E7441: Scientific computing in biology and biomedicine

Stochastic methods

Vlad Popovici, Ph.D.

RECETOX

Outline

Introduction to Monte Carlo methods

Numerical experiments: simulations

General approach:

1. identify the random variable of interest X
2. identify/postulate its distributional properties
3. generate one or several *large* samples *identical and independely distributed* $X_1, \dots, X_n$ from the distribution of X
4. estimate the quantity of interest (e.g. estimate $\mathbb{E}X$ using sample average) and assess its accuracy (e.g. via confidence intervals)

Random number generators (RNGs)

- all random variables can be generated by transforming a *uniformly distributed* random variable $X \in U(0, 1)$
- there is no algorithmic (deterministic) way of generating infinitely long sequences of true random numbers
- computers generate *pseudorandom numbers*
- there exist devices to generate (believed to be) random sequences: e.g. radioactive decay: the time elapsed between emission of two consecutive particles (α, β, γ). See: http://www.fourmilab.ch/hotbits

RNGs, cont'd

- two aspects:
  1. generate *good* pseudorandom numbers in $U(0,1)$: independent and uniformly distributed
  2. find proper trasformations to the desired distribution
- you cannot prove that an RNG is truly random
- there are a batteries of tests that an RNG must pass to be *acceptable*
- for any RNG, one can find a statistical test that will reject it as a good generator

RNGs, cont'd

Formalism:

- an RNG is a structure (S, μ, f, U, g) where
  - S is a finite set of *states*
  - μ is a probability distribution on S used to select the initial *seed (state)* s_0
  - $f : S \to S$ is a *transition function*. The state of the RNG evolves according to the recurrence $s_i = f(s_{i-1})$ for $i \geq 1$
  - U is the *output space*. Usually $U = (0, 1)$
  - $g : S \to U$ is the *output function*. The numbers $u_i = g(s_i)$ are called *random numbers* produced by the RNG

RNGs, cont'd

- S is finite $\Rightarrow \exists l \geq 0, j > 0$ finite such that $s_{l+j} = s_l$
- this implies that $\forall i \geq l$, $u_{i+j} = u_i$ since both f and g are deterministic
- the smallest positive j for which this happens is called *period lenght* of the RNG and is denoted by ρ
- obviously, $\rho \leq |S|$
- ex.: if the state is represented on k bits, then $\rho \leq 2^k$

RNGs, cont'd

Quality criteria:

- extremly long period ρ
- efficient implementation
- repeatability
- portability
- availability of jump-ahead property: quickly compute the s_{i+v} given s_i, so you can partition a long sequence in subsequences to be used in parallel
- *randomness*

RNGs, cont'd

Coverage:

- let $\Psi_t = \{(u_0, \ldots, u_t) | s_0 \in S\}$
- is Ψ_t uniformly covering the hypercube $(0, 1)^t$?
- tests of *discrepancy* between the empirical distribution of Ψ_t and the uniform distribution
- *figure of merit*: a measure of the coverage quality

RNGs, cont'd

Randomness and *i.i.d*:

- statistical tests: try to detect empirical evidence against H_0: "u_i are realizations of i.i.d $U(0,1)$". Example: diehard tests (Marsaglia, 1995)
- passing more tests improves the confidence in RNG, but cannot *prove* the RNG is foolproof for all cases
- *good* RNG passes a set of simple tests
- *polynomial time perfect* RNG: there is no polynomial-time algorithm the can predict any given bit of u_i with a probability of success $\geq 1/2 + 2^{-k\epsilon}$, for some $\epsilon > 0$, after observing $u_0, \ldots, u_{i-1}$
- the usual RNGs are not polynomial time perfect

RNGs, cont'd

Multiple Recursive Generator has a general recurrence

$$x_i = (a_1 x_{i-1} + \cdots + a_k x_{i-k}) \bmod m$$

where m (modulus) and k (order) are integers carefully selected, and coefficients $a_1, \ldots, a_k \in \mathbb{Z}_m$.
The state is $s_i = (x_{i-k+1}, \ldots, x_i)^T$.
When m is prime, it is possible to select a_i such that the period length $\rho = m^k - 1$.

RNGs, cont'd

Example (historical, not in serious use anymore): MLCG (Lehmer, 1948): multiplicative linear congruential generator:

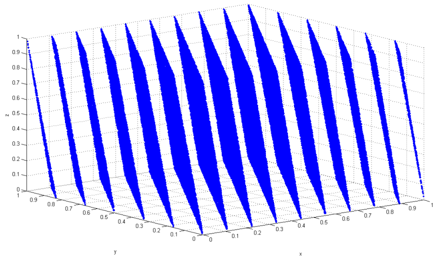
$$s_{i+1} = (a_1 s_i + a_0) \bmod m$$

This generates integers that are converted to $(0, 1)$ by division with m.
Weakness: (Marsaglia, 1968): if $(s_i, \ldots, s_{i+d})$ represent some points in a $d-$dimensional space, they have a lattice structure: they lie in a number of specific hyperplanes.

RNGs, cont'd

Famous multipliers ($a_0 = 0$):

- $a_1 = 23, m = 10^8 + 1$: original version, has higher order correlations
- $a_1 = 65539, m = 2^{29}$: infamous RANDU generator (IBM 360 series, in the 1970s): catastrophic higher order correlations
- $a_1 = 69069, m = 2^{32}$ (Marsaglia, 1972): good properties and converage up to 6 dimensions



(x, y, z) coordinates taken as consecutive values generated by RANDU ($a_1 = 65539, m = 2^{29}$) - from Wikipedia

RNGs, cont'd - Exercise

- write a function

```
random_sample_mlcg(n, a0=0, a1=20, m=53, s0=21)
```

  which implements the procedure MLCG (with some default parameters), and returns a sequence of *n* numbers.
- generate a sequence and plot u_{i+1} vs u_i

```
u = random_sample_mlcg(200)
plt.scatter(u[2:],u[:-1])
```

- discuss!

RNGs, cont'd - Exercise

- let $n = 20000$
- execute

```
n = 20000
u = random_sample_mlcg(n, a0=0, a1=65539, m=2**31, seed=10)
z = (u - 0.5) / (2**31-1)
```

- is the histogram reasonably uniform?

```
_ = plt.hist(z, bins=20)
```

- what about the coverage of $(0, 1) \times (0, 1)$?

```
z1 = z[:-2]; z2 = z[1:-1]; z3 = z[2:]; plt.scatter(z1, z2)
```

- any structure?

```
i = np.argwhere(z3 < 0.01); plt.scatter(z1[i], z2[i])
```

- discuss!

RNGs, cont’d

In general: don’t let the RNG to be "randomly" selected!

- for serious work, always set the seed, check the RNG, etc: they might be version-dependent; also you want other to be able to reproduce your results
- read the help for `numpy.random`
- using `numpy` one can specify the generator and a wide range of distributions using something like

```
numpy.random.<GENERATOR>.<DISTRIBUTION>(<parameters>)
```

like `numpy.random.default_rng(seed=42).uniform(0, 1, 20)`

Non-uniform r.v. generation (NRNG)

Requirements:

- correctness: a good approximation of the theoretical distribution
- robustness: RNG should work well on a large range of parameters
- efficiency

NRNG: inversion method

- best choice, when feasible
- to generate X with distribution function F, starting from a uniform variate $U \in (0,1)$, apply the inverse F^{-1} to U:

$$X = F^{-1}(U) := \min\{x | F(x) \geq U\}$$

- easy to see that the distribution of X is as required:

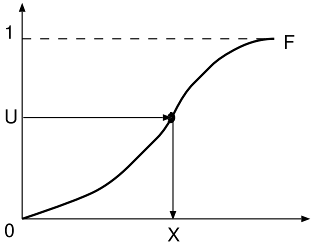
$$P[X \leq x] = P[F^{-1}(U) \leq x] = P[U \leq F(x)] = F(x)$$

- for some distributions, F^{-1} can be obtained analytically. Ex.: Weibull distribution $F(x) = 1 - \exp(-(x/\beta)^{\alpha})$, with $\alpha, \beta > 0$; has the inverse $F^{-1}(U) = \beta[-\ln(1-U)]^{1/\alpha}$
- other distributions do not have a close form inverse: e.g. normal, χ^2,... $\Rightarrow$ approximations

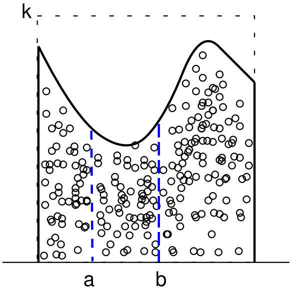
NRNG: inversion method, cont'd

Example (principle of inversion):

```
# return X with cdf F, for a
# uniform r.v. 0 < U < 1
# (look-up table method)
X = 0
while (F(X) < U) X = X + 1
return (X)
```



NRNG: Rejection method



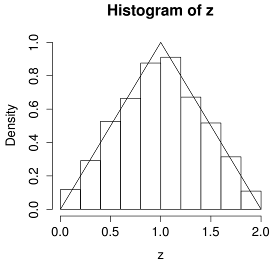
- consider F with a compact support and bounded $F(x) \leq k$
- consider a series of points (X_i, Y_i) uniformly distributed under the density function
- the distribution of X_i is the same as the distribution of X (F): $P[a < X_i < b] =$ probability of a point falling in the region $= \int_a^b F(x)dx$
- procedure:
  1. generate $X \sim U[a, b]$ and $Y \sim U[0, 1]$ independently
  2. if $Y < F(X)$ return X, otherwise repeat

NRNG: Rejection method - Exercise

Implement the rejection method for generating random variates from the pdf

$$F(x) = \begin{cases} x & \text{if } 0 < x < 1 \\ 2 - x & \text{if } 1 \leq x < 2 \\ 0 & \text{otherwise} \end{cases}$$

Generate $n = 5000$ r.v. in $[0, 2]$ and plot their histogram.



Generating normally distributed r.v.

- you can use the rejection method
- alternative: Box-Muller algorithm: based on the observation that the coordinates of points in a 2D Cartesian system described by 2 independent normal distributions correspond to polar coordinates that are realizations of 2 independent uniform distributions
- Box-Muller transform: if U_1, U_2 are independent uniformly distributed on (0,1), then

$$Z_1 = r\cos\theta = \sqrt{-2\ln U_1}\cos(2\pi U_2)$$
$$Z_2 = r\sin\theta = \sqrt{-2\ln U_1}\sin(2\pi U_2)$$

Improved Box-Muller algorithm, with rejection step:

1. generate $U_1, U_2 \sim U(-1, 1)$
2. accept $S^2 = U_1^2 + U_2^2$ if $S^2 < 1$, else go to step 1
3. set $W = \sqrt{-2\frac{\ln S^2}{S^2}}$
4. return $X = U_1 W$ and $Y = U_2 W$

Exercise: Implement the procedure above in PYTHON!

Other methods for NRNG

- kernel density estimation: approximate the inverse using a kernel for which efficient generators exist
- composition: consider F to be a convex combination of several distributions F_j:

  $$F(x) = \sum_j p_j F_j(x)$$

  To generate from F, one generates J with probability p_j and then generates X from F_j
- convolution: if $X = Y_1 + \cdots + Y_n$, with Y_j independent with specified distributions, then generate the Y_j's and sum them
- etc etc
- `numpy.random` has efficient implementations for many standard distributions

MC methods for inference

General approach:

1. identify the random variable of interest X
2. identify/postulate its distributional properties
3. generate one or several *large* samples *identical and independently distributed* $X_1, \ldots, X_n$ from the distribution of X
4. estimate the quantity of interest (e.g. estimate $\mathbb{E}X$ using sample average) and assess its accuracy (e.g. via confidence intervals)

MC inference about the mean

Reminder:

- problem: compute $z = \mathbb{E}Z$ when z is not available analytically, but Z can be simulated
- consider n replicates $Z_1, \dots, Z_n$ of Z and estimate z by the empirical mean $\hat{z} = \sum_i Z_i / n$
- denote $\sigma^2 = Var\{Z\} < \infty$
- central limit theorem:

$$\sqrt{n}(\hat{z} - z) \to \mathcal{N}(0, \sigma^2), \text{ as } n \to \infty$$

- from this, an $1 - \alpha$ confidence interval can be obtained as

$$\left(\hat{z} - z_{1-\alpha/2}\frac{\sigma}{\sqrt{n}}, \hat{z} - z_{\alpha/2}\frac{\sigma}{\sqrt{n}}\right)$$

where z_α denotes the $\alpha-$quantile of the normal distribution $(\Phi(z_\alpha) = \alpha)$

MC for inference about the mean - Exercise

Implement the following procedure:

- write the PYTHON function `pdf1(n)` to generate $n = 1000$ r.v. drawn from

$$f(X) = 0.2N_1(X) + 0.3N_2(X) + 0.5N_3(X)$$

  where N_i are Gaussians with parameters $\mu_1 = 0, \sigma_1 = 0.5$, $\mu_2 = 6.5, \sigma_2 = 1.25, \mu_3 = 14.5, \sigma_3 = 0.75$. Do not use for loops or any function from the various nonstandard packages!
- plot the histogram
- repeat the procedure for $n = 10000$ and $n = 100000$. what do you see?

- generate $p = 1000$ samples of $n = 1000$ r.v.: $X[p \times n]$
- compute $\hat{x}_i$ as the sample average for each of the p samples and the grand average $\hat{X}$
- what is the true mean of this mixture of Gaussians?
- test the normality of the distribution of $\hat{x}_i$ (find an appropriate test!)
- estimate the 95% empirical confidence interval (using quantiles of the distribution of $\hat{x}_i$) and compare it with the theoretical one (using sample variance for σ^2) obtained from a single sample (say, X[1,])

Introduction to bootstrapping

Introduction

- resampling technique for statistical inference: assess uncertainty
- especially useful when no assumptions can be made on the underlying model
- confidence intervals without distributional assumptions
- there are many versions of bootstrapping

Example (from Efron, Tibshirani, 1993):

Group	Heart attacks	Subjects
aspirin	104	11037
placebo	189	11034

The odds ratio:

$$\hat{\theta} = \frac{104/11037}{189/11034} = 0.55$$

so it seems that aspirin reduced the incidence of heart attacks.

Log-odds can be approximated by the normal distribution, so we use it to construct a 95% CI. Standard error is

$$SE(\log(OR)) = \sqrt{1/104 + 1/189 + 1/11037 + 1/11034} = 0.1228$$

giving a 95% CI for $\log\theta$:

$$\log\hat{\theta} \pm 1.96 \times SE(\log(OR)) = (-0.839, -0.357)$$

with a corresponding 95% for θ obtained by exponentiating: $(0.432, 0.700)$.

At the same time, aspirin seems to have a detrimental effect on strokes

Group	Stroke	Subjects
aspirin	119	11037
placebo	98	11034

which leads to an odds ratio of $\hat{\theta} = 1.21$ with a 95% CI of $(0.925, 1.583)$.

...and how bootstrap would proceed to infering the CI:

- create a sample for the treatment (s_1) and one for the placebo (s_2) group as vectors containing as many 1s as case there are
- draw *with replacement* a random sample from s_1 and from s_2, of the same size as the groups
- compute the odds ratios based on the drawn samples
- repeat the process a number of times and record all the odds ratios computed
- using their empirical distribution, estimate the CI of interest

A naive implementation

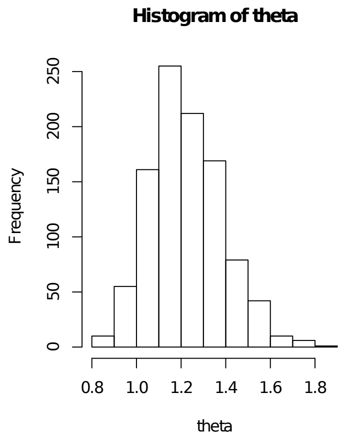
```
n1 = 11037
n1_cases = 119
n2 = 11034
n2_cases = 98

s1 = np.ones((n1, ), dtype=np.int64); s1[n1_cases:] = 0
s2 = np.ones((n2, ), dtype=np.int64); s2[n2_cases:] = 0

B = 1000  # no. of bootstraps
p = n2 / n1
theta = np.zeros((B,), dtype=np.float64)

for i in np.arange(B):
  theta[i] = p * np.sum(
    np.random.choice(s1, n1, replace=True) ) /
    np.sum( np.random.choice(s2, n2, replace=True) )

_ = plt.hist(theta, 50)
print("95% Confidence interval for theta: ",
  np.quantile(theta, q=(0.025, 0.975)))
```



- the CI estimate by the quantiles is not the most precise nor efficient that can be obtained by bootstrapping
- it works for symmetric, close to normal distributions of the bootstrap replicate

The empirical distribution

- the underlying probability distribution F generates the observed sample:

$$F \to (x_1, \dots, x_n) = \mathbf{x}$$

- the empirical distribution $\hat{F}$ is the *discrete* distribution that puts probability $1/n$ at each value $x_i, i = 1, \dots, n$
- $\hat{F}$ assigns to a set A in the sample space of x its empirical probability:

$$\widehat{Prob}\{A\} = \frac{\#\{x_i \in A\}}{n} = Prob_{\hat{F}}\{A\}$$

- a *parameter* is a functional of the distribution function, $\theta = t(F)$. Example: the mean
$$\mu(F) = \int x dF(x)$$
- a *statistic* is a function of the sample x. Example: the sample average,
$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$$
- the plug-in estimate of a parameter $\theta = t(F)$ is defined to be
$$\hat{\theta} = t(\hat{F})$$
(sometimes called summary statistics, estimates or estimator)

Bootstrap estimate of the standard error

- bootstrap sample: $\hat{F} \rightarrow (x_1^*, \ldots, x_n^*) = \mathbf{x}^*$ (resampling with replacement)
- let $\hat{\theta} = s(\mathbf{x})$ be an estimate for the parameter of interest
- the question is: what is the standard error of the estimate?
- bootstrap replication of $\hat{\theta}$ is

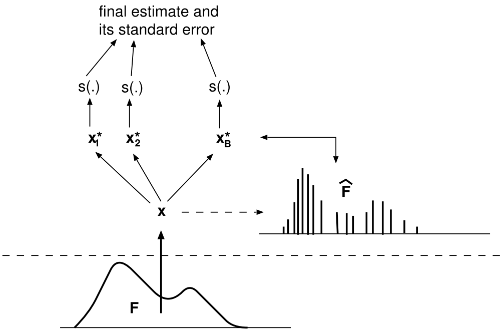
$$\hat{\theta}^* = s(\mathbf{x}^*)$$

- *ideal bootstrap estimate* of SE:

$$se_{\hat{F}}(\hat{\theta}^*)$$

  i.e. the standard error of $\hat{\theta}$ for data sets of size n randomly sampled from $\hat{F}$
- unfortunately, close-form formulas exist only for some estimators

General form of the bootstrap method



- by resampling with replacement from **x** one samples from the empirical distribution $\hat{F}$
- $\mathbf{x}_b^*$ are the bootstrap samples of size n
- $s(\mathbf{x}_b^*) = \hat{\theta}_b^*$ are the bootstrap replications of the parameter of interest θ

Bootstrap estimation of standard errors

1. select B independent bootstrap samples $\mathbf{x}_1^*, \dots, \mathbf{x}_B^*$
2. evaluate the bootstrap replicate of each bootstrap sample $\hat{\theta}_b^* = s(\mathbf{x}_b^*)$, $b = 1, 2, \dots, B$
3. estimate the standard error $se_{\hat{F}}(\hat{\theta})$ by the sample standard deviation of the B replications:

$$\widehat{se}_B = \sqrt{\frac{1}{B-1} \sum_{b=1}^{B} \left[\hat{\theta}_b^* - \hat{\theta}_0^*\right]^2}$$

where $\hat{\theta}_0^* = \frac{1}{B} \sum_{b=1}^{B} \hat{\theta}_b^*$

Homework

Implement the previous procedure in Python:

- write a function `bstrap_nonparam(x, B, s, ...)` which will generate B bootstrap samples $\mathbf{x}_b^*$ and for each of them will compute the bootstrap replicate of the parameter: $\hat{\theta}_b^* = s(\mathbf{x}_b^*, \cdots)$
- write a function `bstrap_theta0(T)` which computes the bootstrap estimate of the parameter, given the bootstrap replicates in the vector T ($\hat{\theta}_0^*$)
- write a function `bstrap_se(T)` which computes the bootstrap estimate of the standard error of the parameter, given the bootstrap replicates in the vector T ($\widehat{se}_B$)
- use the Rainfall data set to compute the bootstrap estimate of the mean, median and corresponding standard errors - see the Jupyter notebook for data.
- compare with textbook results! (discuss!)

Bias–corrected and accelerated CI

- the quantile-based CI is not tight enough nor robust
- idea: better exploit the quantiles of the empirical distribution by:
  - correcting the bias
  - improving convergence
- simple bootstrap quantile-based CI: for an $(1-2\alpha)$ coverage, the bounds of the CI are given by $(\hat{\theta}^{*(\alpha)}, \hat{\theta}^{*(1-\alpha)})$ where $\hat{\theta}^{*(q)}$ is the q–th quantile of the bootstrap replicates

The BCa CI is given by $(\hat{\theta}^{*(\alpha_1)}, \hat{\theta}^{*(\alpha_2)})$ where

$$\alpha_1 = \Phi\left(\hat{z}_0 + \frac{\hat{z}_0 + z^{(\alpha)}}{1 - \hat{a}(\hat{z}_0 + z^{(\alpha)})}\right)$$
$$\alpha_2 = \Phi\left(\hat{z}_0 + \frac{\hat{z}_0 + z^{(1-\alpha)}}{1 - \hat{a}(\hat{z}_0 + z^{(1-\alpha)})}\right)$$

where

- $\Phi(\cdot)$ is the standard normal CDF
- $z^{(q)}$ is the q–th quantile of standard normal distribution
- $\hat{a}$ and $\hat{z}_0$ are cleverly chosen

The parameters of BCa CIs:

$$\hat{z}_0 = \Phi^{-1}\left(\frac{\#\{\hat{\theta}^*_b < \hat{\theta}\}}{B}\right)$$

$$\hat{a} = \frac{\sum_{i=1}^{n}\left(\hat{\theta}_{(\cdot)} - \hat{\theta}_{(i)}\right)^3}{6\left[\sum_{i=1}^{n}\left(\hat{\theta}_{(\cdot)} - \hat{\theta}_{(i)}\right)^2\right]^{3/2}}$$

where

- $\hat{\theta}_{(i)}$ is the value of the parameter computed on the vector **x** with the i–th component removed (*jackknife values* of the parameter)
- $\hat{\theta}_{(\cdot)} = \sum_{i=1}^{n}\hat{\theta}_{(i)}/n$

Exercise: implement the BCa procedure in **R**: (yes, not in PYTHON)

- write a function bstrap.bca(x, B, s, ..., alpha=c(0.025, 0.05)) that returns the low and upper bounds of the CI computed by BCa method
- you can use (call) the previous function bstrap.nonparam
- compute the 90% and 95% BCa CIs for the mean of Rainfall data: bstrap.bca(Rainfall, 2000, mean)

Important properties of BCa CIs

- *transformation respecting*: the bounds of the CIs transform correctly if the parameter is changed by some function: e.g. the CIs for $\sqrt{\cdot}$-transformed parameter are obtained by taking $\sqrt{}$ of the bounds of the parameter itself
- *second order accurate*: convergence rate of $1/n$ to true coverage

Bootstrapping for tests

- consider two possibly different distributions F and G,

$$F \rightarrow \mathbf{z} = (z_1, \ldots, z_n)$$
$$G \rightarrow \mathbf{y} = (y_1, \ldots, y_m)$$

- hypotheses:

$$H_0 : F = G$$
$$H_1 : F \neq G$$

- $F = G \Leftrightarrow Prob_F\{A\} = Prob_G\{A\}$ for all sets A
- *observe* a test statistic $\hat{\theta}$ (e.g. mean difference)
- *achieved significance level (ASL)*: probability of observing that large a value under H_0:

$$ASL = Prob_{H_0}\{\hat{\theta}^* \geq \hat{\theta}\}$$

Bootstrapping hypothesis testing procedure

1. choose a test statistic (not necessary a parameter): $t(\mathbf{x})$ (for example: $t(\mathbf{x}) = \bar{\mathbf{z}} - \bar{\mathbf{y}}$)
2. draw B samples of size $n + m$ from $\mathbf{x} = (\mathbf{z}, \mathbf{y})$ and call the first n observations $\mathbf{z}^*$ and the remaining m $\mathbf{y}^*$
3. evaluate $t(\cdot)$ for each sample: $t(\mathbf{x}_b^*)$
(for example
$$t(\mathbf{x}_b^*) = \bar{\mathbf{z}}_b^* - \bar{\mathbf{y}}_b^*$$
)
for $b = 1, 2, \ldots, B$
4. approximate ASL_{boot} by
$$\widehat{ASL}_{boot} = \#\{t(\mathbf{x}_b^*) \geq t(\mathbf{x})\}/B$$

Permutation tests

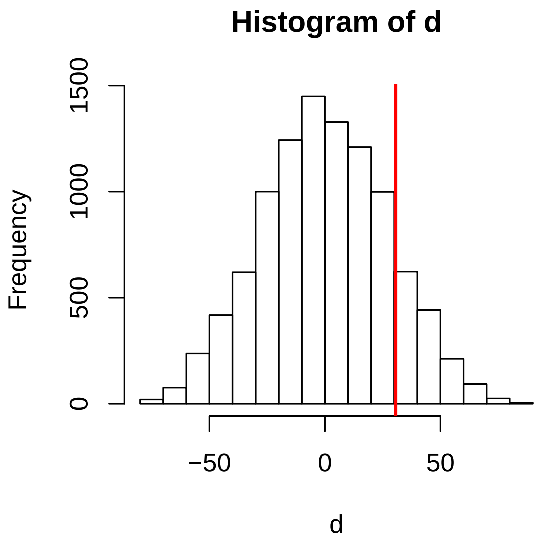
Permutation tests

- nonparametric testing procedure
- allow testing hypotheses when the properties of the test statistic under the null hypothesis are not known
- do not make assumptions on the data
- work on small data sets
- idea: generate the distribution of the test statistic under the null hypothesis *from the data*

- exact permutation tests: for (very) small data sets, generate *all* permutations and compute the corresponding test statistics
- random test: for large data sets, generate a number of random permutations, for which compute the test statistic
- test procedure: count how many times the test statistic from the permutations is more extreme than the real test statistic and reject H_0 if the proportion is below the predefined $\alpha-$level

Example - two populations tests

- consider the data vectors mouse.c and mouse.t for the *control* and *treatment* arms of an experiment (some clinical variable)
- implement a permutation testing procedure for testing
  H_0 : there is no significant difference in the clinical variable between control and treatment
  vs
  H_1 : there is a significant difference in the clinical variable between control and treatment
- which test statistic? what to permute? how many permutations?
- what should be changed if the test was about superiority of treatment vs control?


Histogram of d
Frequency
0
500
1000
1500
−50
0
50
d

Questions?