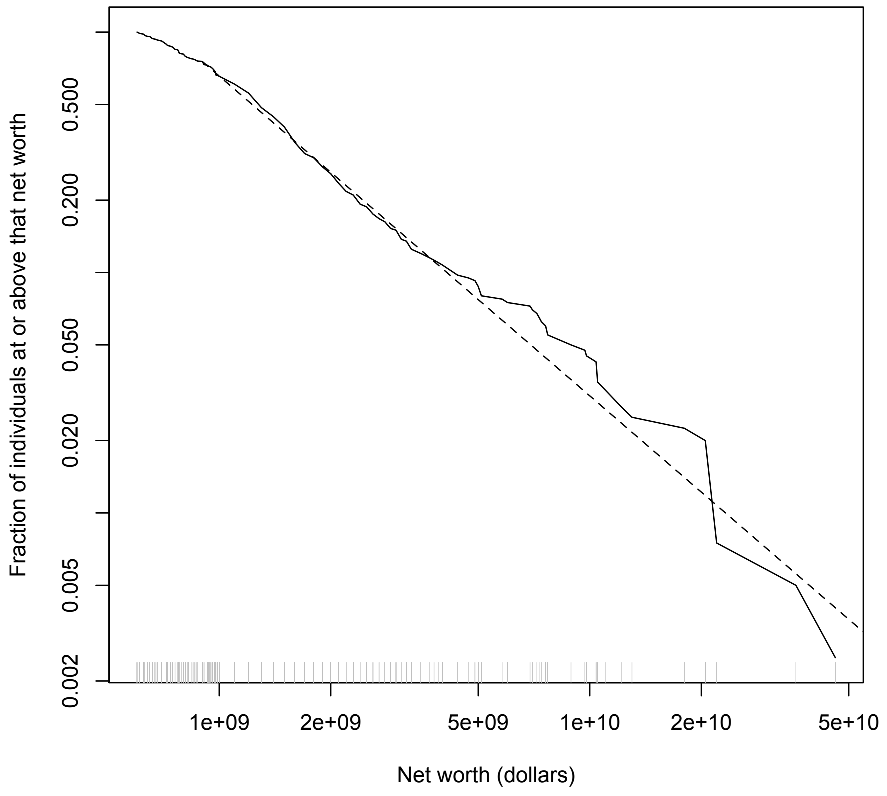
```
> source("pareto.R")
> wealth <- scan("wealth.dat")
> wealth.pareto <- pareto.fit(wealth,threshold=9e8)
> signif(wealth.pareto$exponent,3)
[1] 2.34
```

How much uncertainty is there in this estimate of the exponent? Naturally, we'll bootstrap. We need a function to generate Pareto-distributed random variables; this, along with some related functions, is part of the file `pareto.R` on the course website. With that tool, parametric bootstrapping proceeds as in Code Example 8.

With $\hat{\theta} = 2.34$, $x_0 = 9 \times 10^8$, $n = 302$ and $B = 10^4$, this gives a standard error of ± 0.077. This matches some asymptotic theory reasonably well[7], but didn't require asymptotic assumptions.

[6]Most of the distribution conforms to a log-normal, at least roughly.

[7]"In Asympotpia", the variance of the MLE should be $\frac{(\hat{\theta}-1)^2}{n}$, in this case 0.076. The intuition is that this variance depends on how sharp the maximum of the likelihood function is — if it's sharply peaked, we can find the maximum very precisely, but a broad maximum is hard to pin down. Variance is thus inversely proportional to the second derivative of the negative log-likelihood. (The minus sign is because the second derivative has to be negative at a maximum, while variance has to be positive.) For one



```
plot.survival.loglog(wealth,xlab="Net worth (dollars)",
  ylab="Fraction of individuals at or above that net worth")
rug(wealth,side=1,col="grey")
curve((302/400)*ppareto(x,threshold=9e8,exponent=2.34,lower.tail=FALSE),
      add=TRUE,lty=2,from=9e8,to=2*max(wealth))
```

Figure 5.2: Upper cumulative distribution function (or "survival function") of net worth for the 400 richest individuals in the US (2000 data). The solid line shows the fraction of the 400 individuals whose net worth W equaled or exceeded a given value w, $\Pr(W \geq w)$. (Note the logarithmic scale for both axes.) The dashed line is a maximum-likelihood estimate of the Pareto distribution, taking $x_0 = \$9 \times 10^8$. (This threshold was picked using the method of Clauset *et al.* 2009.) Since there are 302 individuals at or above the threshold, the cumulative distribution function of the Pareto has to be reduced by a factor of (302/400).

```
rboot.pareto <- function(B,exponent,x0,n) {
  replicate(B,pareto.fit(rpareto(n,x0,exponent),x0)$exponent)
}

pareto.se <- function(B,exponent,x0,n) {
  return(sd(rboot.pareto(B,exponent,x0,n)))
}

pareto.bias <- function(B,exponent,x0,n) {
  return(mean(rboot.pareto(B,exponent,x0,n)) - exponent)
}
```

Code Example 8: Standard error and bias calculation for the Pareto distribution, using parametric bootstrapping.

```
pareto.ci <- function(B,exponent,x0,n,alpha) {
  tboot <- rboot.pareto(B,exponent,x0,n)
  ci.lower <- 2*exponent - quantile(tboot,1-alpha/2)
  ci.upper <- 2*exponent - quantile(tboot,alpha/2)
  return(list(ci.lower=ci.lower, ci.upper=ci.upper))
}
```

Code Example 9: Parametric bootstrap confidence interval for the Pareto scaling exponent.

Asymptotically, the bias is known to go to zero; at this size, bootstrapping gives a bias of 3×10^{-3}, which is effectively negligible.

We can also get the confidence interval (Code Example 9). Using, again, 10^4 bootstrap replications, the 95% CI is $(2.16, 2.47)$. In theory, the confidence interval could be calculated exactly, but it involves the inverse gamma distribution (Arnold, 1983), and it is quite literally faster to write and do the bootstrap than go to look it up.

A more challenging problem is goodness-of-fit; we'll use the Kolmogorov-Smirnov statistic.[8] Code Example 10 calculates the p-value. With ten thousand bootstrap replications,

```
> ks.pvalue.pareto(1e4,wealth,2.34,9e8)
[1] 0.0119988
```

sample, the expected second derivative of the negative log-likelihood is $(\theta - 1)^{-2}$. (This is called the **Fisher information** of the model.) Log-likelihood adds across independent samples, giving us an over-all factor of n. In the large-sample limit, the actual log-likelihood will converge on the expected log-likelihood, so this gives us the asymptotic variance.

[8]The `pareto.R` file contains a function, `pareto.tail.ks.test`, which does a goodness-of-fit test for fitting a power-law to the tail of the distribution. That differs somewhat from what follows, because it takes into account the extra uncertainty which comes from having to estimate x_0. Here, I am pretending that an Oracle told us $x_0 = 9 \times 10^8$.

```
ks.stat.pareto <- function(data, exponent, x0) {
  data <- data[data>=x0]
  ks <- ks.test(data, ppareto, exponent=exponent,
    threshold=x0)
  return(ks$statistic)
}

ks.pvalue.pareto <- function(B, data, exponent, x0) {
  testhat <- ks.stat.pareto(data, exponent, x0)
  testboot <- vector(length=B)
  for (i in 1:B) {
    xboot <- rpareto(length(data),exponent=exponent,
     threshold=x0)
    exp.boot <- pareto.fit(xboot,threshold=x0)$exponent
    testboot[i] <- ks.stat.pareto(xboot,exp.boot,x0)
  }
  p <- (sum(testboot >= testhat)+1)/(B+1)
  return(p)
}
```

Code Example 10: Calculating a p-value for the Pareto distribution, using the Kolmogorov-Smirnov test and adjusting for the way estimating the scaling exponent moves the fitted distribution closer to the data.

Ten thousand replicates is enough that we should be able to accurately estimate probabilities of around 0.01 (since the binomial standard error will be $\sqrt{\frac{0.01}{0.99}10^4} \approx 9.9 \times 10^{-4}$; if it weren't, we might want to increase B.

Simply plugging in to the standard formulas, and thereby ignoring the effects of estimating the scaling exponent, gives a p-value of 0.16, which is not outstanding but not awful either. Properly accounting for the flexibility of the model, however, the discrepancy between what it predicts and what the data shows is so large that it would take an awfully big (one-a-hundred) coincidence to produce it.

We have, therefore, detected that the Pareto distribution makes systematic errors for this data, but we don't know much about what they are. In Chapter 17, we'll look at techniques which can begin to tell us something about *how* it fails.

5.3 Non-parametric Bootstrapping

The bootstrap approximates the sampling distribution, with three sources of approximation error. First, **simulation error**: using finitely many replications to stand for the full sampling distribution. Clever simulation design can shrink this, but brute force — just using enough replicates — can also make it arbitrarily small. Second, **statistical error**: the sampling distribution of the bootstrap re-estimates under our estimated model is not exactly the same as the sampling distribution of estimates under the true data-generating process. The sampling distribution changes with the parameters, and our initial estimate is not completely accurate. But it often turns out that distribution of estimates *around* the truth is more nearly invariant than the distribution of estimates themselves, so subtracting the initial estimate from the bootstrapped values helps reduce the statistical error; there are many subtler tricks to the same end. Third, **specification error**: the data source doesn't exactly follow our model at all. Simulating the model then never quite matches the actual sampling distribution.

Efron had a second brilliant idea, which is to address specification error by replacing simulation from the model with re-sampling from the data. After all, our initial collection of data gives us a lot of information about the relative probabilities of different values. In a sense the empirical distribution is the least prejudiced estimate possible of the underlying distribution — anything else imposes biases or pre-conceptions, possibly accurate but also potentially misleading[9]. Lots of quantities can be estimated directly from the empirical distribution, without the mediation of a parametric model. Efron's **non-parametric bootstrap** treats the original data set as a complete population and draws a new, simulated sample from it, picking each observation with equal probability (allowing repeated values) and then re-running the estimation (Figure 5.3). In fact, this is usually what people mean when they talk about "the bootstrap" without any modifier.

Everything we did with parametric bootstrapping can also be done with non-parametric bootstrapping — the only thing that's changing is the distribution the surrogate data is coming from.

The non-parametric bootstrap should remind you of k-fold cross-validation. The analog of leave-one-out CV is a procedure called the **jack-knife**, where we repeat the estimate n times on $n-1$ of the data points, holding each one out in turn. It's historically important (it dates back to the 1940s), but generally doesn't work as well as the non-parametric bootstrap.

An important variant is the **smoothed bootstrap**, where we re-sample the data points and then perturb each by a small amount of noise, generally Gaussian[10].

Code Example 11 shows how to use re-sampling to get a 95% confidence interval for the Pareto exponent[11]. With $B = 10^4$, it gives the 95% confidence interval for the scaling exponent as $(2.18, 2.48)$. The fact that this is very close to the interval we got from parametric bootstrapping should actually reassure us about its validity.

5.3.1 Parametric vs. Nonparametric Bootstrapping

When we have a properly specified model, simulating from the model gives more accurate results (at the same n) than does re-sampling the empirical distribution — parametric estimates of the distribution converge faster than the empirical distribution does. If on the other hand the parametric model is mis-specified, then it is rapidly

[9]See §15.6 in Chapter 15.

[10]We will see in Chapter 15 that this corresponds to sampling from a kernel density estimate

[11]Even if the Pareto model is wrong, the estimator of the exponent will converge on the value which gives, in a certain sense, the best approximation to the true distribution from among all power laws. Econometricians call such parameter values the**pseudo-true**; we are getting a confidence interval for the pseudo-truth. In this case, the pseudo-true scaling exponent can still be a useful way of summarizing *how* heavy tailed the income distribution is, despite the fact that the power law makes systematic errors.

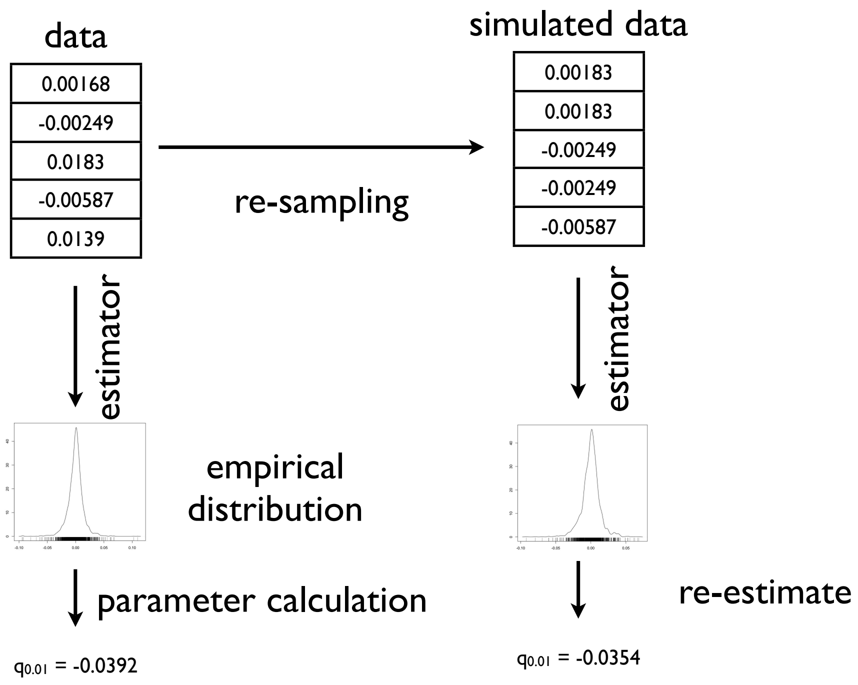


Figure 5.3: Schematic for non-parametric bootstrapping. New data is simulated by re-sampling from the original data (with replacement), and parameters are calculated either directly from the empirical distribution, or by applying a model to this surrogate data.

```
resample <- function(x) {
  sample(x,size=length(x),replace=TRUE)
}

resamp.pareto <- function(B,data,x0) {
  replicate(B,
   pareto.fit(resample(data),threshold=x0)$exponent)
}

resamp.pareto.CI <- function(B,data,alpha,x0) {
  thetahat <- pareto.fit(data,threshold=x0)$exponent
  thetaboot <- resamp.pareto(B,data,x0)
  ci.lower <- 2*thetahat - quantile(thetaboot,1-alpha/2)
  ci.upper <- 2*thetahat - quantile(thetaboot,alpha/2)
  return(list(ci.lower=ci.lower,ci.upper=ci.upper))
}
```

Code Example 11: Non-parametric bootstrap confidence intervals for the Pareto scaling exponent.

converging to the *wrong* distribution. This is of course just another bias-variance trade-off.

Since I am suspicious of most parametric modeling assumptions, I prefer re-sampling, when I can figure out how to do it, or at least until I have convinced myself that a parametric model is very good approximation to reality.

5.4 Bootstrapping Regression Models

With a regression model, which is fit to a set of input-output pairs, $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$, resulting in a regression curve (or surface) $\hat{r}(x)$, fitted values $\hat{y}_i = \hat{r}(x_i)$, and residuals, $\epsilon_i = y_i - \hat{y}_i = \hat{r}(x_i)$, we have a choice of several ways of bootstrapping, in decreasing order of relying on the model.

- Simulate new X values from the model's distribution of X, and then draw Y from the specified conditional distribution $Y|X$.
- Hold the x fixed, but draw $Y|X$ from the specified distribution.
- Hold the x fixed, but make Y equal to $\hat{r}(x)$ plus a randomly re-sampled ϵ_j.
- Re-sample (x, y) pairs.

The first case is pure parametric bootstrapping. (So is the second, sometimes, when the regression model is agnostic about X.) The last case is just re-sampling from the joint distribution of (X, Y). The next-to-last case is called **re-sampling the residuals** or **re-sampling the errors**. When we do that, we rely on the regression model to get the conditional expectation function right, but we don't count on it getting the distribution of the noise around the expectations.

The specific procedure of re-sampling the residuals is to re-sample the ϵ_i, with replacement, to get $\tilde{\epsilon}_1, \tilde{\epsilon}_2, \ldots \tilde{\epsilon}_n$, and then set $\tilde{x}_i = x_i$, $\tilde{y}_i = \hat{r}(\tilde{x}_i) + \tilde{\epsilon}_i$. This surrogate data set is then re-analyzed like new data.

5.4.1 Re-sampling Points: Parametric Example

A classic data set contains the time between 299 eruptions of the Old Faithful geyser in Yellowstone, and the length of the subsequent eruptions; these variables are called `waiting` and `duration`. We'll look at the linear regression of `waiting` on `duration`. We'll re-sample (`duration`, `waiting`) pairs, and would like confidence intervals for the regression coefficients. This is a confidence interval for the coefficients of *best liner predictor*, a functional of the distribution, which, as we saw in Chapters 1 and 2, exists no matter how nonlinear the process really is. It's only a confidence interval for the *true regression parameters* if the real regression function is linear.

Before anything else, look at the model:

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

The first step in bootstrapping this is to build our simulator, which just means sampling rows from the data frame:

```
resample.geyser <- function() {
  sample.rows <- resample(1:nrow(geyser))
  return(sample.rows)
}
```

We can check this by running `summary(geyser[resample.geyser(),]`, and seeing that it gives about the same quartiles and mean for both variables as `summary(geyser)`[12].

Next, we define the estimator:

```
est.waiting.on.duration <- function(subset,data=geyser) {
  fit <- lm(waiting ~ duration, data=data,subset=subset)
  return(coefficients(fit))
}
```

This exploits the fact that `lm()`, like many model-estimation functions, can take as an optional argument a vector of row numbers (`subset`), and look only at those rows of the data. We can check that this function works by seeing that `est.waiting.on.duration(1:n` gives the same results as `coefficients(geyser.lm)`.

Putting the pieces together according to the basic confidence interval recipe (Code Example 12), we get

[12]The minimum and maximum won't match up well — why not?

```
geyser.lm.cis <- function(B,alpha) {
  tboot <- replicate(B,
    est.waiting.on.duration(resample.geyser()))
  low.quantiles <- apply(tboot,1,quantile,probs=alpha/2)
  high.quantiles <- apply(tboot,1,quantile,probs=1-alpha/2)
  low.cis <- 2*coefficients(geyser.lm) - high.quantiles
  high.cis <- 2*coefficients(geyser.lm) - low.quantiles
  cis <- rbind(low.cis,high.cis)
  rownames(cis) <- as.character(c(alpha/2,1-alpha/2))
  return(cis)
}
```

Code Example 12: Bootstrapped confidence intervals for the linear model of Old Faithful, based on re-sampling data points. Relies on functions defined in the text.

```
> signif(geyser.lm.cis(B=1e4,alpha=0.05),3)
      (Intercept) duration
0.025        96.5    -8.70
0.975       102.0    -6.92
```

Notice that we do not have to assume homoskedastic Gaussian noise — fortunately, because that's a very bad assumption here[13].

[13]We have calculated 95% confidence intervals for the intercept β_0 and the slope β_1 separately. These intervals cover their coefficients all but 5% of the time. Taken together, they give us a rectangle in (β_0, β_1) space, but the coverage probability of *this* rectangle could be anywhere from 95% all the way down to 90%. To get a confidence *region* which simultaneously covers both coefficients 95% of the time, we have two big options. One is to stick to a box-shaped region and just increase the confidence level on each coordinate (to 97.5%). The other is to define some suitable metric of how far apart coefficient vectors are (e.g., ordinary Euclidean distance), find the 95% percentile of the distribution of this metric, and trace the appropriate contour around $\hat{\beta}_0, \hat{\beta}_1$.

5.4.2 Re-sampling Points: Non-parametric Example

Nothing in the logic of re-sampling data points for regression requires us to use a parametric model. Here we'll provide 95% confidence bounds for the kernel smoothing of the geyser data. Since the functional is a whole curve, the confidence set is often called a **confidence band**.

We use the same simulator, but start with a different regression curve, and need a different estimator.

```
library(np)
npr.waiting.on.duration <- function(subset,data=geyser,tol=0.1,ftol=0.1) {
  bw <- npregbw(waiting ~ duration, data=data, subset=subset,
    tol=tol, ftol=ftol)
  fit <- npreg(bw)
  return(fit)
}
geyser.npr <- npr.waiting.on.duration(1:nrow(geyser))
```

Now we construct pointwise 95% confidence bands for the regression curve. For this end, we don't really need to keep around the whole kernel regression object — we'll just use its predicted values on a uniform grid of points, extending slightly beyond the range of the data (Code Example 13). Observe that this will go through bandwidth selection again for each bootstrap sample. This is slow, but it is the most secure way of getting good confidence bands. Applying the bandwidth we found on the data to each re-sample would be faster, but would introduce an extra level of approximation, since we wouldn't be treating each simulation run the same as the original data.

Figure 5.4 shows the curve fit to the data, the 95% confidence limits, and (faintly) all of the bootstrapped curves. Doing the 800 bootstrap replicates took 4 minutes on my laptop[14].

[14]Specifically, I ran `system.time(geyser.npr.cis <- npr.cis(B=800,alpha=0.05))`, which not only did the calculations and stored them in `geyser.npr.cis`, but told me how much time it took R to do them.

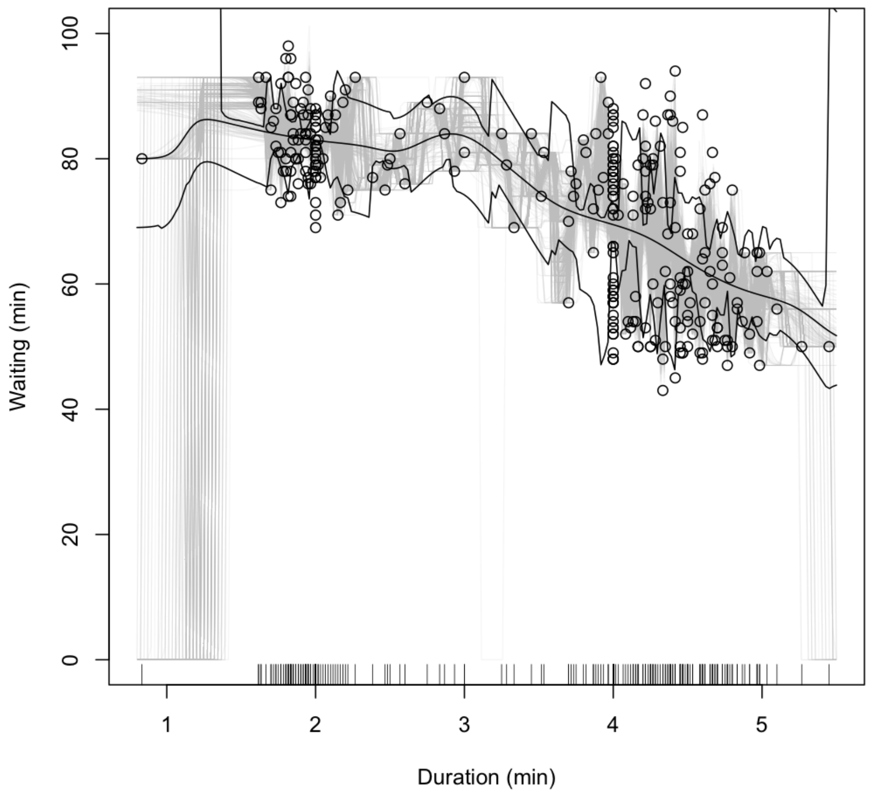
```
evaluation.points <- seq(from=0.8,to=5.5,length.out=200)
evaluation.points <- data.frame(duration=evaluation.points)

eval.npr <- function(npr) {
  return(predict(npr,newdata=evaluation.points))
}

main.curve <- eval.npr(geyser.npr)

npr.cis <- function(B,alpha) {
  tboot <- replicate(B,
    eval.npr(npr.waiting.on.duration(resample.geyser())))
  low.quantiles <- apply(tboot,1,quantile,probs=alpha/2)
  high.quantiles <- apply(tboot,1,quantile,probs=1-alpha/2)
  low.cis <- 2*main.curve - high.quantiles
  high.cis <- 2*main.curve - low.quantiles
  cis <- rbind(low.cis,high.cis)
  return(list(cis=cis,tboot=t(tboot)))
}
```

Code Example 13: Finding confidence bands around the kernel regression model of Old Faithful by re-sampling data points. Notice that much of `npr.cis` is the same as `geyser.lm.cis`, and the other functions for calculating confidence intervals. It would be better programming practice to extract the common find-the-confidence-limits part as a separate function, which could be called as needed. (Taking the transpose of the `tboot` matrix at the end is just so that it has the same orientation as the matrix of confidence limits.)



```
geyser.npr.cis <- npr.cis(B=800,alpha=0.05)
plot(0,type="n",xlim=c(0.8,5.5),ylim=c(0,100),
     xlab="Duration (min)", ylab="Waiting (min)")
for (i in 1:800) {
  lines(evaluation.points$duration,geyser.npr.cis$tboot[i,],
    lwd=0.1,col="grey")
}
lines(evaluation.points$duration,geyser.npr.cis$cis[1,])
lines(evaluation.points$duration,geyser.npr.cis$cis[2,])
lines(evaluation.points$duration,main.curve)
rug(geyser$duration,side=1)
points(geyser$duration,geyser$waiting)
```

Figure 5.4: Kernel regression curve for Old Faithful (central black line), with 95% confidence bands (other black lines), the 800 bootstrapped curves (thin, grey lines), and the data points. Notice that the confidence bands get wider where there is less data. *Caution:* doing the bootstrap took 4 minutes to run on my computer.

5.4.3 Re-sampling Residuals: Example

As an example of re-sampling the residuals, rather than data points, let's take a linear regression, from the homework using the Penn World Tables, of `gdp.growth` on `log(gdp)`, `pop.growth`, `invest` and `trade`:

```
penn <- read.csv("http://www.stat.cmu.edu/~cshalizi/uADA/12/hw/02/penn-select.csv")
penn.formula <- "gdp.growth ~ log(gdp) + pop.growth + invest + trade"
penn.lm <- lm(penn.formula, data=penn)
```

(Why make the formula a separate object here?) The estimated parameters are

```
> signif(coefficients(penn.lm),3)
(Intercept)    log(gdp)  pop.growth      invest       trade
   5.71e-04    5.07e-04   -1.87e-01    7.15e-04    3.11e-05
```

Code Example 14 shows the new simulator for this set-up (`resample.residuals.oecd`)[15], the new estimation function (`penn.estimator`)[16], and the confidence interval calculation (`penn.lm.cis`).

Which delivers our confidence intervals:

```
> signif(penn.lm.cis(1e4,0.05),3)
         (Intercept) log(gdp) pop.growth   invest     trade
low.cis      -0.0153 -0.00151     -0.358 0.000499 -2.00e-05
high.cis      0.0175  0.00240     -0.021 0.000937  8.19e-05
```

Doing ten thousand linear regressions took 45 seconds on my computer, as opposed to 4 minutes for eight hundred kernel regressions.

[15]How would you check that this was working right?
[16]How would you check that this was working right?

```
resample.residuals.penn <- function() {
  new.frame <- penn
  new.growths <- fitted(penn.lm) +
    resample(residuals(penn.lm))
  new.frame$gdp.growth <- new.growths
  return(new.frame)
}

penn.estimator <- function(data) {
  fit <- lm(penn.formula, data=data)
  return(coefficients(fit))
}

penn.lm.cis <- function(B,alpha) {
  tboot <- replicate(B,
    penn.estimator(resample.residuals.penn()))
  low.quantiles <- apply(tboot,1,quantile,probs=alpha/2)
  high.quantiles <- apply(tboot,1,quantile,probs=1-alpha/2)
  low.cis <- 2*coefficients(penn.lm) - high.quantiles
  high.cis <- 2*coefficients(penn.lm) - low.quantiles
  cis <- rbind(low.cis,high.cis)
  return(cis)
}
```

Code Example 14: Re-sampling the residuals to get confidence intervals in a linear model.

5.5 Bootstrap with Dependent Data

If the data point we are looking are vectors (or more complicated structures) with dependence between components, but each data point is independently generated from the same distribution, then dependence isn't really an issue. We re-sample vectors, or generate vectors from our model, and proceed as usual. In fact, that's what we've done so far in several cases.

If there is dependence *across* data points, things are more tricky. If our model incorporates this dependence, then we can just simulate whole data sets from it. An appropriate re-sampling method is trickier — just re-sampling individual data points destroys the dependence, so it won't do. We will revisit this question when we look at time series in Chapter 9, and again in Chapters 26–27.

5.6 Things Bootstrapping Does Poorly

The principle behind bootstrapping is that sampling distributions under the true process should be close to sampling distributions under good estimates of the truth. If small perturbations to the data-generating process produce huge swings in the sampling distribution, bootstrapping will not work well, and may fail spectacularly. For parametric bootstrapping, this means that small changes to the underlying parameters must produced small changes to the functionals of interest. Similarly, for non-parametric bootstrapping, it means that adding or removing a few data points must change the functionals only a little[17].

Re-sampling in particular has trouble with extreme values. Here is a simple example: Our data points X_i are IID, with $X_i \sim Unif(0, \theta_0)$, and we want to estimate θ_0. The maximum likelihood estimate $\hat{\theta}$ is just the sample maximum of the x_i. We'll use the non-parametric bootstrap to get a confidence interval for this, as above — but I will fix the true $\theta_0 = 1$, and see how often the 95% confidence interval covers the truth.

```
x <- runif(100)
is.covered <- function() {
  max.boot <- replicate(1e3,max(resample(x)))
  all(1 >= 2*max(x) - quantile(max.boot,0.975),
      1 <= 2*max(x) - quantile(max.boot,0.025))
}
sum(replicate(1000,is.covered()))
```

When I run the last line, I get 19, so the true coverage probability is not 95% but 1.9%.

If you suspect that your use of the bootstrap may be setting yourself up for a similar epic fail, your two options are (1) learn some of the theory of the bootstrap

[17]More generally, moving from one distribution function f to another $(1-\epsilon)f + \epsilon g$ mustn't change the functional very much when ϵ is small, no matter in what "direction" g we perturb it. Making this idea precise calls for some fairly deep mathematics, about differential calculus on spaces of functions.

from the references in the "Further Reading" section below, or (2) set up a simulation experiment like this one.

5.7 Further Reading

The original paper on the bootstrap, Efron (1979), is extremely clear, and for the most part presented in the simplest possible terms; it's worth reading. His later small book (Efron, 1982), while often cited, is not in my opinion so useful nowadays[18]. Davison and Hinkley (1997) is both a good textbook, and the reference I consult most often; the CRAN package `boot` is based on the code written for this book. Efron and Tibshirani (1993), while also very good, is more theoretical. Canty *et al.* (2006) has useful advice for serious applications.

5.8 Exercises

To think through, not to hand in.

1. Derive the maximum likelihood estimator for the Pareto distribution (Eq. 5.16) from the density (Eq. 5.15).
2. Find confidence bands for the linear regression model of using
   (a) The usual Gaussian assumptions (*hint*: try the `intervals="confidence"` option to `predict`)
   (b) Resampling of residuals
   (c) Resampling of cases

[18]It seems to have done a good job of explaining things to people who were already professional statisticians in 1982.