



HEC MONTRÉAL

Theme 2

Simulation and
Bootstrap

Montréal Casino mishap



NEWS

'Computer nerd' outsmarts casino Wins \$200,000 pot - twice in a row

CP
352 mots
20 avril 1994



- Ask Daniel Corriveau how he beat staggering odds to win \$400,000 at the Montreal Casino, and he'll talk about a butterfly flapping its wings in Beijing.
- Celebrated by Quebecers as a mild-mannered genius who beat the system, the province's latest hero is a computer nerd who claims to have used "chaos theory" to defy mind-numbing odds at the casino. [...] if a butterfly flaps its wings in Beijing, it will have an effect on the weather system in New York City.
- Corriveau visited the casino about a dozen times over four months, writing down the winning sequences of numbers.
- "I found the same 19-number sequence twice in 240 draws," he explained

How to generate (pseudo-)random numbers?

You need two ingredients:

- A uniform random number generator
- Formulas or algorithms to change those uniform numbers into the distribution that you need.

The two tools are very different in nature.



Simple Random Number Generator (RNG)

- Computers are not able to generate pure randomness.
- RNGs are deterministic algorithms that generate numbers that look random.
- All RNGs start from an initial value called **seed**.

Let us consider a “[linear congruential generator](#)” (with parameters a , c and m).

The seed x_0 is a number between 0 and $m - 1$.

Then pseudo-random numbers are generated as:

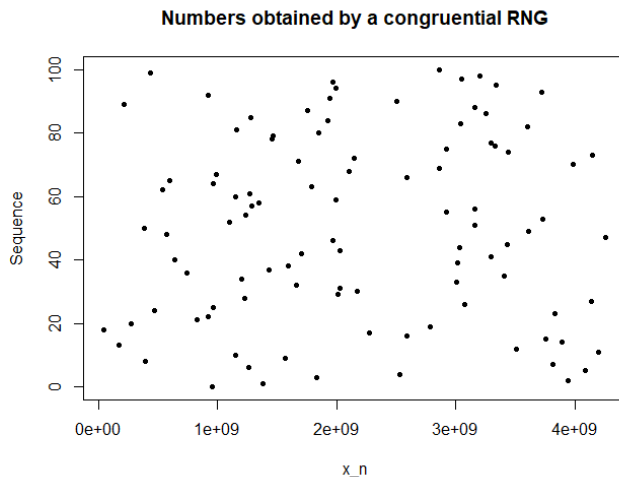
$$x_{n+1} = (ax_n + c) \bmod m$$

- The “mod” operator keeps the remainder of the division of $ax_n + c$ by m .
- The result of this operation generates integers between 0 and $m - 1$.
- With appropriate choices of a , c , and m , the resulting sequence of x_n will look perfectly random.
- The RND as a period: after a while, the algorithm will necessarily fall again on x_0 and reproduce the same sequence again.

Example

Consider a generator with $a = 1,664,525$, $c = 1,013,904,223$ and $m = 2^{32}$.

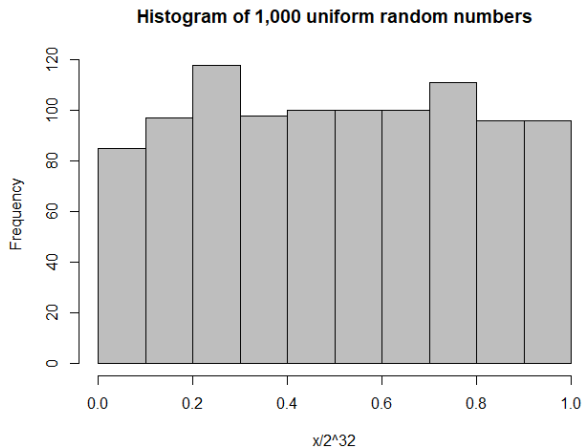
The plot below shows values for (x_n, n) for $n = 1, \dots, n$.



No clear pattern is apparent to the eye

To get numbers that are uniform on the unit interval, we use x_n/m .

Here is the histogram of 1,000 values from our RNG.



The histogram looks rather flat, which is expected for uniform numbers.

What makes a good RNG?

A good RNG:

- Has a very long period (time before it falls on x_0 again), much longer than the number of numbers needed (good choices for a , c and m are key),
- Does not show any non-random pattern in its output.



This cartoon is incorrect: RNGs do get tested, and very thoroughly so.

Testing uniform numbers

To check that the numbers generated are random, they are used to generate different random situations, and their result is compared to the values that would be expected.



The “[diehard](#)” battery of test (Marsaglia, 1995) generates:

- Games of craps and count the wins,
- Numbers and look at the length of ascending sequences,
- Groups of 5 numbers and look at the frequency of each permutation,
- Groups of 100 numbers and look at their sum,
- Etc.

The values obtained are compared to the distribution expected if the numbers were coming from pure randomness. A good RNG fails those tests at just the right rate.

Those experiments are designed to detect some regularity in the numbers produced.

Other batteries of tests also exist, including TESTU01 by L'Écuyer and Simard (UdeM).

Default RNG in R: Mersenne-Twister

Details on [Wikipedia](#).

Similar in spirit:

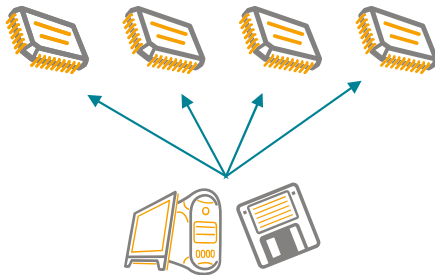
- need a seed to start,
- a recurrence formula is used,
- instead of integers x_n , large binary matrices are generated,
- instead of one number, we get many (623) random numbers out of each iteration,
- although it is larger, the period is still finite (you can loopback).

The idea is not to code our own algorithm but to understand how they work.

Parallel computing

In parallel computing, many nodes will be running simulations simultaneously.

This is equivalent to having multiple computers run side-by-side.



With the parallel library, many instances of R are run simultaneously on the same machine, each using a different core.

- Never initiate the RNG on the different cores with the same seed. They would yield the same output,
- Select seeds carefully: if initial values are too close, one core could catch up with the seed of the other (there is a function for doing this right).

Generating nonuniform random variables

Uniform numbers on the intervals are the building block for other distributions.

With a good RNG, the following strategies can then be employed:

- Transformation
- Rejection algorithms
- Markov chain Monte Carlo
- Simulating a model

There are also a few less common tricks.

Transformation

A transformation consists of applying a function to a random variable.

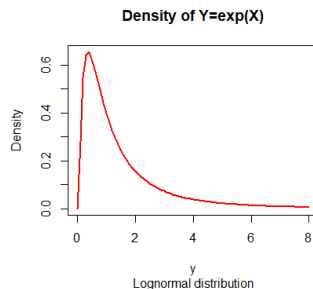
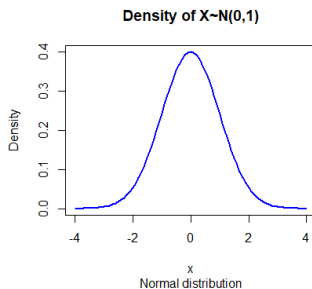
Let X be a random variable with density $f_X(x)$.

Define $Y = h^{-1}(X)$ for some function h .

Then the distribution of Y is

$$f_Y(y) = f_X\{h(y)\} \left| \frac{d}{dy} h(y) \right|$$

The formula is almost identical to a change of variables when evaluating an integral.
In multiple dimensions, the element to the right is the determinant of the Jacobian.



Equivalently, one could look at the cumulative distribution function (CDF):

$$F_Y(y) = F_X\{h(y)\}$$

Since $P(Y \leq y) = P\{X \leq h(y)\}$.

This leads to a very important result:

If U is uniform on $[0,1]$, then $Y = F_X^{-1}(U)$ follows the same distribution as X .

The proof is even quite straightforward:

$$F_Y(y) = P(Y \leq y) = P(F_X^{-1}(U) \leq y) = P\{U \leq F_X(y)\} = F_X(y)$$

Since the CDF of a uniform distribution is

$$F_U(u) = \begin{cases} 0 & \text{if } u \leq 0 \\ u & \text{if } 0 < u \leq 1 \\ 1 & \text{if } u > 1 \end{cases}$$

In other words, uniform numbers may be changed into **any** distribution.

Special transformations

Computing the CDF or its inverse is not always easy. There are other transformations that have been designed to transform variables in a more efficient way. For instance,

Box-Mueller: If U_1, U_2 are independent uniform variables, then

$$X_1 = \sqrt{-2 \ln U_1} \cos(2\pi U_2)$$

$$X_2 = \sqrt{-2 \ln U_1} \sin(2\pi U_2)$$

Give two independent normal variables with mean 0 and variance 1.

Multivariate normal: Starting with X_1, \dots, X_p independent $N(0,1)$, we have that

$$\mathbf{Y} = \mathbf{\Sigma}^{1/2} \mathbf{X} + \boldsymbol{\mu} = \mathbf{\Sigma}^{1/2} \begin{bmatrix} X_1 \\ \vdots \\ X_p \end{bmatrix} + \boldsymbol{\mu}$$

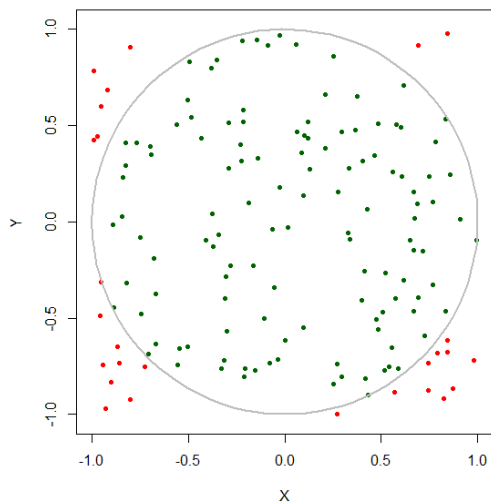
Follows a multivariate normal distribution in p dimensions with mean $\boldsymbol{\mu}$ and covariance $\mathbf{\Sigma}$. This transformation uses properties of the normal distribution rather than the general result on the inversion of the CDF.

Rejection algorithms

Rejection algorithms generate points but will select only some of them.

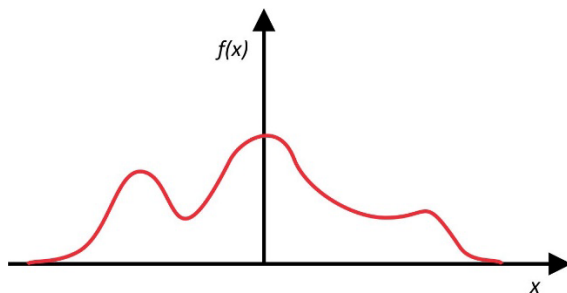
Suppose you need points to be uniformly distributed over the surface of a circle of radius one centered at the origin. It would not be easy to generate those points directly, but it is much easier to:

1. Generate the point $X = 2U_1 - 1$, $Y = 2U_2 - 1$ on the square centered at 0 where U_1 and U_2 are independent uniform variables on the unit interval.
2. If the point is inside the circle, keep it. Otherwise, redo step 1.
3. Stop when you reach the desired sample size.



Rejection for a density

Rejection may also be used to simulate a distribution from its density. Consider:



Simulating from a density can be interpreted as simulating uniform points under that curve. If we can find a distribution g and a constant a such that

$$f(x) \leq ag(x)$$

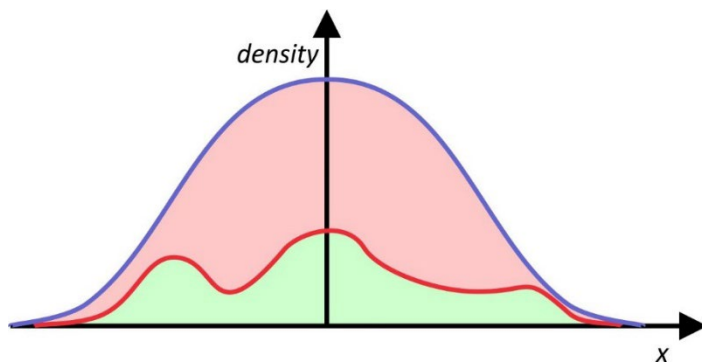
Then we can:

- Generate x from g ,
- Generate a uniform random variable u
- Keep x if $u \leq \frac{f(x)}{ag(x)}$, and reject it otherwise

In other words, we can place $f(x)/a$ underneath $g(x)$.

We generate points uniformly under $g(x)$ but accept them only if they are in the green zone – generating u is similar to deciding the height of the point.

Conditional on being kept, these points follow $f(x)$.



Markov Chain Monte Carlo (MCMC)

For density functions known up to a constant...

Consider the gamma distribution:

$$f(x) = \frac{\Gamma(\alpha)}{\beta^\alpha} x^{\alpha-1} e^{-\beta x}$$

- The part in orange contains x . It defines the **shape** of the function.
- The part in blue is a **constant**. It is the only value that will make $\int f(x)dx = 1$.

We sometimes know the **shape** of the density, but getting its **constant** would be hard.

This is very frequent in Bayesian statistics: posterior distributions are typically complex and known up to a constant.

A Markov chain is a sequence where each random variable depends only on the previous value (they are not independent values). Advanced results in probability have been used to design MCMC algorithms that produce Markov chains with the right stable distribution.

MCMC is in some way a rejection method:

We can generate candidates from density g , but we want to get f

1. Start with x_i
2. Generate a new candidate x^* (from g)
3. Generate a uniform value u .
4. If $u < \max \left\{ 1, \frac{f(x^*)g(x_i)}{g(x^*)f(x_i)} \right\}$ then $x_{i+1} = x^*$. Otherwise, we keep $x_{i+1} = x_i$.

The new values are accepted at just the right rate to be present the right proportion of times in the final chain.

R functions for generating random numbers

While it is interesting and important to know how random numbers are produced, there is already a wealth of functions in R to generate random numbers. Those include:

- `runif`: uniform
- `rnorm`: normal
- `rt`: Student
- `rgamma`: Gamma
- `rbinom`: binomial
- `rpois`: Poisson
- `mvrnorm`: multivariate normal
- `sample`: drawing with or without replacement

All these functions use multiple parameters to define the distribution and have the ability to generate many random numbers with one call.

Consult their documentation to identify which parameters change what.

Repetitions

The purpose of simulations is not to produce a single set of data, but to get the ability to generate many of them and observe how a method/algorithm/strategy can behave.

Consider this example with many draws of the lottery, and observe the frequency at which we have a certain number of winning numbers.

```
sample(1:49,6)           # Generates a winning number
draw=function(i){ sum(sample(1:49,6)<=6) }      # Generates one draw
# We can arbitrarily suppose the ticket had numbers 1 to 6
res=sapply(1:100000,draw)
table(as.factor(res))/100000
```

0	1	2	3	4	5
0.43485	0.41298	0.13345	0.01769	0.00102	0.00001

Having an idea of the chances of winning did not require mathematical formulas. We could instead generate many copies of the situation.

Repetitions are especially useful to generate variations in possible outcomes and assess risk.

Let's consider a different betting problem. I have \$20 and play the roulette \$1 at a time on a single number until I lose the \$20 dollars or exceed \$40. The roulette has 38 slots, only one of which will make me win \$35 if I get the number right.

We can generate 1,000 game trajectory. What are the interesting info we might want to know?

- The probability of losing the \$20?
- The expected amount that we have left after using this strategy?
- The distribution of the amount we get when we walk away winning?
- How many times do we play before reaching the stopping criterion?

How could we convey that information? Numbers, plots, etc.?

```
# Roulette
set.seed(13246)
onerep=function(i,money=20){
  n=0
  while(money>0 & money <40) {
    n=n+1
    if((runif(1)*38)>1) {money=money-1}
    else {money=money+35}
  }
  return(c(money,n))
}
out=sapply(1:1000,onerep)
```

```
# Probability of losing
mean(out[,]==0)
```

```
[1] 0.609
```

```
# Expected amount left
mean(out[,])
```

```
[1] 19.307
```

```
# Distribution of gains
```

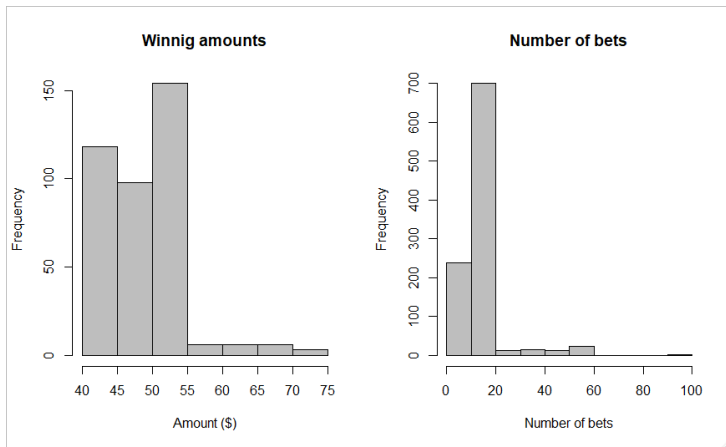
```
Par(mfrow=c(1,2))
```

```
hist(out[1,out[1,]>0],main="Winning amounts",col="Gray",  
      xlab="Amount ($)")
```

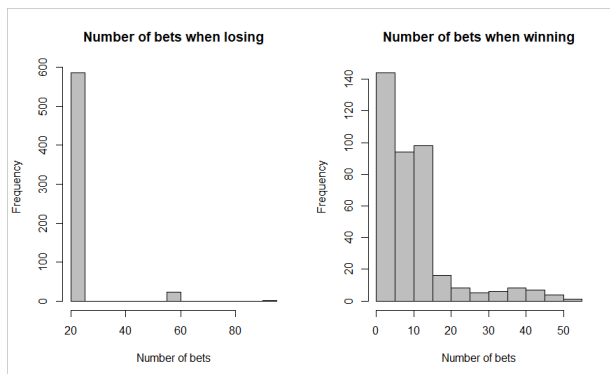
```
# Distribution of number of bets
```

```
hist(out[2,],main="Number of bets",col="Gray",xlab="Number of bets")  
mean(out[2,])
```

[1] 17.073




```
# Number of bets when winning or not
hist(out[2,out[1,]==0],main="Number of bets when losing",
     col="Gray",xlab="Number of bets")
hist(out[2,out[1,]>0],main="Number of bets when winning",
     col="Gray",xlab="Number of bets")
```



How many repetitions?

The number of repetitions is often arbitrary (500 or 1000 are most frequent).

It should be large enough to make the results stable: that the numbers reported do not vary much if you ran the simulation with another seed.

Simulating a model

Statistical models typically describe the mechanism that generates the data.

To simulate from such models, you simply need to reproduce that structure.

Linear regression

The linear regression model states that:

$$Y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \varepsilon$$

Where ε are independent normal random variables with mean 0 and variance σ^2 .

To simulate, you need:

- To decide on values for the covariates – they can come from an existing set, or be generated at your leisure.
- Decide the parameters: $\beta_0, \dots, \beta_p, \sigma^2$.

The choice of the parameters depends on what you want to investigate.

Consider the regression model:

$$Y = 2 + X_1 - 5X_2 + 3X_3 + 2X_4 + \varepsilon$$

Where X_5 is measured Y , and the variance of the error is $\sigma^2 = 4$.

A sample of size $n = 200$ may be generated this way:

```
library(MASS)
set.seed(23905)
n=200; p=5
Sigma=matrix(0.2,p,p)
diag(Sigma)=1
X=mvrnorm(n,rep(0,p),Sigma)
X=cbind(1,X)
beta=c(2,1,-5,3,2,0)

Y=X%%beta+rnorm(n,4)
summary(lm(Y~X[,-1]))
```

In a simulation, such samples would be generated multiple times to answer different questions such as: How often do we wrongly think that X_5 is significant? How good are we at predicting that $\beta_1 = 1$ (what is its RMSE)? Etc.

Call:

```
lm(formula = Y ~ X[, -1])
```

Residuals:

	Min	1Q	Median	3Q	Max
	-2.79167	-0.63450	-0.01327	0.59118	2.31499

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	5.95067	0.06495	91.616	<2e-16	***
X[, -1]1	0.95323	0.06910	13.794	<2e-16	***
X[, -1]2	-4.91106	0.07088	-69.285	<2e-16	***
X[, -1]3	2.88689	0.07401	39.007	<2e-16	***
X[, -1]4	1.88707	0.06785	27.811	<2e-16	***
X[, -1]5	0.15590	0.06594	2.364	0.0191	*

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.911 on 194 degrees of freedom


Multiple R-squared: 0.973, Adjusted R-squared: 0.9723

F-statistic: 1398 on 5 and 194 DF, p-value: < 2.2e-16

Logistic regression

The same strategy applies, but the response is drawn as a Bernoulli variable.

```
p=exp(X%*%beta) / (1+exp(X%*%beta))  
u=runif(n)  
Y2=as.numeric(u<p)  
summary(glm(Y2~X[,-1]), family="binomial")
```



[...]

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	0.64206	0.02130	30.141	< 2e-16	***
X[, -1]1	0.09042	0.02266	3.990	9.37e-05	***
X[, -1]2	-0.33331	0.02325	-14.338	< 2e-16	***
X[, -1]3	0.22069	0.02427	9.092	< 2e-16	***
X[, -1]4	0.10843	0.02225	4.873	2.28e-06	***
X[, -1]5	0.03476	0.02163	1.608	0.11	

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Generating data from machine learning methods

Contrarily to statistical models, machine learning methods are often based on algorithms that do not specify the structure of the data generating process. In that sense, it would be unusual to generate data “from” a tree or a random forest. However, we could generate data with a structure that will suit those methods well.

Strategy A: Define binary variables based on conditions on the existing covariates:

- $X_1 = 1$ if more than 1500 square feet
- $X_2 = 1$ if more than two bathrooms
- $X_3 = 1$ if more than 1500 square feet and lot over 5000 square foot
- *etc.*

Then simulate a regression model based on these variables.

Strategy B: Define conditions like above, then generate random variables with different means and/or variance for each of these groups.

Nonparametric bootstrap

Suppose that you need to know the variance of the median of 50 data points.

If you know the distribution of your data, a simulation could be used:

1. Generate 1,000 samples of 50 variables,
2. Compute the median on each sample,
3. Compute the variance of the 1000 estimated medians.

```
x=matrix(rnorm(50*1000),ncol=1000)
meds=apply(x,2,median)
var(meds)
```

The response obtained is random, but by making the number of copies arbitrarily large, the variation due to the simulation can be made negligible.

This is easy when you know the distribution of the data, but what if you do not?

Given a sample X_1, \dots, X_{50} of unknown distribution, how should we simulate data?

The empirical cumulative distribution function (CDF) of a sample of size n gives a weight of $1/n$ to each of its data points. One solution is, therefore, to simulate from the empirical distribution function. This is nonparametric bootstrap.

This means that new data are generated
by sampling with replacement from the data.

```
data(iris)
dat=iris %>%
  filter(Species=="setosa") %>%
  select(Petal.Length)
set.seed(45213)
x=matrix(sample(dat[[1]],50*1000,replace=TRUE),ncol=1000)
meds=apply(x,2,median)
var(meds)
[1] 0.002130808
```


Knowing the truth

In a simulation, you know the “**true model**”.

It means that you can explore the ability of a method to “get it right”.

Consider estimating a true parameter θ , that you estimate with $\hat{\theta}$.

You may evaluate:

- Bias: $E(\hat{\theta} - \theta)$
- Variance: $\text{var}(\hat{\theta})$
- MSE: $E\{(\hat{\theta} - \theta)^2\}$ or $RMSE = \sqrt{MSE}$

Consider a test of hypothesis: you may check that p-values are uniform and you reject H_0 as expected. You can even assess the power of your test under different scenarios.

Confidence intervals: you may evaluate if their coverage (the proportion of times that they contain the true value when they are used repetitively) is as described.

All because you know the “truth”!

Conditions

While in some cases, you want to just confirm that your method works, sometimes, simulations are used to test the limits:

- What if there is contamination (data that do not belong and may be extreme)
- What if the model is misspecified (some assumptions are violated)
- Does this asymptotic result work well on small finite samples?
- Can my method handle multicollinearity
- Etc.

The idea is not to attempt to test all the conditions but to investigate the relevant ones that a practitioner would need to know before he considers using the method.

Motivating examples

The handout will cover some examples of questions that are answered with a short simulation.

Designing a simulation

- Identify questions of interest
- Identify meaningful outcomes/results that can answer those questions
- Choose the parameters involved in the question of interest carefully to represent it well.
- Other parameters should have a reasonable value.
- You may use existing data as an inspiration to determine your parameters.
- Report your results in simple tables and plots that highlight your answers
- If you use multiple simulations, build a story: incrementally discover things about your method.

Paper about simulations

Morris, T. P., White, I. R., & Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in medicine*, 38(11), 2074-2102.

Worth reading, some wisdom and insight.

Example 1: BIRCH

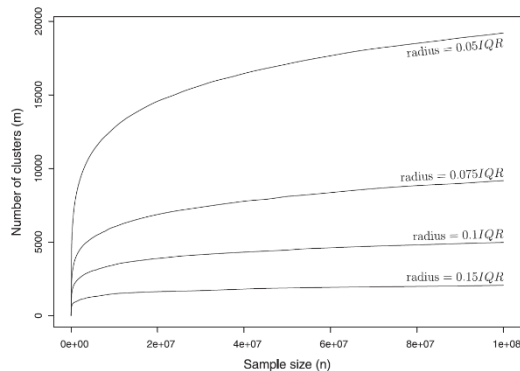
L. Charest & J.-F. Plante (2014). Using BIRCH to compute approximate rank statistics on massive datasets, *Journal of Statistical Computation and Simulation*, 84, 2214-2232.

An algorithm is designed to handle huge databases by saving only a limited number of summary statistics about the data read so far. We use it to evaluate rank correlation.

On the right, the memory requirements as the sample size increases.

In the table below, the relative performance of the BIRCH based algorithm vs. using a random sample of the data.

We describe the properties of the algorithm as well as its performance.



n	\bar{m}	MSE as RE		MSE as ESS		1000 \times bias	
		ρ_W	τ_W	ρ_W	τ_W	ρ_W	τ_W
10^6	2491	19.0	16.0	190,000	160,000	1.2	0.97
10^7	3867	2.8	2.5	280,000	250,000	1.1	0.85

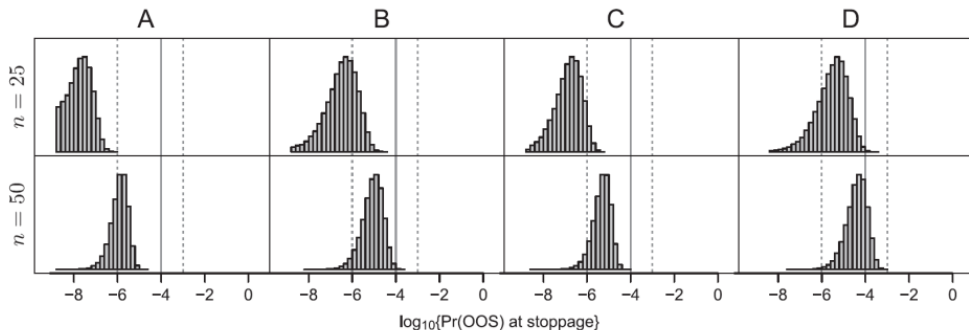
Example 2: Simulated quality

J.-F. Plante & G. Windfeldt (2012). Using predictive statistics for process control, *Australian and New Zealand Journal of Statistics*, 54, 485-504.

In classic quality control, any instability in a process results in a stoppage to fix it.

We propose a strategy where the probability of producing out-of-spec items is monitored. You may thus sometimes continue the production if the instability is not a big issue.

Under different scenarios of production, what is the risk for the producer (probability to stop when not needed) and for the consumer (probability of buying and out-of-spec item)?

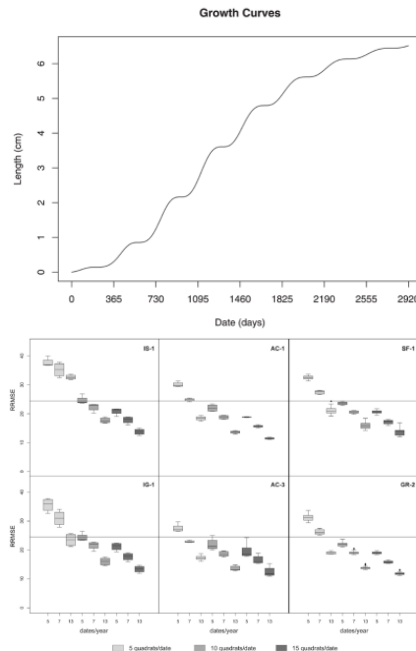


Example 3: *Moules(no)frites!*

M. Cusson, J.-F. Plante & C. Genest (2006). The effect of different sampling designs and methods on the estimation of secondary production: A simulation. *Limnology and Oceanography: Methods*, **4**, 38-48.

Many techniques and strategies (sample in winter or not, more frequently in growing season or not, etc.) are proposed to estimate the biomass of a population of mussels. Which of these methods is best? When should you go out and how many samples should you collect each time?

We generated the growth of a large population of mussels to compare (bias, variance, and RMSE) of the different methods and strategies. Getting realistic population parameters was key – we used real data and expert advice to get the parameters right.



Example 4: Genetic algorithms

M. Larocque, J.-F. Plante & M. Adès (2018). Bagged parallel genetic algorithms for objective model selection.

We propose an objective criterion to select which variables should be in a regression model based on a parallel genetic algorithm. Do we recover the right variables?

The boxplots represent the values from 1,000 repetitions of our algorithm on a regression model with three active features (in grey). The dashed red line shows our objective criterion to keep a feature. Gray boxplots show which variables are the true active features, and their importance is indeed larger.

