Resampling

Jackknife and Bootstrap

Motivation

Suppose we use a sample $x_1, \ldots, x_n$ to obtain an estimate $\hat{\theta}$ of a parameter of interest θ.

What can we say about the precision/variability of $\hat{\theta}$ (over all possible samples)?

If using standard deviation to measure precision/variability, this asks about the **standard error**

$$\mathrm{se}(\hat{\theta}) = \mathrm{sd}(\hat{\theta}(X_1, \ldots, X_n)).$$

In some cases (e.g., for the mean) we can compute the sd, and use suitable (approximate) **plug-in** estimates.

Old-style: heavy use of complicated formulas obtained using Taylor expansions.

New-style: use resampling methods!

The jackknife estimate of standard error

Suppose we have observations $x_1, \ldots, x_n$ from i.i.d. $X_1, \ldots, X_n$ with unknown distribution (function) F. and we compute a real-valued statistic

$$\hat{\theta} = s(x) = s(x_1, \ldots, x_n)$$

using a suitable "algorithm" s.

Write $x_{(i)} = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ for the sample with x_i removed, and

$$\hat{\theta}_{(i)} = s(x_{(i)})$$

for the corresponding estimate.

(Note that this implies that s works for samples of sizes n and $n-1$.)

The jackknife estimate of standard error

The jackknife estimate of standard error for $\hat{\theta}$ is

$$\widehat{\mathrm{se}}_{\mathrm{jack}} = \left(\frac{n-1}{n} \sum_{i=1}^{n} \left(\hat{\theta}_{(i)} - \hat{\theta}_{(\cdot)} \right)^2 \right)^{1/2}, \qquad \hat{\theta}_{(\cdot)} = \frac{1}{n} \sum_{i=1}^{n} \hat{\theta}_{(i)}.$$

If $\hat{\theta}$ is the mean $\bar{x}$, clearly

$$\hat{\theta}_{(i)} = \frac{n\bar{x} - x_i}{n-1}, \qquad \hat{\theta}_{(\cdot)} = \bar{x}, \qquad \hat{\theta}_{(i)} - \hat{\theta}_{(\cdot)} = \frac{\bar{x} - x_i}{n-1}$$

so that

$$\widehat{\mathrm{se}}_{\mathrm{jack}} = \left(\frac{1}{n(n-1)} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{1/2} = \left(\frac{s_n^2(x_1, \ldots, x_n)}{n} \right)^{1/2}$$

is the classical (plug-in) estimate based on $\mathrm{sd}(\bar{X}) = \mathrm{var}(X)/n$.

The jackknife estimate of standard error

Some features of the jackknife formula:

- It is non-parametric: no special form about the underlying distribution F is assumed.
- It is completely automatic (good for computers): we can easily write code which inputs x and s and returns $\widehat{\mathrm{se}}_{\mathrm{jack}}$.
- Algorithm uses samples of size $n-1$. Nice if s is "smooth" (across sample sizes), but e.g. not the case for the median.
- Jackknife estimate of standard error is upwardly biased as an estimate of the "true" standard error.

The jackknife estimate of standard error

For the German data, let us investigate the correlation between Amount and Duration:

```
R> load("german.rda")
R> with(german, cor(Amount, Duration))
```

```
[1] 0.6249842
```

Computing the jackknife estimates by hand:

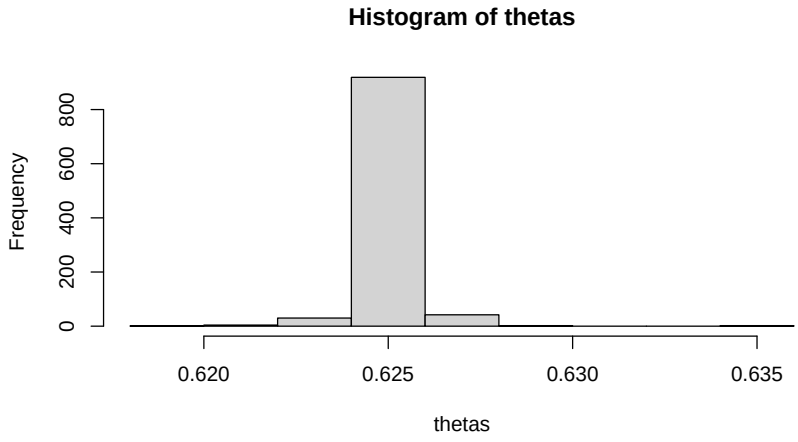
```
R> n <- nrow(german)
R> thetas <- with(german,
+                 vapply(seq(1, n),
+                        function(i) cor(Amount[-i], Duration[-i]),
+                        0))
R> se_jack <- sqrt( (n - 1) * mean( (thetas - mean(thetas)) ^ 2 ) )
R> se_jack
```

```
[1] 0.02597192
```

The jackknife estimate of standard error

Inspect distribution of $\hat{\theta}_{(i)}$ graphically:

```
R> hist(thetas)
```



The non-parametric bootstrap

The frequentist standard error of an estimate $\hat{\theta} = s(x)$ would be obtained by repeatedly sampling new x from F.

This is impossible, because F is unknown.

But perhaps we can sample from suitable approximations to F?

The non-parametric bootstrap uses the "obvious approximation": the empirical distribution function $\hat{F}$ which puts mass $1/n$ on each of the observed $x_1, \ldots, x_n$.

I.e., bootstrap samples x^* are obtained by sampling from x **with replacement**.

Each bootstrap sample provides a **bootstrap replication** of the statistic of interest:

$$\hat{\theta}^* = s(x^*).$$

The non-parametric bootstrap

We draw a "sufficiently large" number B of bootstrap samples $x^*_{(1)}, \ldots, x^*_{(B)}$, and compute the corresponding bootstrap replications:

$$\hat{\theta}^*_{(b)} = s(x^*_{(b)}), \qquad b = 1, \ldots, B.$$

The bootstrap estimate of standard error for $\hat{\theta}$ is the empirical standard deviation of the $\hat{\theta}^*_{(b)}$:

$$\widehat{\mathrm{se}}_{\mathrm{boot}} = \left(\frac{1}{B-1} \sum_{b=1}^{B} \left(\hat{\theta}^*_{(b)} - \hat{\theta}^*_{(\cdot)} \right)^2 \right)^{1/2}, \qquad \hat{\theta}^*_{(\cdot)} = \frac{1}{B} \sum_{b=1}^{B} \hat{\theta}^*_{(b)}.$$

The non-parametric bootstrap

Some more motivation: the original process is

$$F \xrightarrow{\text{i.i.d.}} x \xrightarrow{s} \hat{\theta}.$$

We don't know F, but we can estimate using the empirical distribution function $\hat{F}$. This actually maximizes the probability of observing the given sample x over all possible F: it is the **non-parametric MLE** of F.

Bootstrap replications are obtained by mimicking the original process:

$$\hat{F} \xrightarrow{\text{i.i.d.}} x^* \xrightarrow{s} \hat{\theta}^*.$$

We know that as $n \to \infty$, $\hat{F} \to F$, so for n large enough, the above should be a good idea, and hence $\widehat{\text{se}}_{\text{boot}}$ should be a good approximation of the "true" standard error.

The non-parametric bootstrap

If we could draw samples $x_{(1)}, \ldots, x_{(B)}$ from (the unknown) F, we could approximate the standard error of $\hat{\theta}$ by the empirical standard deviation of the corresponding $\hat{\theta}_{(b)} = s(x_{(b)})$ values:

$$\widehat{\mathrm{se}}(F) = \left(\frac{1}{B-1} \sum_{b=1}^{B} \left(\hat{\theta}_{(b)} - \hat{\theta}_{(\cdot)} \right) \right)^{1/2}, \qquad \hat{\theta}_{(\cdot)} = \frac{1}{B} \sum_{b=1}^{B} \hat{\theta}_{(b)},$$

where the "F" is used to denote sampling from F.

The bootstrap estimate of standard error for $\hat{\theta}$ is the plug-in estimate

$$\widehat{\mathrm{se}}_{\mathrm{boot}} = \widehat{\mathrm{se}}(\hat{F}).$$

The non-parametric bootstrap

Some important points about the bootstrap approach:

- We have described the **one-sample non-parametric bootstrap**.
- It is completely automatic (good for computers): we can easily write code which inputs x and s and returns $\widehat{\mathrm{se}}_{\mathrm{boot}}$.
- Bootstrapping "shakes" the data much more than the jackknife, and hence is more dependable on unsmooth statistics (as it is not based on local derivatives).
- $B = 200$ is usually sufficient for $\widehat{\mathrm{se}}_{\mathrm{boot}}$. For confidence intervals (discussed later on), larger values (e.g., $B = 1000$ or $B = 2000$) are needed.
- There is nothing special about standard errors: could use the same approach for e.g. the expected absolute error $\mathbb{E}|\hat{\theta} - \theta|$.

The non-parametric bootstrap

Compute the bootstrap standard error of the correlation between Amount and Duration:

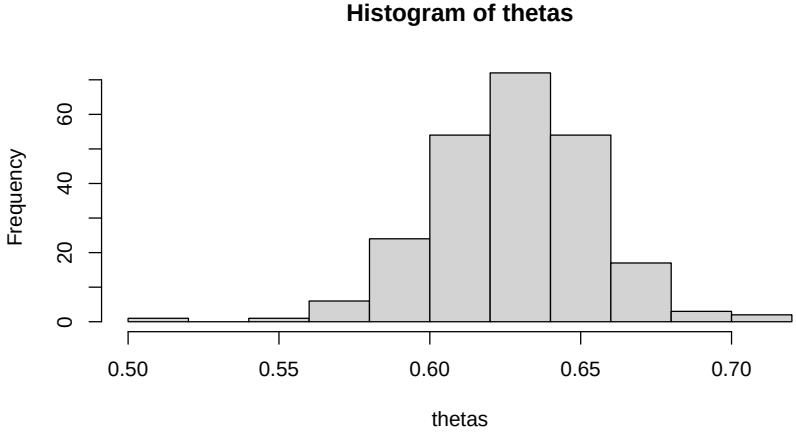
```
R> B <- 234
R> thetas <-
+     with(german,
+          replicate(B, {
+              i <- sample(n, n, replace = TRUE) # Sample with replacement
+              cor(Amount[i], Duration[i])
+          }))
R> se_boot <- sd(thetas)
R> se_boot
```

```
[1] 0.02661181
```

The non-parametric bootstrap

Inspect distribution of $\hat{\theta}^*_{(b)}$ graphically:

```
R> hist(thetas)
```



boot::boot

Recommended package boot implements the "completely automatic" procedure described above and much more.

```
R> require("boot")
```

The basic function is boot() with arguments:

```
R> args(boot)
```

```
function (data, statistic, R, sim = "ordinary", stype = c("i",
    "f", "w"), strata = rep(1, n), L = NULL, m = 0, weights = NULL,
    ran.gen = function(d, p) d, mle = NULL, simple = FALSE, ...,
    parallel = c("no", "multicore", "snow"), ncpus = getOption("boot.ncpus",
        1L), cl = NULL)
NULL
```

Ouch! That looks rather complicated ...

boot::boot

For getting started, not so bad:

- `data` is the data (x in our case)
- `statistic` corresponds to our function s
- `R` is our B.

From the docs: (unless parametric), `statistic` must take at least two arguments, the original data and a vector of indices (default), frequencies or weights which define the bootstrap sample.

boot::boot

In our correlation example, we can either arrange subscripting the variables of interest when passing the data, or when computing the statistic. Doing the latter:

```
R> statistic <- function(d, i)
+     with(d, cor(Amount[i], Duration[i]))
R> bon <- boot(german, statistic, B)
R> bon

ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = german, statistic = statistic, R = B)


Bootstrap Statistics :
     original       bias    std. error
t1* 0.6249842 -0.001305521  0.02632988
```

boot::boot

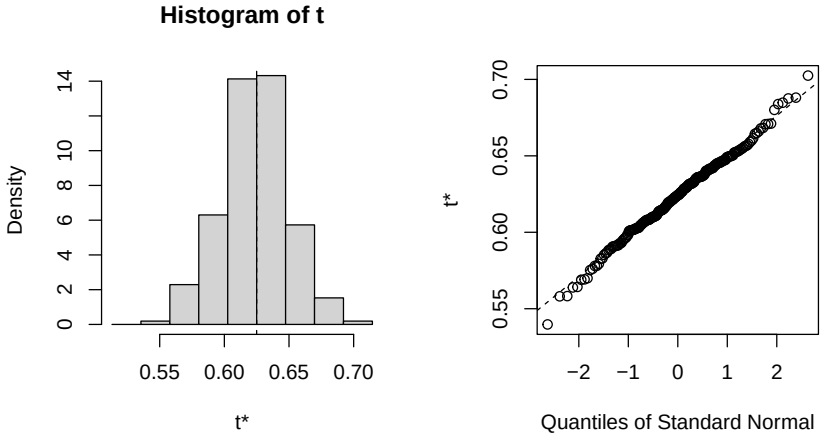
Not so bad!

In fact, what we get is an object of class `boot` which has some useful methods, and can be re-used in subsequent computations.

```
R> class(bon)

[1] "boot"

R> names(bon)

 [1] "t0"        "t"         "R"         "data"      "seed"      "statistic"
 [7] "sim"       "call"      "stype"     "strata"    "weights"

R> dim(bon$t)

[1] 234   1
```

boot::boot

```
R> plot(bon)
```



Resampling plans

The jackknife and bootstrap formulas look somewhat similar, so maybe there is a unifying framework?

Think of s not as acting directly on the sample values $x_1, \ldots, x_n$, but on probability distributions $p = (p_1, \ldots, p_n)$. I.e., suppose s can work with **weights** for the individual observations.

In the basic case of the mean,

$$s(x, p) = \sum_{i=1}^{n} p_i x_i$$

is the weighted mean.

Of course, we must have that when using equals weights, i.e., $p_0 = (1, \ldots, 1)/n$, we get the original statistic: $\hat{\theta} = s(x, p_0)$.

Resampling plans

For the jackknife, we replaced p_0 by

$$p_{(i)} = (1, \ldots, 1, 0, 1, \ldots, 1)/(n-1),$$

with the 0 in the ith place.

For the bootstrap, we replaced p_0 by resampling vectors $p^*_{(b)}$ of the form

$$p^* = (N_1, \ldots, N_n)/n$$

with the N_i non-negative integers summing to n. When sampling with replacement, $(N_1, \ldots, N_n)$ has a multinomial distribution with parameters n and p_0.

Clearly, we can use other "resampling plans" as well!

Resampling plans

E.g., one can try the effect of random reweighting (i.e., drawing p according to the uniform distribution on the probability simplex).

One such approach is the **Bayesian bootstrap**.

One can show: if $G_1, \ldots, G_n$ are i.i.d. standard exponential, then

$$p^* = (G_1, \ldots, G_n) \Big/ \sum_{i=1}^{n} G_i$$

has a uniform distribution on the probability simplex.

This gives p^* with slightly smaller covariance matrix than when using the non-parametric bootstrap.

Resampling plans

For the jackknife vectors,

$$\|p_{(i)} - p_0\| = \frac{1}{\sqrt{n(n-1)}}.$$

For the non-parametric bootstrap, each N_i is Binomial with parameters size n and probability $1/n$, so that

$$\mathbb{E}(N_i) = 1, \qquad \mathrm{var}(N_i) = n \cdot \frac{1}{n}\left(1 - \frac{1}{n}\right) = \frac{n-1}{n}$$

Thus,

$$\mathbb{E}\|p^* - p_0\|^2 = n\mathrm{var}(N_i/n) = \frac{n-1}{n^2}$$

so that $(\mathbb{E}\|p^* - p_0\|^2)^{1/2}$ is $\sqrt{n}$ times the distance to the jackknife vectors: thus, the bootstrap shakes much ore wildly.

The parametric bootstrap

For the bootstrap recipe

$$\hat{F} \xrightarrow{\text{i.i.d.}} x^* \xrightarrow{s} \hat{\theta}^*.$$

we do not have to insist that $\hat{F}$ is the non-parametric MLE of F. Suppose we have a parametric model $\mathcal{F}$ with suitable densities:

$$\mathcal{F} = \{F_\pi : \pi \in \Pi\}$$

(notation is a bit awkward: but π is not necessarily the same as the parameter of interest already denoted by θ).

With $\hat{\pi}$ the MLE of π, we can then use

$$f_{\hat{\pi}} \xrightarrow{\text{i.i.d.}} x^* \xrightarrow{s} \hat{\theta}^*.$$

to generate the bootstrap samples.

This is the **parametric bootstrap**.

The parametric bootstrap

Let us try to use the parametric bootstrap for the standard error of the correlation between Amount and Duration.

We need a "reasonable" model for generating the (Amount, Duration) pairs.

A bivariate normal may come to mind: but we already established that both variables are (significantly) non-normal! Taking logs makes things somewhat better (worse for Duration as this is more discrete).

Hence, let us use a bivariate log-normal for the purpose of illustration (only!).

The parametric bootstrap

First, compute suitable parameter estimates:

```
R> x <- with(german,
+            cbind(lA = log(Amount), lD = log(Duration)))
R> (mu <- colMeans(x))

      lA       lD
7.788691 2.877018

R> (Sigma <- var(x))

          lA        lD
lA 0.6029120 0.2813133
lD 0.2813133 0.3390034
```

(Not quite the MLE for Σ.)

The parametric bootstrap

Now compute the bootstrap standard error:

```
R> thetas <-
+     with(german,
+          replicate(B, {
+              ## Draw from the log-normal distribution with mu and Sigma:
+              x <- exp(mvtnorm::rmvnorm(n, mu, Sigma))
+              cor(x[, 1], x[, 2])
+          }))
R> se_boot <- sd(thetas)
R> se_boot
```

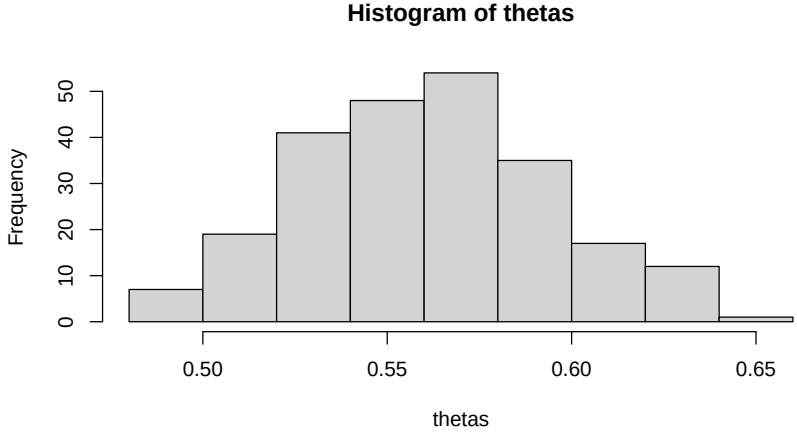
```
[1] 0.03356057
```

(This is larger than before because the model does not really fit the data well.)

The parametric bootstrap

Inspect distribution of $\hat{\theta}^*_{(b)}$ graphically:

```
R> hist(thetas)
```



boot::boot

Alternatively, let us use the `boot()` function again.

This time we need the parameteric bootstrap, i.e., `sim = "parametric"`, and we need to implement the `ran.gen` function to do the parametric sampling.

Perhaps the simplest is to arrange for `statistic` to compute the correlation of the columns of a 2-column matrix, and for `ran.gen` to sample such matrices from the bivariate log-normal distribution.

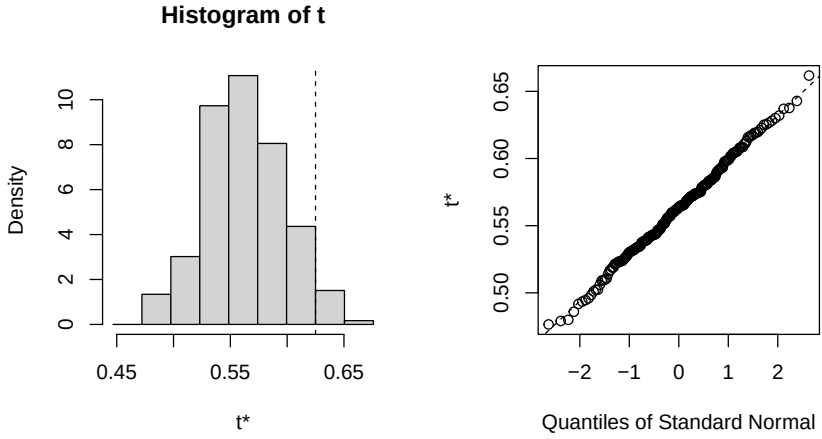
```
R> gad_d <- with(german, cbind(Amount, Duration))
R> gad_s <- function(d)
+     cor(d[, 1], d[, 2])
R> gad_r <- function(d, mle)
+     exp(mvtnorm::rmvnorm(nrow(d), mle$mu, mle$Sigma))
```

boot::boot

```
R> bop <- boot(gad_d, gad_s, B, sim = "parametric", ran.gen = gad_r,
+              mle = list(mu = mu, Sigma = Sigma))
R> bop

PARAMETRIC BOOTSTRAP


Call:
boot(data = gad_d, statistic = gad_s, R = B, sim = "parametric",
    ran.gen = gad_r, mle = list(mu = mu, Sigma = Sigma))


Bootstrap Statistics :
     original      bias    std. error
t1* 0.6249842 -0.06205363  0.03457102
```

boot::boot

```
R> plot(bop)
```



Bootstrap Confidence Intervals

Motivation

In general, standard errors provide some insight into the precision/variability of estimates, but are comprehensive really only with good normal approximations in place.

I.e., when

$$\frac{\hat{\theta} - \theta}{\widehat{\mathrm{se}}} \approx N(0, 1)$$

so that e.g.

$$\hat{\theta} \pm z_{\alpha/2}\widehat{\mathrm{se}}$$

gives an approximate $1 - \alpha$ confidence interval.

However, sampling and bootstrap distributions are often "rather non-normal" (skewed, ...): how can we obtain better confidence intervals in these cases?

Package `boot` implements five different types of (equi-tailed two-sided nonparametric) confidence intervals.

Bootstrap confidence intervals: First order normal approximation

This simply uses

$$(\hat{\theta} - z_{1-\alpha/2}\widehat{\mathrm{se}}_{\mathrm{boot}}, \hat{\theta} - z_{\alpha/2}\widehat{\mathrm{se}}_{\mathrm{boot}}),$$

with $z_\alpha = \Phi^{-1}(\alpha)$ the α quantile of the standard normal distribution.

As discussed, only works well if $\hat{\theta}$ is approximately normal.

Bootstrap confidence intervals: Percentile method

This uses the empirical quantiles of the bootstrap distribution.

Write

$$\hat{G}(t) = \frac{1}{B}\#\left\{1 \le b \le B : \hat{\theta}^*_{(b)} \le t\right\}.$$

Then $\hat{G}$ gives the empirical distribution (function) for the bootstrap replication sample $\hat{\theta}^*_{(1)}, \ldots, \hat{\theta}^*_{(B)}$.

Write $\hat{G}^{-1}$ for the corresponding quantile function.

The $1 - \alpha$ central percentile interval is

$$(\hat{G}^{-1}(\alpha/2), \hat{G}^{-1}(1 - \alpha/2)).$$

Bootstrap confidence intervals: Basic method

This also uses the empirical quantiles of the bootstrap distribution, but together with the idea that the distribution of $\hat{\theta} - \theta$ can be approximated by that of $\hat{\theta}^* - \hat{\theta}$.

One can then use

$$\mathbb{P}(L \leq \hat{\theta} - \theta \leq U) \approx \mathbb{P}(L \leq \hat{\theta}^* - \hat{\theta} \leq U) = 1 - \alpha$$

to obtain an approximate $1 - \alpha$ confidence interval as $(\hat{\theta} - U, \hat{\theta} - L)$, with $L + \hat{\theta}$ and $U + \hat{\theta}$, respectively, the empirical $\alpha/2$ and $1 - \alpha/2$ quantiles of the bootstrap distribution. I.e.,

$$L = \hat{G}^{-1}(\alpha/2) - \hat{\theta}, \quad U = \hat{G}^{-1}(1 - \alpha/2) - \hat{\theta}$$

so that the basic (aka reverse percentile) intervals are obtained as

$$(2\hat{\theta} - \hat{G}^{-1}(1 - \alpha/2), 2\hat{\theta} - \hat{G}^{-1}(\alpha/2)).$$

Bootstrap confidence intervals: Advanced methods

The studentized confidence intervals are based on studentized values of the bootstrap distribution: studentizing does approximate pivoting and hence helps approximations. Complicated.

Bias-corrected (BC) confidence intervals correct for bias. In the simplest approach, one uses the bias-correction value $z_0 = \Phi^{-1}(\hat{G}(\hat{\theta}))$, and BC level α confidence interval endpoints given by

$$\hat{\theta}_{\mathrm{BC}}[\alpha] = \hat{G}^{-1}(\Phi(2z_0 + \Phi^{-1}(\alpha))).$$

Complicated, and not provided by package `boot`.

The bias-corrected and accelerated (BCa) method uses endpoints of the form

$$\hat{\theta}_{\mathrm{BCa}}[\alpha] = \hat{G}^{-1}\left(\Phi\left(z_0 + \frac{z_0 + \Phi^{-1}(\alpha)}{1 - a(z_0 + \Phi^{-1}(\alpha))}\right)\right).$$

for suitable a. Complicated, but works rather well (often with second-order accuracy).

Bootstrap confidence intervals

Using package `boot`, one can obtain the confidence intervals for the implemented types/methods using function `boot.ci()`.

This takes as arguments:

- An object of class `boot` with the results of bootstrap calculations using function `boot()`.
- A scalar or vector containing the confidence level(s) of the required interval(s) (named `conf`, default: 0.95).
- A vector of character strings giving the types of intervals desired (named `type`, default: `"all"`).

And more ... (complicated).

Bootstrap confidence intervals

Compute bootstrap confidence intervals for the correlation between Amount and Duration, first using the non-parametric and then a parametric bootstrap (remember that the latter is for illustration only).

For computing confidence intervals, we should use $B \geq 1000$.

```
R> B <- 1234
```

Bootstrap confidence intervals

```
R> statistic <- function(d, i)
+     with(d, cor(Amount[i], Duration[i]))
R> bon <- boot(german, statistic, B)
R> bon

ORDINARY NONPARAMETRIC BOOTSTRAP


Call:
boot(data = german, statistic = statistic, R = B)


Bootstrap Statistics :
     original        bias    std. error
t1* 0.6249842 -0.0005716118  0.02471814
```

Bootstrap confidence intervals

```
R> boot.ci(bon, conf = c(0.9, 0.95), type = c("norm", "basic", "perc"))

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 1234 bootstrap replicates

CALL :
boot.ci(boot.out = bon, conf = c(0.9, 0.95), type = c("norm",
    "basic", "perc"))

Intervals :
Level      Normal              Basic              Percentile
90%   ( 0.5849,  0.6662 )   ( 0.5878,  0.6684 )   ( 0.5816,  0.6621 )
95%   ( 0.5771,  0.6740 )   ( 0.5817,  0.6761 )   ( 0.5739,  0.6683 )
Calculations and Intervals on Original Scale
```

Bootstrap confidence intervals

```
R> bop <- boot(gad_d, gad_s, B, sim = "parametric", ran.gen = gad_r,
+              mle = list(mu = mu, Sigma = Sigma))
R> bop

PARAMETRIC BOOTSTRAP


Call:
boot(data = gad_d, statistic = gad_s, R = B, sim = "parametric",
    ran.gen = gad_r, mle = list(mu = mu, Sigma = Sigma))


Bootstrap Statistics :
     original      bias    std. error
t1* 0.6249842 -0.06245128  0.03423014
```

Bootstrap confidence intervals

```
R> boot.ci(bop, conf = c(0.9, 0.95), type = c("norm", "basic", "perc"))

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 1234 bootstrap replicates

CALL :
boot.ci(boot.out = bop, conf = c(0.9, 0.95), type = c("norm",
    "basic", "perc"))

Intervals :
Level      Normal              Basic              Percentile
90%   ( 0.6311,  0.7437 )   ( 0.6313,  0.7434 )   ( 0.5066,  0.6187 )
95%   ( 0.6203,  0.7545 )   ( 0.6165,  0.7538 )   ( 0.4962,  0.6335 )
Calculations and Intervals on Original Scale
```

Bootstrap Hypothesis Tests

Two-sample t test

One can also use the bootstrap for hypothesis testing (with applicability more general than for permutation tests).

Consider again the problem of testing the equality of the means in two different populations (without assuming normality or equal variances).

The "obvious" test statistic was

$$T = \frac{\bar{X}_m - \bar{Y}_n}{\sqrt{S_X^2/m + S_Y^2/n}}.$$

We can compute a (two-sample) bootstrap distribution for this by independently drawing $x^*_{(b)}$ with replacement from x and drawing $y^*_{(b)}$ with replacement from y, and computing the test statistic for these bootstrap samples, **after** centering to common means.

Two-sample t test

```
R> x <- with(german, Amount[Purpose == "car/new"])
R> y <- with(german, Amount[Purpose == "business"])
R> m <- length(x); n <- length(y)
R> tstat <- function(xs, ys)
+     (mean(xs) - mean(ys)) / sqrt( var(xs) / m + var(ys) / n )
R> tvals <- replicate(1234, {
+     mu <- mean(c(x, y))
+     xc <- x - mean(x) + mu
+     yc <- y - mean(y) + mu
+     xs <- sample(xc, m, replace = TRUE)
+     ys <- sample(yc, n, replace = TRUE)
+     tstat(xs, ys)
+ })
R> mean(tvals < tstat(x, y))
```

```
[1] 0.0008103728
```

Estimating Prediction Error

Motivation

Thus far, we considered statistical modeling in the context of making inference about the underlying data generating processes.

We can also use these models (and other "supervised learning methods") for making **predictions**.

E.g., for linear models we predict the response y from covariates x using the linear predictor: $\hat{y} = \beta' x$.

For logistic regression models we predict the distribution of the binary response y from covariates x as $\mathbb{P}(y = 1|x) = \text{plogis}(\beta' x)$.

How can we estimate the quality of these predictions?

Prediction rules

Prediction usually starts with a **training set**

$$D = \{(x_i, y_i), i = 1, \ldots, n\}$$

of pairs of predictors x_i and corresponding (target) responses y_i.

From this, we construct a **prediction rule** r_D as a map from the space $\mathcal{X}$ of predictors to the space $\mathcal{Y}$ of responses. For every $x \in \mathcal{X}$, we predict y as

$$\hat{y} = r_D(x).$$

The rule could be derived from a statistical model, but could also come from applying learning algorithms without a solid foundation from statistical inference.

Prediction rules

To evaluate the performance of a prediction rule r, we need to quantify how much for a pair (x, y) the prediction $\hat{y} = r(x)$ differs from the target response y, i.e., need a suitable **discrepancy measure** $d(y, \hat{y})$ (traditional terminology: loss function L).

Two most common choices: **squared error**

$$d(y, \hat{y}) = (y - \hat{y})^2$$

for (single) numeric y, and **classification error**

$$d(y, \hat{y}) = I_{y \neq \hat{y}}$$

for (single) binary y.

Prediction error

If r_D is a prediction rule obtained from a training data set D of pairs (x_i, y_i), its **true error rate** Err_D is the expected discrepancy $\mathbb{E}(d(y, r_D(x)))$, with (x, y) drawn from a "suitable" probability distribution F on the predictor-response pairs (x, y), and independently from how r was obtained (in particular from D):

$$\mathrm{Err}_D = E_{(x,y)\sim F}(d(y, r_D(x))).$$

Some references refer to this as the **conditional** error.

Integrating out over all possible training sets then gives the **expected** error

$$\mathrm{Err} = E_D(\mathrm{Err}_D) = E_D E_{(x,y)\sim F}(d(y, r_D(x))).$$

Typically, one assumes that D is obtained by random sampling from the same F.

Note: the statistical models considered thus far are for the conditional distributions of y given x. Prediction errors look at the joint distribution of y and x!

Prediction error

A naive idea is to estimate Err using the **in-sample** (training set) apparent error

$$\text{err} = \frac{1}{n}\sum_{i=1}^{n} d(y_i, \hat{y}_i),$$

the average discrepancy over D between the y_i and the $\hat{y}_i = r_D(x_i)$.

This usually underestimates Err because r_D was obtained to fit the training responses y_i.

(Cf. the need to use $\sum_i (y_i - \hat{y}_i)^2/(n-p)$ in the linear modeling case to obtain an unbiased estimate of the noise variance σ^2.)

Prediction error

The "obvious" solution is to estimate the error using an independent **validation set** (or **test set**) $D_{\text{test}} = \{(x_{0j}, x_{0j}), j = 1, \ldots, n_{\text{test}}\}$:

$$\widehat{\text{Err}}_{\text{test}} = \frac{1}{n_{\text{test}}} \sum_{j=1}^{n_{\text{test}}} d(y_{0j}, \hat{y}_{0j}), \qquad \hat{y}_{0j} = r_D(x_{0j}).$$

(The x_{0j} notation is trying to suggest that this is the jth "new" x.)

This would provide an unbiased estimate of Err.

Sometimes this makes a lot of sense for "stability reasons": e.g., how does a rating model developed using older data actually work for newer data?

Typically there are no "dedicated" validation sets, and one has to split the given data into suitably "independent" training and test data sets D_{train} and D_{test}.

Obvious problem: with n_{test} small the error estimate has high variance. Otoh, the larger n_{test}, the higher the variability in obtaining the rule from the training set.

Prediction error

Let us try to evaluate the performance of predicting Amount from Duration and Job.

Helper function to compute the mean squared errors:

```
R> MSE <- function(y, yhat) mean((y - yhat)^2)
```

Apparent error using all available data:

```
R> m <- lm(Amount ~ Duration + Job, data = german)
R> with(german, MSE(Amount, predict(m)))
```

```
[1] 4428006
```

Now try several independent 50-50 training-test data splits:

Prediction error

```
R> n <- nrow(german)
R> errs_50_50 <- replicate(123, {
+     i <- sample(1 : n, n / 2)
+     m <- lm(Amount ~ Duration + Job, data = german[i, ])
+     MSE(german$Amount[-i], predict(m, german[-i, ]))
+ })
R> mean(errs_50_50)
```

```
[1] 4487242
```

```
R> sd(errs_50_50)
```

```
[1] 357094.9
```

And now redo using several independent 75-25 splits:

Prediction error

```
R> errs_75_25 <- replicate(123, {
+     i <- sample(1 : n, 0.75 * n)
+     m <- lm(Amount ~ Duration + Job, data = german[i, ])
+     MSE(german$Amount[-i], predict(m, german[-i, ]))
+ })
R> mean(errs_75_25)

[1] 4475154

R> sd(errs_75_25)

[1] 641700.7
```

Cross-validation

One extreme idea is to try all possible test sets of size one.

Define $D_{(i)}$ as the reduced data set with (x_i, y_i) omitted, and use

$$\widehat{\mathrm{err}}_{\mathrm{LOOCV}} = \frac{1}{n}\sum_{i=1}^{n} d(y_i, \hat{y}_{(i)}), \qquad \hat{y}_{(i)} = r_{D_{(i)}}(x_i).$$

This is "leave one out" cross-validation.

Cross-validation

Continuning our example with brute force:

```
R> errs_LOOCV <-
+     vapply(1 : n,
+            function(i) {
+                m <- lm(Amount ~ Duration + Job, data = german[-i, ])
+                MSE(german$Amount[i], predict(m, german[i, ]))
+            },
+            0)
R> mean(errs_LOOCV)

[1] 4500835

R> sd(errs_LOOCV)

[1] 12071545
```

Cross-validation

This nicely shows the problem with LOOCV: the test sets of size one produce extremely variable error estimates.

In addition, these are rather correlated, so that the true error rate is often overestimated.

And of course, in general computing rules for all n training data sets $D_{(i)}$ will be prohibitively expensive.

For linear models, one can actually show that

$$\widehat{\mathrm{err}}_{\mathrm{LOOCV}} = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{y_i - \hat{y}_i}{1 - h_i} \right)^2 .$$

Cross-validation

A more common tactic therefore is to leave out "several" pairs at a time: D is randomly partitioned into k groups of approximate size n/k. With $D_{(j)}$ the training set with group j omitted, one then evaluates the error of rule $r_{D_{(j)}}$ on the pairs in group j.

This is k-fold cross-validation.

For moderate k (usually, $k = 10$), this has several advantages:

- Only k rules need to be learned.
- Using about $(k-1)n/k$ pairs for training and n/k pairs for testing gives a reasonable compromise between the needs for the training and test sets.

Cross-validation

Continuing our example "by hand", we need a function to compute a random partition of $\{1, \ldots, n\}$ into k groups ("folds") of approximately equal sizes.

A simple way is to create the groups explicitly (instead of numbering them from 1 to k:

```
R> folds <- function(n, k)
+     unname(split(sample(1 : n, n), cut(1 : n, breaks = k)))
R> ## Small example:
R> folds(10, 3)

[[1]]
[1] 7 1 5 3

[[2]]
[1]  9 10  2

[[3]]
[1] 4 8 6
```

Cross-validation

Using 10-fold cross-validation for our German example:

```
R> errs_10CV <-
+     vapply(folds(n, 10),
+            function(i) {
+                m <- lm(Amount ~ Duration + Job, data = german[-i, ])
+                MSE(german$Amount[i], predict(m, german[i, ]))
+            },
+            0)
R> mean(errs_10CV)

[1] 4519866

R> sd(errs_10CV)

[1] 1163656
```

Cross-validation

Issues with cross-validation:

- For $k < n$, estimates vary according to the random partition obtained. Could try the average from several such partitions?
- What we really estimate is the (average) performance of rules trained from data sets using $k - 1$ folds, but not all data.
- Which of the k rules should we use?
- Simple random splitting only works in the i.i.d. case.

boot::cv.glm

Package boot provides function cv.glm() to calculate the estimated k-fold cross-validation prediction error for generalized linear models (and hence also for linear models using the default gaussian family).

Arguments:

```
R> args(cv.glm)
```

```
function (data, glmfit, cost = function(y, yhat) mean((y - yhat)^2),
    K = n)
NULL
```

where the default for cost is mean squared error.

boot::cv.glm

Using 10-fold cross-validation for our German example, version 2:

```
R> m <- glm(Amount ~ Duration + Job, data = german)
R> cv.glm(german, m, K = 10)$delta
```

```
[1] 4491039 4487696
```

Bootstrap methods

How can we use the bootstrap to estimate prediction error?

Simplest idea: obtain prediction rules from bootstrap samples, and evaluate on the original data as ("independent"?) test set.

I.e., with $r^*_{(b)}$ the rule obtained from sample $D^*_{(b)}$, use

$$\widehat{\mathrm{Err}}_{\mathrm{Boot}} = \frac{1}{B}\frac{1}{n}\sum_{b=1}^{B}\sum_{i=1}^{n} d(y_i, r^*_{(b)}(x_i)).$$

Does not really work, as the $D^*_{(b)}$ are obtained by resampling D, and hence not really independent.

Perhaps use only the observations in the original data **not** in the bootstrap sample?

Bootstrap methods

These observations are called the **out-of-bag** (OOB) observations.

Remember: with $N_1, \ldots, N_n$ the counts of the (mutually distinct) $x_1, \ldots, x_n$ in the boostrap samples, $(N_1, \ldots, N_n)$ is Multinomial with parameters n and $1/n, \ldots, 1/n$.

Each N_i is thus Binomial with parameters n and $p = 1/n$. Hence:

$$\mathbb{P}(N_i = 0) = (1 - 1/n)^n \approx e^{-1} = 0.3678794.$$

On average, roughly $1/3$ of the observations are not in a bootstrap sample (and hence, are OOB observations).

The **leave-one-out bootstrap** estimate of prediction error is defined as

$$\widehat{\mathrm{Err}}_{\mathrm{Boot}(1)} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{|C^{-i}|} \sum_{b \in C^{-i}} d(y_i, r^*_{(b)}(x_i)),$$

where C^{-i} is the set of all b such that $D^*_{(b)}$ does not contain observation i.

Bootstrap methods

Solves overfitting problem of the simple bootstrap estimate, but has training-set-size bias.

One possible further improvement is the **.632 estimator**

$$\widehat{\mathrm{Err}}_{.632} = .368 \cdot \mathrm{err} + .632 \cdot \widehat{\mathrm{Err}}_{\mathrm{Boot}(1)}.$$

Still not perfect: yet another improvement is the .632+ **estimator**.

Rather complicated . . . cross-validation is simpler!