11

Computer Implementation

11.1 Introduction

The key requirements for computer implementation of resampling methods are a flexible programming language with a suite of reliable quasi-random number generators, a wide range of built-in statistical procedures to bootstrap, and a reasonably fast processor. In this chapter we outline how to use one implementation, using the current (May 1997) commercial version `Splus` 3.3 of the statistical language `S`, although the methods could be realized in a number of other statistical computing environments.

The remainder of this section outlines the installation of the library, and gives a quick summary of features of `Splus` essential to our purpose. Each subsequent section describes aspects of the library needed for the material in the corresponding chapter: Section 11.2 corresponds to Chapter 2, Section 11.3 to Chapter 3, and so forth. These sections take the form of a tutorial on the use of the library functions. The outline given here is not intended to replace the help files distributed with the library, which can be viewed by typing `help(boot,library="boot")` within `Splus`. At various points below, you will need to consult these files for more details on functions.

The main functions in the library are summarized in Table 11.1.

The best way to learn to use software is to use it, and from Section 11.1.2 onwards, we assume that you, dear reader, know the basics of `S`, including how to write simple functions, that you are seated comfortably at your favourite computer with `Splus` launched and a graphics window open, and that you are working through this chapter. We do not show the `Splus` prompt >, nor the continuation prompt .

Table 11.1 Functions in the Splus bootstrap library.

Function	Purpose
`abc.ci`	Nonparametric ABC confidence intervals
`boot`	Parametric and nonparametric bootstrap simulation
`boot.array`	Array of indices or frequencies from bootstrap simulation
`boot.ci`	Bootstrap confidence intervals
`censboot`	Bootstrap for censored and survival data
`control`	Control methods for estimation of quantiles, bias, variance, etc.
`cv.glm`	Cross-validation prediction error estimate for generalized linear model
`empinf`	Calculate empirical influence values
`envelope`	Calculate simulation envelope
`exp.tilt`	Exponential tilting to calculate probability distributions
`glm.diag`	Generalized linear model diagnostics
`glm.diag.plot`	Plot generalized linear model diagnostics
`imp.moments`	Importance resampling moment estimates
`imp.prob`	Importance resampling tail probability estimates
`imp.quantile`	Importance resampling quantile estimates
`imp.weights`	Calculate importance resampling weights
`jack.after.boot`	Jackknife-after-bootstrap plot
`linear.approx`	Calculate linear approximation to a statistic
`saddle`	Saddlepoint approximation
`saddle.distn`	Saddlepoint approximation for a distribution
`simplex`	Simplex method of linear programming
`smooth.f`	Frequency smoothing
`tilt.boot`	Automatic importance re-weighting bootstrap simulation
`tsboot`	Bootstrap for time series data

11.1.1 Installation

UNIX

The bootstrap library can be obtained from the home page for this book,

```
http://statwww.epfl.ch/davison/BMA/
```

in the form of a compressed `shar` file `bootlib.sh.Z`. This file should be uncompressed and moved to an appropriate directory. The file can then be unpacked by

```
sh bootlib.sh
rm bootlib.sh
```

You should then follow the instructions in the `README` file to complete the installation of the library.

It is best to set up an `Splus` library `boot` containing the library files; you may need to ask your system manager to do this. Once this is done, and once inside `Splus` in your usual working directory, the functions and data are accessed by typing

```
library(boot,first=T)
```

This will avoid cluttering your working directory with library files, and reduce the chance that you accidentally overwrite them.

Windows

The library functions and documentation for use with `Splus for Windows` can also be retrived in the form of a `zip` file from the home page for the book given above. For instructions on the installation, see the file `README.TXT`.

11.1.2 Some key `Splus` ideas

Quasi-random numbers

To put 20 quasi-random $N(0,1)$ data into `y` and to see its contents, type

```
y <- rnorm(20)
y
```

Here `<-` is the assignment symbol. To see the contents of any S object, simply type its name, as above. This is often done below, and we do not show the output.

In general quasi-random numbers from a distribution are generated by the functions `rexp`, `rgamma`, `rchisq`, `rt`,..., with arguments to give parameters where needed. For example,

```
y <- rgamma(n=10,shape=2)
```

generates 10 gamma observations with shape parameter 2, and

```
y <- rgamma(n=10,shape=c(1:10))
```

generates a vector of ten gamma variables with shape parameters $1, 2, \ldots, 10$.

The function `sample` is used to sample from a set with or without replacement. For example, to get a random permutation of the numbers $1, \ldots, 10$, a random sample with replacement from them, a random permutation of 11, 22, 33, 44, 55, a sample of size 10 from them, and a sample of size 10 taken with unequal probabilities:

```
sample(10)
sample(10,replace=T)
set <- c(11,22,33,44,55)
sample(set)
sample(set,size=10,replace=T)
sample(set,size=10,replace=T,prob=c(0.1,0.1,0.1,0.1,0.6))
```

Subscripts

The city population data with $n = 10$ are

```
city
city$u
city$x
```

where the second two commands show the individual variables of `city`. This `Splus` object is a dataframe — an array of data in which rows correspond to cases, and the named columns to variables. Elements of an object are accessed by subscripts, so

```
city$x[1]
city$x[c(1:4)]
city$x[c(1,5,10)]
city[c(1,5,10),2]
city$x[-1]
city[c(1:3),]
```

give various subsets of the elements of `city`. To make a nonparametric bootstrap sample of the rows of `city`, you could type:

```
i <- sample(10,replace=T)
city[i,]
```

The row labels result from the algorithm used to give unique labels to rows, and can be ignored for our purposes.

11.2 Basic Bootstraps

11.2.1 Nonparametric bootstrap

The main bootstrap function, `boot`, works on a vector, a matrix, or a dataframe. A simple use of `boot` to bootstrap the ratio $t = \bar{x}/\bar{u}$ for the city population data of Example 2.8 is

```
city.fun <- function(data, i)
{ d <- data[i,]
  mean(d$x)/mean(d$u) }
city.boot <- boot(data=city, statistic=city.fun, R=50)
```

The function `city.fun` takes as input the dataframe `data` and the vector of indices `i`. Its first command sets up the bootstrapped dataframe, and its second makes and returns the bootstrapped ratio. The last command instructs the function `boot` to bootstrap the data in `city` $R = 50$ times, apply the statistic `city.fun` to each bootstrap dataset and put the results in `city.boot`.

Bootstrap objects

The result of a call to boot is a bootstrap object. This is implemented as a list of quantities which is given the class "boot" and for which various methods are defined. For example, typing

```
city.boot
```

prints the original statistic, its estimated bias and its standard error, while

```
plot(city.boot)
```

gives suitable summary plots.

To see the names of the elements of the bootstrap object city.boot, type

```
names(city.boot)
```

You see various names, of which city.boot$t0, city.boot$t, city.boot$R, city.boot$seed contain the original value of the statistic, the bootstrap values, the value of R, and the value of the Splus random number generation seed when boot was invoked. To see their contents, type their names.

Timing

To repeat the simulation, checking how long it takes, type

```
unix.time(city.boot <- boot(city,city.fun,R=50))
```

on a UNIX system or

```
dos.time(city.boot <- boot(city,city.fun,R=50))
```

on a DOS system. The first number returned is the time the simulation took, and is useful for estimating how long a larger simulation would take.

Although code is generally clearer when dataframes are used, the computation can be speeded up by avoiding them, as here:

```
mat <- as.matrix(city)
mat.fun <- function(data, i)
{ d <- data[i,]
  mean(d[,2])/mean(d[,1]) }
unix.time(mat.boot <- boot(mat,mat.fun,R=50))
```

Compare this with the time taken using the dataframe city.

Frequency array

To obtain the $R \times n$ array of bootstrap frequencies for city.boot and to display its first 20 lines, type

```
f <- boot.array(city.boot)
f[1:20,]
```

The rows of `f` are the vectors of frequencies for individual bootstrap samples. The array is useful for many *post hoc* calculations, and is invoked by post-processing functions such as `jack.after.boot` and `imp.weight`, which are discussed below. It is calculated from `city.boot$seed`. The array of indices for the bootstrap samples can be obtained by `boot.array(city.boot, indices=T)`.

Types of statistic

For a nonparametric bootstrap, the function `statistic` can be of one of three types. We have already seen examples of the first, index type, where the arguments are the dataframe `data` and the vector of indices, `i`; this is specified by `stype="i"` (the default).

For the second, weighted type, the arguments are `data` and a vector of weights `w`. For example,

```
city.w <- function(data, w=rep(1,nrow(data))/nrow(data))
{ w <- w/sum(w)
  sum(w*data$x)/sum(w*data$u)}
city.boot <- boot(city, city.w, R=20, stype="w")
```

writes

$$\frac{\bar{x}^*}{\bar{u}^*} = \frac{\sum w_j^* x_j / \sum w_j^*}{\sum w_j^* u_j / \sum w_j^*},$$

where w_j^* is the weight put on the jth case of the dataframe in the bootstrap sample; the first line of `city.w` ensures that $\sum w_j^* = 1$. Setting `w` in the initial line of the function gives the default value for `w`, which is a vector of n^{-1}s; this enables the original value of t to be obtained by `city.w(city)`. A more complicated example is given by the library correlation function `corr`. Not all statistics can be written in this form, but when they can, numerical differentiation can be used to obtain empirical influence values and *ABC* confidence intervals.

For the third, frequency type, the arguments are `data` and a vector of frequencies `f`. For example,

```
city.f <- function(data, f) mean(f*data$x)/mean(f*data$u)
city.boot <- boot(city, city.f, R=20, stype="f")
```

uses

$$\frac{\bar{x}^*}{\bar{u}^*} = \frac{n^{-1}\sum f_j^* x_j}{n^{-1}\sum f_j^* u_j},$$

where f_j^* is the frequency with which the jth row of the dataframe occurs in the bootstrap sample. Not all statistics can be written in this form. It differs from the preceding type in that whereas weights can in principle take any positive values, frequencies must be integers. Of course in this example it would be easiest to use the function `city.fun` given earlier.

Function `statistic`

The contents of `statistic` can be more-or-less arbitrarily complicated, provided that its output is a scalar or fixed-length vector. For example,

```
air.fun <- function(data, i)
{ d <- data[i,]
  c(mean(d), var(d)/nrow(data)) }
air.boot <- boot(data=aircondit, statistic=air.fun, R=200)
```

performs a nonparametric bootstrap for the average of the air-conditioning data, and returns the bootstrapped averages and their estimated variances. We give more complex examples below. Beware of memory and storage problems if you make the output too long.

By default the first element of `statistic` (and so the first column of `boot.out$t`) is treated as the main statistic for certain calculations, such as calculation of empirical influence values, the jackknife-after-bootstrap plot, and confidence interval calculations, which are described below. This is changed by use of the `index` argument, usually a single number giving the column of `statistic` to which the calculation is to be applied.

Further arguments can be passed to `statistic` using the `...` argument to `boot`. For example,

```
city.subset <- function(data, i, n=10)
{  d <- data[i[1:n],]
   mean(d[,2])/mean(d[,1]) }
city.boot <- boot(data=city, statistic=city.subset, R=200, n=5)
```

gives resampled ratios for bootstrap samples of size 5. Note that the frequency array for `city.boot` would not be useful in this case. The indices can be obtained by

```
boot.array(city.boot,indices=T)[,1:5]
```

11.2.2 Parametric bootstrap

For a parametric bootstrap, the first argument to `statistic` remains a vector, matrix, or dataframe, but `statistic` need take no second argument. Instead three further arguments to `boot` must be supplied. The first, `ran.gen`, tells `boot` how to simulate bootstrap data, and is a function that takes two arguments, the original data, and an object containing any other parameters, `mle`. The output of `ran.gen` should have the same form and attributes as the original dataset. The second new argument to `boot` is a value for `mle` itself. The third new argument to boot, sim="parametric", tells boot to perform a parametric simulation: by default the simulation is nonparametric and sim="ordinary". Other possible values for sim are described below.

For example, for parametric simulation from the exponential model fitted to the air-conditioning data in Table 1.2, we set

```
aircondit.fun <- function(data) mean(data$hours)
aircondit.sim <- function(data, mle)
{ d <- data
  d$hours <- rexp(n=nrow(data), rate=mle)
  d }
aircondit.mle <- 1/mean(aircondit$hours)
aircondit.para <- boot(data=aircondit, statistic=aircondit.fun,
                  R=20, sim="parametric", ran.gen=aircondit.sim,
                  mle=aircondit.mle)
```

Air-conditioning data for a different aircraft are given in aircondit7. Obtain their sample average, and perform a parametric bootstrap of the average using the fitted exponential model. Give the bias and variance estimates for the average. Do the bootstrapped averages look normal for this sample size?

A more complicated example is parametric simulation based on a log bivariate normal distribution fitted to the city population data:

```
l.city <- log(city)
city.mle <- c(apply(l.city,2,mean),sqrt(apply(l.city,2,var)),
              corr(l.city))
city.sim <- function(data, mle)
{ n <- nrow(data)
  d <- matrix(rnorm(2*n),n,2)
  d[,2] <- mle[2] + mle[4]*(mle[5]*d[,1]+sqrt(1-mle[5]^2)*d[,2])
  d[,1] <- mle[1] + mle[3]*d[,1]
  data$x <- exp(d[,2])
  data$u <- exp(d[,1])
  data }
city.f <- function(data) mean(data[,2])/mean(data[,1])
city.para <- boot(city, city.f, R=200, sim="parametric",
                  ran.gen=city.sim, mle=city.mle)
```

With this definition of city.f, a nonparametric bootstrap can be performed by

```
city.boot <- boot(data=city,
     statistic=function(data, i) city.f(data[i,]), R=200)
```

This is useful when comparing parametric and nonparametric bootstraps for the same problem. Compare them for the city data.

11.2.3 Empirical influence values

For a statistic boot.fun in weighted form, function empinf returns the empirical influence values l_j, obtained by numerical differentiation. For the ratio function city.w given above, for example, these and the exact values (Problem 2.9) are

```
L.diff <- empinf(data=city, statistic=city.w, stype="w")
cbind(L.diff,(city$x-city.w(city)*city$u)/mean(city$u))
```

Empirical influence values can also be obtained from the output of boot by regression of the values of t^* on the frequency array. For example,

```
city.boot  <- boot(city, city.fun, R=999)
L.reg <- empinf(city.boot)
L.reg
```

uses regression with the 999 samples in city.boot to estimate the l_j.

Jackknife values can be obtained by

```
J <- empinf(data=city,statistic=city.fun,stype="i",type="jack")
```

The argument type controls how the influence values are to be calculated, but this also depends on the quantities input to empinf: for details see the help file.

Variance approximations

var.linear uses empirical influence values to calculate the nonparametric delta method variance approximation for a statistic:

```
var.linear(L.diff)
var.linear(L.reg)
```

Linear approximation

linear.approx uses output from a nonparametric bootstrap simulation to calculate the linear approximations to the bootstrapped quantities. The empirical influence values can be supplied, but if not, they are estimated by a call to empinf. For the city population ratio,

```
city.tL.reg <- linear.approx(city.boot)
city.tL.diff <- linear.approx(city.boot, L=L.diff)
split.screen(c(1,2))
screen(1); plot(city.tL.reg,city.boot$t); abline(0,1,lty=2)
screen(2); plot(city.tL.diff,city.boot$t); abline(0,1,lty=2)
```

calculates the linear approximation for the two sets of empirical influence values and plots the actual t^* against them.

11.3 Further Ideas

11.3.1 Stratified sampling

Stratified sampling is performed by including argument `strata` in the call to `boot`. Suppose that we wish to bootstrap the difference in the trimmed averages for the last two groups of gravity data (Example 3.2):

```
gravity
grav <- gravity[as.numeric(gravity$series)>=7,]
grav
grav.fun <- function(data, i, trim=0.125)
{ d <- data[i,]
  m <- tapply(d$g, d$series, mean, trim=trim)
  m[7]-m[8] }
grav.boot <- boot(grav, grav.fun, R=200, strata=grav$series)
```

Check that the expected properties of `boot.array(grav.boot)` hold.

Empirical influence values, linear approximations, and nonparametric delta method variance approximations are calculated by

```
grav.L <- empinf(grav.boot)
grav.tL <- linear.approx(grav.boot)
var.linear(grav.L, strata=grav$series)
```

`grav.boot$strata` contains the strata used in the resampling, which are taken into account automatically if `grav.boot` is used, but otherwise must be supplied, as in the final line of the code above.

11.3.2 Smoothing

The neatest way to perform smooth bootstrapping is to use `sim="parametric"`. For example, to estimate the variance of the median of the data in `y`, using smoothing parameter $h = 0.5$:

```
y <- rnorm(99)
h <- 0.5
y.gen <- function(data, mle)
{  n <- length(data)
   i <- sample(n, n, replace=T)
   data[i] + mle*rnorm(n) }
```

```
y.boot <- boot(y, median, R=200, sim="parametric",
               ran.gen=y.gen, mle=h)
var(y.boot$t)
```

This guarantees that `y.boot$t0` contains the original median. For shrunk smoothing, see Practical 4.5.

11.3.3 Censored data

`censboot` is used to bootstrap censored data. Suppose that we wish to assess the variability of the median survival time and the probability of survival beyond 20 weeks for the first group of AML data (Example 3.9).

```
aml1 <- aml[aml$group==1,]
aml1.fun <- function(data)
{  surv <- survfit(Surv(data$time,data$cens))
   p1 <- min(surv$surv[surv$time<20])
   m1 <- min(surv$time[surv$surv<0.5])
   c(p1, m1) }
aml1.ord <- censboot(data=aml1, statistic=aml1.fun, R=50)
aml1.ord
```

This involves ordinary bootstrap resampling, and hence could be performed with `boot`, although `aml1.fun` would then have to be rewritten to have another argument. For conditional simulation, two additional arguments must be supplied containing the estimated survivor functions for the times to failure and the censoring distribution:

```
aml1.fail <- survfit(Surv(time,cens),data=aml1)
aml1.cens <- survfit(Surv(time-0.01*cens,1-cens),data=aml1)
aml1.con <- censboot(data=aml1, statistic=aml1.fun, R=50,
                 F.surv=aml1.fail, G.surv=aml1.cens, sim="cond")
```

11.3.4 Bootstrap diagnostics

Jackknife-after-bootstrap

Function `jack.after.boot` produces a jackknife-after-bootstrap plot of the first column of `boot.out$t` based on a nonparametric simulation. For example, for the `city` data ratio:

```
city.fun <- function(data, i)
 { d <- data[i,]
   rat <- mean(d$x)/mean(d$u)
   L <- (d$x-rat*d$u)/mean(d$u)
   c(rat, sum(L^2)/nrow(d)^2, L) }
```

```
city.boot <- boot(city, city.fun, R=999)
city.L <- city.boot$t0[3:12]
split.screen(c(1,2)); screen(1); split.screen(c(2,1)); screen(4)
attach(city)
plot(u,x,type="n",xlim=c(0,300),ylim=c(0,300))
text(u,x,round(city.L,2))
screen(3)
plot(u,x,type="n",xlim=c(0,300),ylim=c(0,300))
text(u,x,c(1:10)); abline(0,city.boot$t0[1],lty=2)
screen(2)
jack.after.boot(boot.out=city.boot, useJ=F, stinf=F, L=city.L)
close.screen(all=T)
```

The two left panels show the data with case numbers and empirical influence values as plotting symbols. The jackknife-after-bootstrap plot on the right shows the effect of deleting cases in turn: values of t^* are more variable when case 4 is deleted and less variable when cases 9 and 10 are deleted. We see from the empirical influence values that the distribution of t^* shifts downwards when cases with positive empirical influence values are deleted, and conversely.

This plot is also produced by setting true the `jack` argument to `plot` when applied to a bootstrap object, as in `plot(city.boot,jack=T)`.

Other arguments for `jack.after.boot` control whether the influence values are standardized (by default they are, `stinf=T`), whether the empirical influence values are used (by default jackknife values are used, based on the simulation, so the default values are `useJ=T` and `L=NULL`).

Most post-processing functions allow the user to specify either an index for the component of interest, or a vector of length `boot.out$R` to be treated as the main statistic. Thus a jackknife-after-bootstrap plot using the second component of `city.boot$t` — the estimated variances for t^* — would be obtained by either of

```
jack.after.boot(city.boot,useJ=F,stinf=F,index=2)
jack.after.boot(city.boot,useJ=F,stinf=F,t=city.boot$t[,2])
```

Frequency smoothing

`smooth.f` smooths the frequencies of a nonparametric bootstrap object to give a "typical" distribution with expected value roughly at θ. In order to find the smoothed frequencies for $\theta = 1.4$ for the city ratio, and to obtain the corresponding value of t, we set

```
city.freq <- smooth.f(theta=1.4, boot.out=city.boot)
city.w(city, city.freq)
```

The smoothing bandwidth is controlled by the width argument to smooth.f and is width$\times v^{1/2}$, where v is the estimated variance of t — width=0.5 by default.

11.4 Tests

11.4.1 Parametric tests

Simple parametric tests can be conducted using parametric simulation. For example, to perform the conditional simulation for the data in fir (Example 4.2):

```
fir.mle <- c(sum(fir$count), nrow(fir))
fir.gen <- function(data, mle)
{ d <- data
  y <- sample(x=mle[2],size=mle[1],replace=T)
  d$count <- tabulate(y,mle[2])
  d }
fir.fun <- function(data)
 (nrow(data)-1)*var(data$count)/mean(data$count)
fir.boot <- boot(fir, fir.fun, R=999, sim="parametric",
                 ran.gen=fir.gen, mle=fir.mle)
qqplot(qchisq(c(1:fir.boot$R)/(fir.boot$R+1),df=49),fir.boot$t)
abline(0,1,lty=2); abline(h=fir.boot$t0)
```

The last two lines here display the results (almost) as in the right panel of Figure 4.1.

11.4.2 Permutation tests

Approximate permutation tests are performed by setting sim="permutation" when invoking boot. For example, suppose that we wish to perform a permutation test for independence of the two columns of dataframe ducks:

```
perm.fun <- function(data, i)  cor(data[,1],data[i,2])
ducks.perm <- boot(ducks, perm.fun, R=499, sim="permutation")
(sum(ducks.perm$t>ducks.perm$t0)+1)/(ducks.perm$R+1)
qqnorm(ducks.perm$t,ylim=c(-1,1))
abline(h=ducks.perm$t0,lty=2)
```

If strata is included in the call to boot, permutation is performed independently within each stratum.

11.4.3 Bootstrap tests

For a bootstrap test of the hypothesis of independence in the ducks data, we make a new dataframe and function:

```
duck <- c(ducks[,1],ducks[,2])
n <- nrow(ducks)
duck.fun <- function(data, i, n)
{ x <- data[i]
  cor(x[1:n],x[(n+1):(2*n)]) }
.Random.seed <- ducks.perm$seed
ducks.boot <- boot(duck, duck.fun, R=499,
                    strata=rep(c(1,2),c(n,n)), n=n)
(sum(ducks.boot$t>ducks.boot$t0)+1)/(ducks.boot$R+1)
```

This uses the same seed as for the permutation test, for a more precise comparison. Is the significance level similar to that for the permutation test?

Why cannot boot be directly applied to ducks to perform a bootstrap test?

Exponential tilting

The test of equality of means for two sets of data in Example 4.16 involves exponential tilting. The null distribution puts probabilities given by (4.25) on the two sets of data, and the tilt parameter λ solves the equation

$$\sum_{i=1}^{2} \frac{\sum_j z_{ij} \exp(\lambda z_{ij})}{\sum_j \exp(\lambda z_{ij})} = \theta,$$

where $z_{1j} = y_{1j}$, $z_{2j} = -y_{2j}$, and $\theta = 0$. The fitted null distribution is obtained using exp.tilt, as follows:

```
z <- grav$g
z[grav$series==8] <- -z[grav$series==8]
z.tilt <- exp.tilt(L=z, theta=0, strata=grav$series)
z.tilt
```

where z.tilt contains the fitted probabilities (which sum to one for each stratum) and the values of λ and θ. Other arguments can be input to exp.tilt: see its help file.

The significance probability is then obtained by using the weights argument to boot. This argument is a vector containing the probabilities with which to select the rows of data, when bootstrap sampling is to be performed with unequal probabilities. In this case the unequal probabilities are given by the tilted distribution, under which the expected value of the test statistic is zero. The code needed to perform the simulation and get the estimated significance level is:

```
grav.test <- function(data, i)
{ d <- data[i,]
  diff(tapply(d$g,d$series,mean))[7] }
grav.boot <- boot(data=grav, statistic=grav.test, R=999,
                  weights=z.tilt$p, strata=grav$series)
(sum(grav.boot$t>grav.boot$t0)+1)/(grav.boot$R+1)
```

11.5 Confidence Intervals

The main function for setting bootstrap confidence intervals is `boot.ci`, which takes as input a bootstrap object. For example, to get a 95% confidence interval for the ratio in the `city` data, using the `city.boot` object created in Section 11.3.4:

```
boot.ci(boot.out=city.boot)
```

By default the confidence level is 0.95, but other values can be obtained using the `conf` argument. Here invoking `boot.ci` shows the normal, basic, studentized bootstrap, percentile, and BC_a intervals. Subsets of these intervals are obtained using the `type` argument. For example, if `city.boot$t` only contained the ratio and not its estimated variance, it would be impossible to obtain the studentized bootstrap interval, and an appropriate use of `boot.ci` would be

```
boot.ci(boot.out=city.boot,type=c("norm","perc","basic","bca"),
        conf=c(0.8,0.9))
```

By default `boot.ci` assumes that the first and second columns of `boot.out$t` contain the statistic itself and its estimated variance; otherwise the `index` argument can be used, as outlined in the help file.

To calculate intervals for the parameter $h(\theta)$, and then back-transform them to the original scale, we use the `h`, `hinv`, and `hdot` arguments. For example, to calculate intervals for the city ratio, using $h(\cdot) = \log(\cdot)$, we set

```
boot.ci(city.boot, h=log, hinv=exp, hdot=function(u) 1/u)
```

where `hinv` and `hdot` are the inverse and first derivative of $h(\cdot)$. Note how transformation improves the basic bootstrap interval.

Nonparametric ABC intervals are calculated using `abc.ci`. For example

```
abc.ci(data=city, statistic=city.w)
```

calculates the 95% ABC interval for the city ratio; `statistic` must be in weighted form for this. As usual, strata are incorporated using the `strata` argument.

11.6 Linear Regression

11.6.1 Basic approaches

Resampling for linear regression models is performed using boot. It is simplest when bootstrapping cases. For example, to compare the biases and variances for parameter estimates from bootstrapping least squares and L_1 estimates for the mammals data:

```
fit.model <- function(data)
{  fit <- glm(log(brain)~log(body),data=data)
   l1 <- l1fit(log(data$body),log(data$brain))
   c(coef(fit), coef(l1)) }
mammals.fun <- function(data, i) fit.model(data[i,])
mammals.boot <- boot(mammals, mammals.fun, R=99)
mammals.boot
```

For model-based resampling it is simplest to set up an augmented dataframe containing the residuals and fitted values. Although the model is a straightforward linear model, we fit it using glm rather than lm so that we can calculate residuals using the library function glm.diag, which calculates various types of residuals, approximate Cook statistics, and measures of leverage for a glm object. (The diagnostics are exact for a linear model.) A related function is glm.diag.plots, which produces standard diagnostic plots for a generalized linear model fit:

```
mam.lm <- glm(log(brain)~log(body),data=mammals)
mam.diag <- glm.diag(mam.lm)
glm.diag.plots(mam.lm)
res <- (mam.diag$res-mean(mam.diag$res))*mam.diag$sd
mam <- data.frame(mammals,res=res,fit=fitted(mam.lm))
mam.fun <- function(data, i)
{ d <- data
  d$brain <- exp(d$fit+d$res[i])
  fit.model(d) }
mam.boot <- boot(mam, mam.fun, R=99)
mam.boot
```

Empirical influence values and the nonparametric delta method standard error for the slope of the linear model could be obtained by putting the slope estimate in weighted form:

```
mam.w <- function(data, w)
  coef(glm(log(data$brain)~log(data$body), weights=w))[2]
mam.L <- empinf(data=mammals, statistic=mam.w)
sqrt(var.linear(mam.L))
```

For more complicated regressions, for example with unequal response variances, more information must be added to the new dataframe.

Wild bootstrap

The wild bootstrap can be implemented using `sim="parametric"`, as follows:

```
mam.mle <- c(nrow(mam), (5+sqrt(5))/10)
mam.wild <- function(data, mle)
{ d <- data
  i <- 2*rbinom(mle[1], size=1, prob=1-mle[2])-1
  d$brain <- exp(d$fit+d$res*(1-i*sqrt(5))/2)
  d }
mam.boot.wild <- boot(mam, fit.model, R=20, sim="parametric",
                      ran.gen=mam.wild, mle=mam.mle)
```

11.6.2 Prediction

Now consider prediction of the log brain weight of new mammals with body weights equal to those for the chimpanzee and baboon. For this we introduce yet another argument to `boot` — `m`, which gives the number of ε^*_{rm} to be simulated with each bootstrap sample (see Algorithm 6.4). In this case we want to predict at $m = 2$ "new" mammals, with covariates contained in `d.pred`. The `statistic` function supplied to `boot` must now take at least one more argument, namely the additional indices for constructing the bootstrap versions of the two "new" mammals. We implement this as follows:

```
d.pred <- mam[c(46,47),]
pred <- function(data, d.pred)
  predict(glm(log(brain)~log(body),data=data), d.pred)
mam.pred <- function(data, i, i.pred, d.pred)
{ d <- data
  d$brain <- exp(d$fit+d$res[i])
  pred(d, d.pred) - (d.pred$fit + d$res[i.pred]) }
mam.boot.pred <- boot(mam, mam.pred, R=199, m=2, d.pred=d.pred)
orig <- matrix(pred(mam, d.pred),mam.boot.pred$R,2,byrow=T)
exp(apply(orig+mam.boot.pred$t,2,quantile,c(0.025,0.5,0.975)))
```

giving the 0.025, 0.5, and 0.975 prediction limits for the brain sizes of the "new" mammals. The actual brain sizes lie close to or above the upper limits of these intervals: primates tend to have larger brains than other mammals.

11.6.3 Aggregate prediction error and variable selection

Practical 6.5 shows how to obtain the various estimates of aggregate prediction error based on a given model.

For consistent bootstrap variable selection, a subset of size $n-m$ is used to fit each of the possible models. Consider Example 6.13, where a fake set of data is made by

```
x1 <- runif(50); x2 <- runif(50); x3 <- runif(50)
x4 <- runif(50); x5 <- runif(50); y <- rnorm(50)+2*x1+2*x2
fake <- data.frame(y,x1,x2,x3,x4,x5)
```

As in that example, we consider the six possible models with no covariates, with just x_1, with x_1, x_2, and so forth, finishing with $x_1, \ldots, x_5$. The function `subset.boot` fits these to a subset of `n-size` observations, and calculates the prediction mean squared error for all the data. It is then applied using `boot`:

```
subset.boot <- function(data, i, size=0)
{ n <- nrow(data)
  i.t <- i[1:(n-size)]
  data.t <- data[i.t,  ]
  res0 <- data$y - mean(data.t$y)
  lm.d <- lm(y ~ x1, data=data.t)
  res1 <- data$y - predict.lm(lm.d, data)
  lm.d <- update(lm.d, .~.+x2)
  res2 <- data$y - predict.lm(lm.d, data)
  lm.d <- update(lm.d, .~.+x3)
  res3 <- data$y - predict.lm(lm.d, data)
  lm.d <- update(lm.d, .~.+x4)
  res4 <- data$y - predict.lm(lm.d, data)
  lm.d <- update(lm.d, .~.+x5)
  res5 <- data$y - predict.lm(lm.d, data)
  meansq <- function(y) mean(y^2)
  apply(cbind(res0,res1,res2,res3,res4,res5),2,meansq)/n }
fake.boot.40 <- boot(fake, subset.boot, R=100, size=40)
delta.hat.40 <- apply(fake.boot.40$t,2,mean)
plot(c(0:5),delta hat 40,xlab="Number of covariates",
     ylab="Delta hat (M)",type="l",ylim=c(0,0.1))
```

For results with a different value of `size`, but re-using `fake.boot.40$seed` in order to reduce simulation variability:

```
.Random.seed <- fake.boot.40$seed
fake.boot.30 <- boot(fake, subset.boot, R=100, size=30)
delta.hat.30 <- apply(fake.boot.30$t,2,mean)
lines(c(0:5),delta.hat.30,lty=2)
```

Try this with various values of `size`.

Modify the code above to do variable selection using cross-validation, and compare it with the bootstrap results.

11.7 Further Topics in Regression

11.7.1 Nonlinear and generalized linear models

Nonlinear and generalized linear models are bootstrapped using the ideas in the preceding section. For example, to apply case resampling to the calcium data of Example 7.7:

```
calcium.fun <- function(data, i)
{ d <- data[i,]
  d.nls <- nls(cal~beta0*(1-exp(-time*beta1)),data=d,
               start=list(beta0=5,beta1=0.2))
  c(coefficients(d.nls),sum(d.nls$residuals^2)/(nrow(d)-2)) }
cal.boot <- boot(calcium,calcium.fun,R=19,strata=calcium$time)
```

Likewise, to apply model-based simulation to the leukaemia data of Example 7.1, resampling standardized deviance residuals according to (7.14),

```
leuk.glm <- glm(time~log10(wbc)+ag-1,Gamma(log),data=leuk)
leuk.diag <- glm.diag(leuk.glm)
muhat <- fitted(leuk.glm)
rL <- log(leuk$time/muhat)/sqrt(1-leuk.diag$h)
eps <- 10^(-4)
u <- -log(seq(from=eps,to=1-eps,by=eps))
d <- sign(u-1)*sqrt(2*(u-1-log(u)))/leuk.diag$sd
r.dev <- smooth.spline(d, u)
z <- predict(r.dev, leuk.diag$rd)$y
leuk.mle <- data.frame(muhat,rL,z)
fit.model <- function(data)
{  data.glm <- glm(time~log10(wbc)+ag-1,Gamma(log),data=data)
   c(coefficients(data.glm),deviance(data.glm)) }
leuk.gen <- function(data,mle)
{  i <- sample(nrow(data),replace=T)
   data$time <- mle$muhat*mle$z[i]
   data }
leuk.boot <- boot(leuk, fit.model, R=19, sim="parametric",
                  ran.gen=leuk.gen, mle=leuk.mle)
```

The other procedures for model-based resampling of generalized linear models are applied similarly. Try to modify this code to resample the linear predictor residuals according to (7.13) (they are already calculated above).

11.7.2 Survival data

Further arguments to censboot are needed to bootstrap survival data. For illustration, we consider the melanoma data of Example 7.6, and fit a model in which survival depends on log tumour thickness. The initial fits are given by

```
mel.cox <- coxph(Surv(time,status==1)~log(thickness)
                 +strata(ulcer),data=melanoma)
mel.surv <- survfit(mel.cox)
mel.cens <- survfit(Surv(time-0.01*(status!=1),status!=1)~1,
           data=melanoma)
```

The bootstrap function mel.fun given below need only take one argument, a dataframe containing the data themselves. Note how the function uses a smoothing spline to interpolate fitted values for the full range of thickness; this avoids difficulties due to the variability of the covariate when resampling cases. The output of mel.fun is the vector of fitted linear predictors predicted by the spline.

```
mel.fun <- function(d)
{ attach(d)
  cox <- coxph(Surv(time,status==1)~log(thickness)+strata(ulcer))
  eta <- unique(cox$linear.predictors)
  u <- unique(thickness)
  sp <- smooth.spline(u,eta,df=20)
  th <- seq(from=0.25,to=10,by=0.25)
  eta <- predict(sp,th)$y
  detach("d")
  eta }
```

The next three commands give the syntax for case resampling, for model-based resampling and for conditional resampling. For either of these last two schemes, the baseline survivor functions for the survival times and censoring times, and the fitted proportional hazards (Cox) model for the survival distribution must be supplied via the F.surv, G.surv, and cox arguments.

```
attach(melanoma)
mel.boot <- censboot(melanoma, mel.fun, R=99, strata=ulcer)
mel.boot.mod <- censboot(melanoma, mel.fun, R=99,
           F.surv=mel.surv, G.surv=mel.cens, strata=ulcer,
           cox=mel.cox, sim="model")
mel.boot.con <- censboot(melanoma, mel.fun, R=99,
           F.surv=mel.surv,  G.surv=mel.cens, strata=ulcer,
           cox=mel.cox, sim="cond")
```

The bootstrap results are best displayed graphically. Here is the code for the analogue of the left panels of Figure 7.9:

```
th <- seq(from=0.25,to=10,by=0.25)
split.screen(c(2,1))
screen(1)
plot(th,mel.boot$t0,type="n",xlab="Tumour thickness (mm)",
     xlim=c(0,10),ylim=c(-2,2),ylab="Linear predictor")
lines(th,mel.boot$t0,lwd=3)
rug(jitter(thickness))
for (i in 1:19) lines(th,mel.boot$t[i,],lwd=0.5)
screen(2)
plot(th,mel.boot$t0,type="n",xlab="Tumour thickness (mm)",
     xlim=c(0,10),ylim=c(-2,2),ylab="Linear predictor")
lines(th,mel.boot$t0,lwd=3)
mel.env <- envelope(mel.boot$t,level=0.95)
lines(th,mel.env$point[1,],lty=1)
lines(th,mel.env$point[2,],lty=1)
mel.env <- envelope(mel.boot.mod$t,level=0.95)
lines(th,mel.env$point[1,],lty=2)
lines(th,mel.env$point[2,],lty=2)
mel.env <- envelope(mel.boot.con$t,level=0.95)
lines(th,mel.env$point[1,],lty=3)
lines(th,mel.env$point[2,],lty=3)
detach("melanoma")
```

Note how tight the confidence envelope is relative to that for the more highly parametrized model used in the example. Try again with larger values of *R*, if you have the patience.

11.7.3 Nonparametric regression

Nonparametric regression is bootstrapped in the same way as other regressions. Consider for example bootstrapping the smoothing spline fit to the motorcycle data of Example 7.10. The data without repeats are in `motor`, with components `accel`, `times`, `strata`, and `v`, the last two of which give the strata for resampling and an estimated variance within each stratum. The three fits are obtained by

```
attach(motor)
motor.smooth <- smooth.spline(times,accel,w=1/v)
motor.small <- smooth.spline(times,accel,w=1/v,
                             spar=motor.smooth$spar/2)
motor.big <- smooth.spline(times,accel,w=1/v,
                           spar=motor.smooth$spar*2)
```

Commands to set up and perform the resampling are as follows:

```
res <- (motor$accel-motor.small$y)/sqrt(1-motor.small$lev)
motor.mle <- data.frame(bigfit=motor.big$y,res=res)
xpoints <- c(10,20,25,30,35,45)
motor.fun <- function(data, x)
{ y.smooth <- smooth.spline(data$times,data$accel,w=1/data$v)
  predict(y.smooth,x)$y }
motor.gen <- function(data, mle)
{ d <- data
  i <- c(1:nrow(data))
  i1 <- sample(i[data$strata==1],replace=T)
  i2 <- sample(i[data$strata==2],replace=T)
  i3 <- sample(i[data$strata==3],replace=T)
  d$accel <- mle$bigfit + mle$res[c(i1,i2,i3)]
  d }
motor.boot <- boot(motor, motor.fun, R=999, sim="parametric",
                   ran.gen=motor.gen, mle=motor.mle, x=xpoints)
```

Finally, the 90% basic bootstrap confidence limits are obtained by

```
mu.big <- predict(motor.big,xpoints)$y
mu <- predict(motor.smooth,xpoints)$y
ylims <- apply(motor.boot$t,2,quantile,c(0.05,0.95))
ytop <- mu - (ylims[1,]-mu.big)
ybot <- mu - (ylims[2,]-mu.big)
```

What is the effect of using a smaller smoothing parameter when calculating the residuals?

Try altering this code to apply the wild bootstrap, and see what effect it has on the results.

11.8 Time Series

Model-based resampling for time series is analogous to regression. We consider the sunspot data of Example 8.3, to which we fit the autoregressive model that minimizes AIC:

```
sun <- 2*(sqrt(sunspot+1)-1)
ts.plot(sun)
sun.ar <- ar(sun)
sun.ar$order
```

The best model is AR(9). How well determined is this, and what is the variance of the series average? We bootstrap to see, using

```
sun.fun <- function(tsb)
{ ar.fit <- ar(tsb, order.max=25)
  c(ar.fit$order, mean(tsb), tsb) }
```

which calculates the order of the fitted autoregressive model, the series average, and saves the series itself.

Our function for bootstrapping time series is tsboot. Here are results for fixed-block bootstraps with block length $l = 20$:

```
sun.1 <- tsboot(sun, sun.fun, R=99, l=20, sim="fixed")
tsplot(sun.1$t[1,3:291],main="Block simulation, l=20")
table(sun.1$t[,1])
var(sun.1$t[,2])
qqnorm(sun.1$t[,2])
```

The statistic for tsboot takes only one argument, the time series. The first plot here shows the results for a single replicate using block simulation: note the occasional big jumps in the resampled series. Note also the large variation in the orders of the fitted autoregressive models.

To obtain similar results for the stationary bootstrap with mean block length $l = 20$:

```
sun.2 <- tsboot(sun, sun.fun, R=99, l=20, sim="geom")
```

Are the results similar to having blocks of fixed length?

For model-based resampling we need to store results from the original model, and to make residuals from that fit:

```
sun.model <- list(order=c(sun.ar$order,0,0),ar=sun.ar$ar)
sun.res <- sun.ar$resid[!is.na(sun.ar$resid)]
sun.res <- sun.res - mean(sun.res)
sun.sim <- function(res,n.sim, ran.args)
{ rg1 <- function(n, res) sample(res, n, replace=T)
  ts.orig <- ran.args$ts
  ts.mod <- ran.args$model
  mean(ts.orig)+rts(arima.sim(model=ts.mod, n=n.sim,
    rand.gen=rg1, res=as.vector(res))) }
sun.3 <- tsboot(sun.res, sun.fun, R=99, sim="model", n.sim=114,
     ran.gen=sun.sim,ran.args=list(ts=sun, model=sun.model))
```

Check the orders of the fitted models for this scheme. Are they similar to those obtained using the block schemes above?

For "post-blackening" we need to define yet another function:

```
sun.black <- function(res, n.sim, ran.args)
{ ts.orig <- ran.args$ts
```

```
  ts.mod <- ran.args$model
  mean(ts.orig)+rts(arima.sim(model=ts.mod,n=n.sim,innov=res)) }
sun.1b <- tsboot(sun.res, sun.fun, R=99, l=20, sim="fixed",
  ran.gen=sun.black, ran.args=list(ts=sun, model=sun.model),
  n.sim=length(sun))
```

Compare these results with those above, and try it with `sim="geom"`.

11.9 Improved Simulation

11.9.1 Balanced resampling

The balanced bootstrap is invoked via the `sim` argument to boot:

```
city.bal <- boot(city, city.fun, R=20, sim="balanced")
```

If `strata` is supplied, balancing takes place separately within each stratum.

11.9.2 Control methods

`control` applies the control methods, including post-simulation balance, to the output from an existing bootstrap simulation. For example,

```
control(city.boot, bias.adj=T)
```

produces the adjusted bias estimate, while

```
city.con <- control(city.boot)
```

gives a list consisting of the regression estimates of the empirical influence values, linear approximations to the bootstrap statistics, the control estimates of bias, variance, and the third cumulant of the t^*, control estimates of selected quantiles of the distribution of t^*, and a spline object that summarizes the approximate quantiles used to obtain the control quantile estimates. Saddlepoint approximation is used to obtain these approximate quantiles. Typing

```
city.con$L
city.con$bias
city.con$var
city.con$quantiles
```

gives some of the above-mentioned quantities. Arguments to `control` allow the user to specify the empirical influence values, the spline object, and other quantities to be used by control, if they are already available; see the help file for details.

11.9.3 Importance resampling

We have already met a use of nonparametric simulation with unequal probabilities in Section 11.4, using the `weights` argument to `boot`. The simplest form for `weights`, used there, is a vector containing the probabilities with which to select the rows of `data`, when bootstrap sampling is to be performed with unequal probabilities. If we wish to perform importance resampling using several distributions, we can set them up and then perform the sampling as follows:

```
city.top <- exp.tilt(L=city.L, theta=2, t0=city.w(city))
city.bot <- exp.tilt(L=city.L, theta=1.2, t0=city.w(city))
city.tilt <- boot(city, city.fun, R=c(100,99),
                  weights=rbind(city.top$p,city.bot$p))
```

which performs 100 simulations from the probabilities in `city.top$p` and 99 from the probabilities in `city.bot$p`. In the first two lines `exp.tilt` is used to solve the equation

$$t_0 + \frac{\sum_j l_j \exp(\lambda l_j)}{\sum_j \exp(\lambda l_j)} = \theta,$$

corresponding to exponential tilting of the linear approximation to t to be centred at $\theta = 2$ and 1.2. In the call to `boot`, `R` is a vector, and `weights` a matrix with `length(R)` rows and `nrow(data)` columns, corresponding to the `length(R)` distributions from which resampling takes place.

The importance sampling weights, moments, and selected quantiles of the resamples in `city.tilt$[,1]` are calculated by

```
imp.weights(city.tilt)
imp.moments(city.tilt)
imp.quantile(city.tilt)
```

Each of these returns raw, ratio and regression estimates of the corresponding quantities. Some other uses of important resampling are exemplified by

```
imp.prob(city.tilt, t0=1.2, def=F)
z <- (city.tilt$t[,1]-city.tilt$t0[1])/sqrt(city.tilt$t[,2])
imp.quantile(boot.out=city.tilt, t=z)
```

The call to `imp.prob` calculates the importance sampling estimate of the probability that $t^* \leq 1.2$, without using defensive mixture distributions (by default `def=T`, i.e. defensive mixture distributions are used to obtain the weights and estimates). The last two lines show how importance sampling is used to estimate quantiles of the studentized bootstrap statistic.

For more details and further arguments to the functions, see their help files.

Function `tilt.boot`

The description above relies on exponential tilting to obtain the resampling probabilities, and requires knowing where to tilt to. If this is difficult, `tilt.boot` can be used to avoid this, by performing an initial bootstrap with equal resampling probabilities, then using frequency smoothing to estimate appropriate tilted probabilities. For example,

```
city.tilt <- tilt.boot(city, city.fun, R=c(500,250,249))
```

performs 500 ordinary bootstraps, uses the results to estimate probability distributions tilted to the 0.025 and 0.975 points of the simulations, and then performs 250 bootstraps tilted to the 0.025 quantile, and 249 tilted to the 0.975 quantile, before assigning the result to a bootstrap object. More complex uses of `tilt.boot` are possible; see its help file.

Importance re-weighting

These functions allow for importance re-weighting as well as importance sampling. For example, suppose that we require to re-weight the simulated values so that they appear to have been simulated from a distribution with expected ratio close to 1.4. We then use the `q=` option to the importance sampling functions as follows:

```
q <- smooth.f(theta=1.4, boot.out=city.tilt)
city.w(city, q)
imp.moments(city.tilt, q=q)
imp.quantile(city.tilt, q=q)
```

where the first line calculates the smoothed distribution, the second obtains the corresponding ratio, and the third and fourth obtain the moment and quantile estimates corresponding to simulation from the distribution `q`.

11.9.4 Saddlepoint methods

The function used for single saddlepoint approximation is `saddle`. Its simplest use is to obtain the PDF and CDF approximations for a linear statistic, such as the linear approximation $t + n^{-1}\sum f_j^* l_j$ to a general bootstrap statistic t^*. The same results are obtained by using the approximation $n^{-1}\sum f_j^* l_j$ to $t^* - t$, and this is what `saddle` does. To obtain the approximations at $t^* = 2$ for the `city` data, we set

```
saddle(A=city.L/nrow(city), u=2-city.w(city))
```

which returns the PDF and CDF approximations, and the value of $\hat{\zeta}$.

The function `saddle.distn` returns the saddlepoint estimate of an entire distribution, using the terms $n^{-1}l_j$ in the random sum and an initial idea of the centre and scale for the distribution of $T^* - t$:

```
city.t0 <- c(0, sqrt(var.linear(city.L)))
city.sad <- saddle.distn(A=city.L/nrow(city), t0=city.t0)
city.sad
```

The Lugannani–Rice formula can be applied by setting `LR=T` in the calls to `saddle` and `saddle.dist`; by default `LR=F`.

For more sophisticated applications, the arguments `A` and `u` to `saddle.distn` can be replaced by functions. For example, the bootstrapped ratio can be defined through the estimating equation

$$\sum_j f_j^*(x_j - tu_j) = 0, \tag{11.1}$$

where the f_j^* have a joint multinomial distribution with equal probabilities and denominator $n = 10$, the number of rows of `city`, as outlined in Example 9.16. Accordingly we set

```
city.t0 <- c(city.w(city), sqrt(var.linear(city.L)))
Afn <- function(t, data) data$x-t*data$u
ufn <- function(t, data) 0
saddle(A=Afn(2, city), u=0)
city.sad <- saddle.distn(A=Afn, u=ufn, t0=city.t0, data=city)
```

The penultimate line here gives the exact version of the call to `saddle` that started this section, while the last line calculates the saddlepoint approximation to the exact distribution of T^*. For `saddle.distn` the quantiles of the distribution of T^* are estimated by obtaining the CDF approximation at a number of values of t, and then interpolating the CDF using a spline smoother. The range of values of t used is determined by the contents of `t0`, whose first value contains the original value of the statistic, and whose second value contains a measure of the spread of the distribution of T^*, such as its standard error.

Another use of `saddle` and `saddle.distn` is to give them directly the adjusted cumulant generating function $K(\xi) - t\xi$, and the second derivative $K''(\xi)$. For example, the `city` data above can be tackled as follows:

```
K.adj <- function(xi)
{ L <- city$x-city.t*city$u
  nrow(city)*log(sum(exp(xi*L))/nrow(city))-city.t*xi }
K2 <- function(xi)
{ L <- city$x-city.t*city$u
  p <- exp(L*xi)
  nrow(city)*(sum(L^2*p)/sum(p) - (sum(L*p)/sum(p))^2) }
city.t <- 2
saddle(K.adj=K.adj, K2=K2)
```

This is most useful when $K(\cdot)$ is not of the standard form that follows from a multinomial distribution.

Conditional approximations

Conditional saddlepoint approximation is applied by giving `Afn` and `ufn` more columns, and setting the `wdist` and `type` arguments to `saddle` appropriately. For example, suppose that we want to find the distribution of T^*, defined as the root of (11.1), but resampling 25 rather than 49 cases of `bigcity`. Then we set

```
bigcity.L <- (bigcity$x-city.w(bigcity)*bigcity$u)/
              mean(bigcity$u)
bigcity.t0 <- c(city.w(bigcity), sqrt(var.linear(bigcity.L)))
Afn <- function(t, data) cbind(data$x-t*data$u, 1)
ufn <- function(t, data) c(0,25)
saddle(A=Afn(1.4, bigcity), u=ufn(1.4, bigcity), wdist="p",
       type="cond")
city.sad <- saddle.distn(A=Afn, u=ufn, wdist="p", type="cond",
            data=bigcity, t0=bigcity.t0)
```

Here the `wdist` argument gives the distribution of the random variables W_j, which is Poisson in this case, and the `type` argument specifies that a conditional approximation is required. For resampling without replacement, see the help file. A further argument `mu` allows these variables to have differing means, in which case the conditional saddlepoint will correspond to sampling from multinomial or hypergeometric distributions with unequal probabilities.

11.10 Semiparametric Likelihoods

Basic functions only are provided for semiparametric likelihood inference.

To calculate and plot the log profile likelihood for the mean of a gamma model for the larger air conditioning data (Example 10.1):

```
gam.L <- function(y, tmin=min(y)+0.1, tmax=max(y)-0.1, n.t)
{ gam.loglik <- function(l.nu, mu, y)
  { nu <- exp(l.nu)
    -sum(log(dgamma(nu*y/mu, nu)*nu/mu)) }
  out <- matrix(NA, n.t+1, 3)
  for (it in 0:n.t)
  { t <- tmin + (tmax-tmin)*it/n.t
    fit <- nlminb(0, gam.loglik, mu=t, y=y)
    out[1+it,] <- c(t, exp(fit$parameters), -fit$objective) }
  out }
```

```
air.gam <- gam.L(aircondit7$hours, 40, 120, 100)
air.gam[,3] <- air.gam[,3] - max(air.gam[,3])
plot(air.gam[,1],air.gam[,3],type="l",xlab="theta",
     ylab="Log likelihood",xlim=c(40,120))
abline(h=-0.5*qchisq(0.95,1),lty=2)
```

Empirical and empirical exponential family likelihoods are obtained using the functions `EL.profile` and `EEF.profile`. They are included in the library for demonstration purposes only, and are not intended for serious use, nor are they currently supported as part of the library. These functions give log likelihoods for the mean of their first argument, calculated at `n.t` values of θ from `tmin` and `tmax`. The output of `EL.profile` is a `n.t`$\times$3 matrix whose first column is the values of θ, whose next column is the log profile likelihood, and whose final column is the values of the Lagrange multiplier. The output of `EEF.profile` is a `n.t`$\times$4 matrix whose first column is the values of θ, whose next two columns are versions of the log profile likelihood (see Problem 10.4), and whose final column is the values of the Lagrange multiplier. For example:

```
air.EL <- EL.profile(aircondit7$hours,tmin=40,tmax=120,n.t=100)
lines(air.EL[,1],air.EL[,2],lty=2)
air.EEF <- EEF.profile(aircondit7$hours,tmin=40,tmax=120,
                       n.t=100)
lines(air.EEF[,1],air.EEF[,3],lty=3)
```

Note how close the two semiparametric log likelihoods are, compared to the parametric one. The practicals at the end of Chapter 10 give more examples of their use (and abuse).

More general (and more robust!) code to calculate empirical likelihoods is provided by Professor A. B. Owen at Stanford University; see World Wide Web reference `http://playfair.stanford.edu/reports/owen/el.S`.