Lecture 1: Random number generation, permutation test, and the bootstrap

August 19, 2020

Statistical simulation

- Statistical simulation (Monte Carlo) is an important part of statistical method research.
- The statistical theories/methods are all based on assumptions. So most theorems state something like “if the data follow these models/assumptions, then . . .”.
- The theories can hardly be verified in real world data because (1) the real data never satisfy the assumption; and (2) the underlying truth is unknown (no “gold standard”).
- In simulation, data are “created” in a well controlled environment (model assumptions) and all truth are known. So the claim in the theorem can be verified.

Random number generator (RNG)

- Random number generator is the basis of statistical simulation. It serves to generate random numbers from predefined statistical distributions.
- Traditional methods (flip a coin or dice) work, but can't scale up.
- Computational methods are available to generate "pseudorandom" numbers.

The random number generation often starts from generating uniform(0,1). The most common method: "**Linear congruential generator**":

$$X_{n+1} = (aX_n + c) \bmod m$$

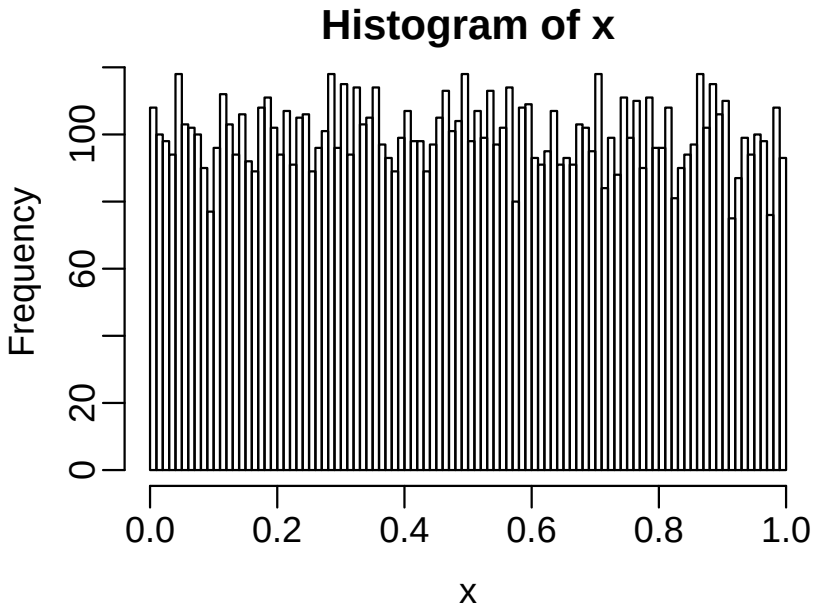
Here, a, c, and m are predefined numbers:

- X_0: random number "seed".
- a: multiplier, 1103515245 in `glibc`.
- c: increment, 12345 in `glibc`.
- m: modulus, for example, 2^{32} or 2^{64}.

$U_n = X_n/m$ is distributed as Uniform(0,1).

Linear congruential generator


```
a = 1103515245; c = 12345; m = 2^32
n = 10000
x = numeric(n)
x[1] = 1
for( i in 2:n) {
    x[i] = (a*x[i-1] + c) %% m
}
x = x/m
hist(x, 100)
```



Random number generator (RNG)

A few remarks about Linear congruential generator:

- The numbers generated will be exactly the same using the same seed.
- Want cycle of generator (number of steps before it begins repeating) to be large.
- Don't generate more than m/1000 numbers.

RNG in `R`:

- `set.seed` is the function to specify random seed.
- Read the help for `.Random.seed` for more description about random number generation in R.
- `runif` is used to generate uniform(0,1) r.v.
- My recommendation: always set and save random number seed during simulation, so that the simulation results can be reproduced.

Simulate R.V. from other distributions

When the distribution **has a cumulative distribution function (cdf)** $\boldsymbol{F}$, the r.v. can be obtained by inverting the cdf ("inversion sampling"). This is based on the theory that the cdf is distributed as Uniform (0,1):

Algorithm: Assume F is the cdf of distribution $\mathcal{D}$. Given $u \sim \text{unif}(0, 1)$, find a unique real number x such that $F(x) = u$. Then $x \sim \mathcal{D}$.

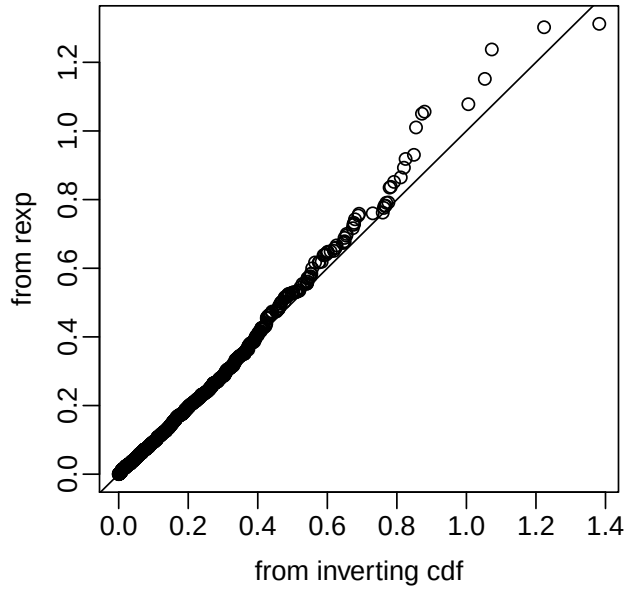
Example: exponential distribution. When $x \sim \exp(\lambda)$, the cdf is: $F(x) = 1 - exp(-\lambda x)$. The inversion of cdf is: $F^{-1}(u) = -log(1 - u)/\lambda$. Then to generate exponential r.v., do:

- Generate uniform(0,1), r.v., denote by u.
- Calculate $x = -log(1 - u)/\lambda$.

When the inverted cdf is unavailable, one has to rely on other methods such as **acceptance-rejection**. This will be covered later in MCMC classes.

Example: simulate exponential r.v.


```
lambda=5
u = runif(1000)
x = -log(1-u) / lambda
## generate from R's function
x2 = rexp(1000, lambda)
## compare
qqplot(x, x2, xlab="from inverting cdf", ylab="from rexp")
abline(0,1)
```



Simulate discrete R.V.

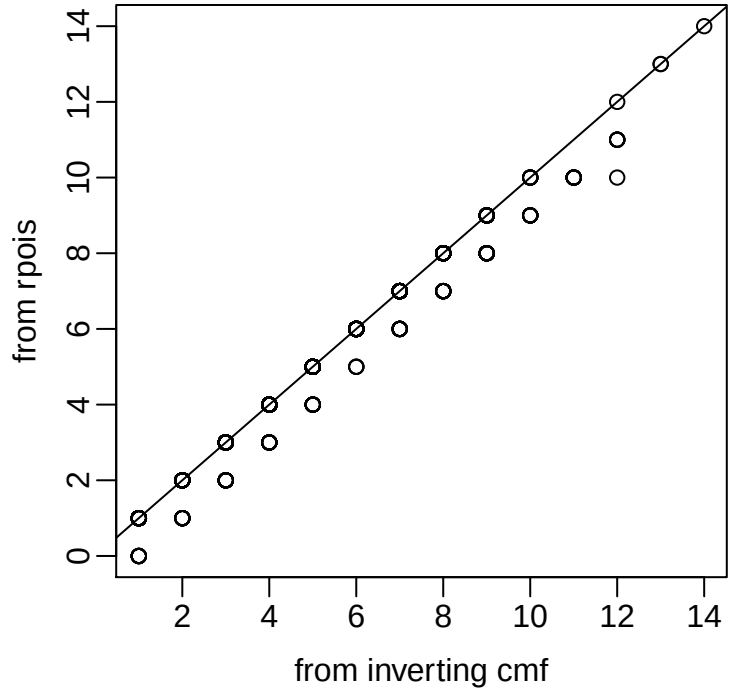
For discrete r.v. (such as from Poisson distribution), the CDF is usually called **CMF (cumulative distribution function)**, and it follows **discrete uniform distribution**. The CMF can be represented as a table.

One can use the same procedure to invert CMF and generate discrete random number. To invert the CMF, one needs to do a search in the CMF table to determine which interval covers each element of uniform rv u.

Example: to generate Poisson random number with rate 5. A couple notes:

- I use the **ppois** function in R to compute the CMF. From scratch, you should compute that from the Poisson CMF function.
- Pay attention to the use of “cut” function. This is a much cleaner and faster way to do search than using a loop.

```
lambda = 5
## generate unif rv
u = runif(1000)
## compute and invert the CMF
cmf = ppois(1:100, lambda=lambda)
ix = min(which(cmf==1))
cmf = c(0, cmf[1:ix])
cmfTbl = table(cut(u, breaks=cmf, include.lowest=TRUE))
Y = rep(1:length(cmfTbl), as.numeric(cmfTbl))
## compare
qqplot(Y, rpois(1000, lambda=lambda), xlab="from inverting cmf", ylab="from rpois")
abline(0,1)
```



Simulate random vectors

Difficulty: Generating random vectors is more difficult, because we need to consider the correlation structure.
Solution: Generate **independent** r.v.'s, then apply some kind of transformation.

Example: simulate from multivariate normal distribution $MVN(\mu, \Sigma)$
Let $\mathbf{Z}$ be a p-vector of independent $N(0,1)$ r.v.'s, Given $p \times p$ matrix $\mathbf{D}$,

$$var(\mathbf{D}^T\mathbf{Z}) = \mathbf{D}^T var(\mathbf{Z})\mathbf{D} = \mathbf{D}^T\mathbf{D}$$

The simulation steps are:

1. Perform **Cholesky decomposition** on Σ to find $\mathbf{D}$: $\Sigma = \mathbf{D}^T\mathbf{D}$.
2. Simulate $\mathbf{Z} = (z_1, \ldots, z_p)' \sim$ iid $N(0,1)$
3. Apply transformation $\mathbf{X} = \mathbf{D}^T\mathbf{Z} + \mu$.

R function `mvrnorm` available in `MASS` pacakge.

Generating multivariate random vector from other distributions are usually harder.
Recommended book: ***Multivariate Statistical Simulation: A Guide to Selecting and Generating Continuous Multivariate Distributions***.

Example: generate from multivariate normal


```
## specify mean and variance/covariance matrix
mu = c(0,1)
Sigma = matrix(c(1.7, 0.5, 0.5, 0.8), nrow=2)

## Cholesky decomposition
D = chol(Sigma)

## generate 500 Z's.
Z = matrix(rnorm(1000), nrow=2)
## transform
X = t(D) %*% Z + mu

## check the means X
> rowMeans(X)
[1] -0.08976896  0.95802769

## check the variance/covariance matrix of X
> cov(t(X))
          [,1]      [,2]
[1,] 1.7392114 0.5609027
[2,] 0.5609027 0.7380548
```

Permutation test

- In statistical inference, it is important to know the distribution of some statistics under null hypothesis (H_0), so that quantities like p-values can be derived.
- The null distribution is available theoretically in some cases. For example, assume $X_i \sim N(\mu, \sigma^2), i = 1, \ldots, n$. Under $H_0 : \mu = 0$, we have $\overline{X} \sim N(0, \sigma^2/n)$. Then H_0 can be tested by comparing $\overline{X}$ with $N(0, \sigma^2/n)$.
- When null distribution cannot be obtained, it is useful to use **permutation test** to "create" a null distribution from data.

The basic procedure of permutation test for H_0

- Permute data under H_0 for a number of times. Each time recompute the test statistics. The test statistics obtained from the permuted data form the null distribution.
- Compare the observed test statistics with the null distribution to obtain statistical significance.

Permutation test example

Assume there are two sets of independent normal r.v.'s with the same known variance and different means: $X_i \sim N(\mu_1, \sigma^2)$, $Y_i \sim N(\mu_2, \sigma^2)$. We wish to test $H_0 : \mu_1 = \mu_2$.

Define test statistics: $t = \overline{X} - \overline{Y}$. We know under null, we have $t \sim N(0, 2\sigma^2/n)$ (assuming same sample size n in both groups). Using permutation test, we do:

1. Pool X and Y together, denote the pooled vector by Z.
2. Randomly shuffle Z. For each shuffling, take the first n items as X (denote as X^*) and the next n items as Y (denote as Y^*).
3. Compute $t^* = \overline{X^*} - \overline{Y^*}$.
4. Repeat steps 2 and 3 for a number of times. The result t^*'s form the null distribution of t.
5. To compute p-values, calculate $Pr(|t^*| > |t|)$.

NOTE: the random shuffling is based on H_0, that X and Y are iid distributed.

Permutation test example - R code

```
> x=rnorm(100, 0, 1)
> y=rnorm(100, 0.5, 1)
> t.test(x,y)

Welch Two Sample t-test

data:  x and y
t = -1.9751, df = 197.962, p-value = 0.04965

> nsims=50000
> t.obs = mean(x) - mean(y)
> t.perm = rep(0, nsims)
> for(i in 1:nsims) {
+     tmp = sample(c(x,y))
+     t.perm[i] = mean(tmp[1:100]) - mean(tmp[101:200])
+ }
> mean(abs(t.obs) < abs(t.perm))
[1] 0.04814
```

Permutation test - another example

- Under linear regression setting (without intercept) $y_i = \beta x_i + \epsilon_i$. We want to test the coefficient: $H_0 : \beta = 0$.
- Observed data are (x_i, y_i) pairs.
- Use ordinary least square estimator for β, denote as $\hat{\beta}(\mathbf{x}, \mathbf{y})$.

The permutation test steps are:

1. Keep y_i unchanged, permute (change the orders of) x_i to obtain a vector, denoted as x_i^*.
2. Obtain estimate under the permuted data: $\hat{\beta}^*(\mathbf{x}^*, \mathbf{y})$
3. Repeat steps 1 and 2. $\hat{\beta}^*$ form the null distribution for $\hat{\beta}$.
4. P-value = $Pr(|\hat{\beta}^*| > |\hat{\beta}|)$.

NOTE: the random shuffling of x_i is based on the H_0, that is there is no association between x and y.

Permutation test another example - R code

```
> x = rnorm(100); y = 0.2 * x + rnorm(100)
> summary(lm(y~x-1))

Coefficients:
  Estimate Std. Error t value Pr(>|t|)
x   0.1502     0.1050   1.431    0.156

> nsims=5000
> beta.obs = coef(lm(y~x-1))
> beta.perm = rep(0, nsims)
> for(i in 1:nsims) {
+     xstar = sample(x)
+     beta.perm[i] = coef(lm(y~xstar-1))
+ }
> mean(abs(beta.obs) < abs(beta.perm))
[1] 0.157
```

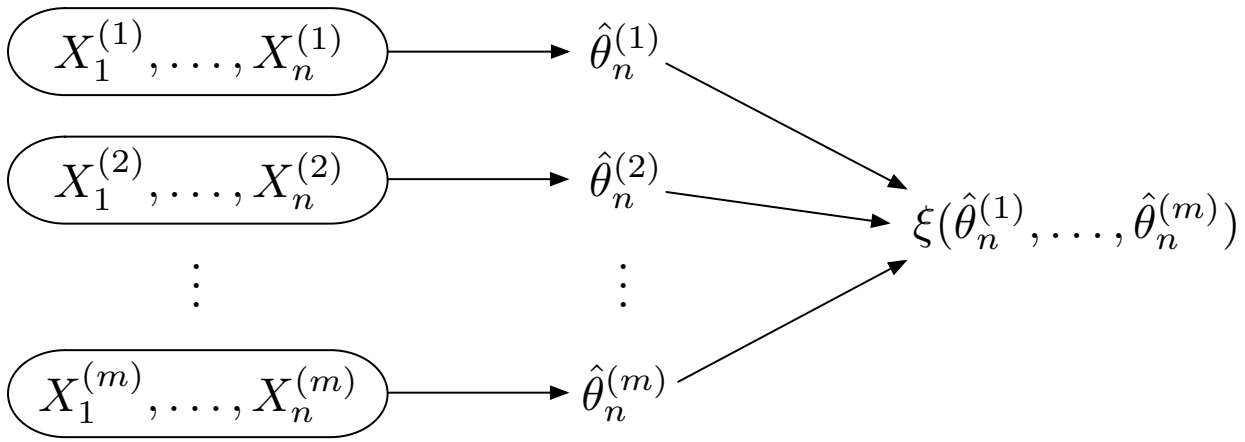
The bootstrap

- “Bootstrap” is a simple procedure to estimate the sampling distribution (such as mean, variance, confidence interval, etc.) of some statistics.
- Developed by Brad Efron (see Efron (1979) AOS), extending the “jackknife” algorithm.
- The basic idea is to resample the observed data **with replacement** and create a distribution of the statistics.
- Show good performances compared with jackknife.
- Computationally intensive, but algorithmically easy.

Problem setup

- Observe data $\mathbf{x} = \{x_1, \ldots, x_n\}$.
- Parameter of interest is θ, for example, $\theta = E[X]$.
- Let $\hat{\theta}(\mathbf{x})$ be an estimator for θ (such as the MLE). Note $\hat{\theta}$ is a random variable.
- We want to obtain some quantity from $\hat{\theta}$, denoted as $\xi(\hat{\theta})$, for example, the distributional properties of $\hat{\theta}$: its mean, variance, quantiles, etc.

Ideally, we would need to observe a number of independent datasets, compute $\hat{\theta}$ from each of them, and then compute the $\xi(\hat{\theta})$.



Parametric bootstrap

Assume $x_i \sim$ iid $f(\theta)$, where f is known.

The **parametric bootstrap** procedure involves repeating following steps for N times. At the k^{th} time, do:

1. Simulate $\mathbf{x}_i^*$ iid from $f(\theta)$.
2. Compute $\hat{\theta}_i(\mathbf{x}_i^*)$.

Then ξ can be calculated from $\hat{\theta}_i(\mathbf{x}_i^*)$.

Non-parametric bootstrap

Problem setup is the same as in parametric bootstrap, except that the distribution f is unknown. In this case, since $\mathbf{x}$ cannot be generated from a known parametric distribution, they will be drawn from the observed data.

The **non-parametric bootstrap** procedure involves repeating following steps for N times. Assume the observed data has n data points. At the k^{th} time, do:

1. Draw $\mathbf{x}_i^*$ from the observed data $\mathbf{x}$. Note that $\mathbf{x}_i^*$ must have the same length as $\mathbf{x}$, and the drawing is sampling ***with replacement***.
2. Compute $\hat{\theta}_i(\mathbf{x}_i^*)$.

Then the ξ can be calculated from $\hat{\theta}_i(\mathbf{x}_i^*)$.

So the only difference between parametric and non-parametric bootstrap is the way to generate data:

- In parametric bootstrap: simulate from parametric distribution.
- In non-parametric bootstrap: sample with replacement from observed data.

Bootstrap example: linear regression

Problem setup:

- Under linear regression setting (again we omit the intercept to simplify the problem): $y_i = \beta x_i + \epsilon_i$.
- We wish to study the property of OLS estimator, denoted by $\hat{\beta}(\mathbf{x}, \mathbf{y})$.

Parametric bootstrap is based on assumption that $\epsilon_i \sim N(0, \sigma^2)$. Steps are:

1. Obtain $\hat{\beta}(\mathbf{x}, \mathbf{y})$ from observed data.
2. Sample $\epsilon_i^* \sim N(0, \sigma^2)$.
3. Create new $\mathbf{y}$: $y_i^* = \hat{\beta} x_i + \epsilon_i^*$.
4. Estimate the coefficient based on new data: $\hat{\beta}^*(\mathbf{x}, \mathbf{y}^*)$

Repeat steps 2–4 for many times, then the properties of OLS estimator (such as mean/variance) can be estimated from $\hat{\beta}^*(\mathbf{x}^*, \mathbf{y})$.

Bootstrap example: linear regression (cont.)

Non-parametric bootstrap doesn't require the distributional assumption on ϵ_i. The residuals are resampled from the observed values.

1. Obtain $\hat{\beta}(\mathbf{x}, \mathbf{y})$ from observed data.
2. Compute the observed residuals: $\hat{\epsilon}_i = y_i - \hat{\beta} x_i$.
3. Sample ϵ_i^* by drawing from $\{\hat{\epsilon}_i\}$ with replacement.
4. Create new $\mathbf{y}$: $y_i^* = \hat{\beta} x_i + \epsilon_i^*$.
5. Estimate the coefficient based on new data: $\hat{\beta}^*(\mathbf{x}, \mathbf{y}^*)$

Bootstrap example: linear regression – R codes

We will estimate the 95% confidence interval for regression coefficient.

Generate data and compute theoretical value:

```
> x = rnorm(100)
> y = 0.5 * x + rnorm(100)
> fit = lm(y~x-1)
> confint(fit)
     2.5 %    97.5 %
x 0.4002036 0.7638744
```

Bootstrap example: linear regression – R codes (cont.)

Parametric bootstrap - sample residual from normal distribution:

```
> nsims = 1000
> beta.obs = coef(fit)
> beta.boot = rep(0, nsims)
> for(i in 1:nsims) {
+     eps.star = rnorm(100)
+     y.star = beta.obs * x + eps.star
+     beta.boot[i] = coef(lm(y.star~x-1))
+ }
> quantile(beta.boot, c(0.025, 0.975))
     2.5%     97.5%
0.4098746 0.7595921
```

Bootstrap example: linear regression – R codes (cont.)

Non-parametric bootstrap - sample residual from observed values:

```
> eps.obs = y - beta.obs*x
> for(i in 1:nsims) {
+     eps.star = sample(eps.obs, replace=TRUE)
+     y.star = beta.obs *x + eps.star
+     beta.boot[i] = coef(lm(y.star~x-1))
+ }
> quantile(beta.boot, c(0.025, 0.975))
     2.5%     97.5%
0.4011628 0.7690787
```

Bootstrap in big data

In big data set, bootstrap poses significant computational challenge, since the bootstrapped data must have the same length as the original data.

The "***b* out of *n* bootstrap**"
Bickel et al. (1997) *Resampling fewer than n observations: Gains, losses, and remedies for losses*, **Statistica Sinica**:

1. Repeatedly subsample b data points **with replacement** from the original data (of size n), and then compute $\hat{\theta}_b$ from the subsample.
2. Compute ξ from $\hat{\theta}_b$'s.
3. Analytically correct the results using prior knowledge of the convergence rate of $\hat{\theta}_b$.

Bootstrap in big data (cont.)

"**Bag of Little Bootstrap**" (BLB) approach, from Kleiner et al. (2014) *A scalable bootstrap for massive data.* **JRSSB**:

1. Subsample s subsets from the original data (of size n), each with size b.
2. For each subsample, do:
   (a) Repeatedly sample n points with replacement from the subsample (up-sampling), and compute $\hat{\theta}_n^*$ on each resample.
   (b) Compute an estimate of ξ based on $\hat{\theta}_n^*$'s, denote as ξ_s.
3. Average ξ_s's as the final estimate of ξ.

Advantages

- More automatic than the "b out of n bootstrap" approach.
- Since $b << n$, the size-n subsamples are highly repetitive. For example each resample can be represented as a vector of counts from an n-trial uniform multinomial distribution over b objects. This leads to less memory usage and faster computing.
- Highly parallelizable.

A quick review

- Random number generation:
  - Linear congruential generator for generating Uniform(0,1) r.v.;
  - Inverting cdf to generate r.v. from other distributions;
  - simulate random vectors from MVN.
- Permutation test. The key is to shuffle data under null hypothesis, then recompute test statistics and form the null distribution.
- Bootstrap algorithm. Include parametric (draw from parametric distribution) or non-parametric (draw from observed data with replacement).
- Smart approaches for big data bootstrap.
- After class: review slides, play with the R codes.