STAT 7650: Computational Statistics

1. Introduction

Peng Zeng

Department of Mathematics and Statistics
Auburn University

Spring 2026

Outline

Example: Horseshoe Crabs

Each female horseshoe crab has a male crab resident in her nest. Satellites mean other male crabs residing nearby.

- $Y = 1$ if at least one satellite; $Y = 0$ otherwise
- $X =$ the width of the female crab

The observations are $\{(y_i, x_i), y_i \in \{0, 1\}, x_i \in \mathbb{R}, i = 1, \ldots, n\}$. Consider a probit model,

$$\pi(x_i) = P(Y_i = 1 \mid X = x_i) = \Phi(\beta_0 + \beta_1 x_i)$$

where Φ is the CDF of $N(0, 1)$. The questions of interest include

- What are the values of β_0 and β_1?
- What are the standard errors of the estimates?
- Does Y depend on X, that is, $\beta_1 = 0$?

Statistical Models

Generally, a statistical problem includes

- Observations: $\{(y_i, x_i), i = 1, \ldots, n\}$
- Model: $(y_i, x_i) \sim^{iid} f(y \mid x, \theta)$ with unknown parameter θ

One can study the statistical problem from different perspectives.

- Theory
    - How to estimate θ? What is the sampling distribution?
    - How to conduct hypothesis testing?
    - Is the method optimal?
- Computation
    - Algorithm to compute the estimate, standard errors, etc
    - Implement the algorithm in software/package
- Application
    - Apply the method to real data, interpret the results

Focus and Prerequisite

This course, STAT 7650 Computational Statistics,

- mainly discuss computational algorithms and methods

Prerequisite

- Calculus and matrices.
- Basic knowledge of probability and statistics.
- Common statistical methods/models (e.g. maximum likelihood estimation, linear regression, ANOVA, etc)
- A programming language (e.g. R, python, or MatLab)

Maximum Likelihood Estimation

Let $y_1, \ldots, y_n \sim^{iid} f(y \mid \theta)$. The maximum likelihood estimate of θ is

$$\hat{\theta} = \arg\max_{\theta} \prod_{i=1}^{n} f(y_i \mid \theta).$$

Under mild conditions, when n is large, the MLE $\hat{\theta}$ follows normal,

$$\hat{\theta} \sim N(\theta, [nI(\theta)]^{-1}),$$

where the Fisher information matrix $nI(\theta)$ can be approximated by either $nI(\hat{\theta})$ or the observed information matrix $J_n(\hat{\theta})$,

$$I(\theta) = E\left(-\frac{\partial^2 \log f(y \mid \theta)}{\partial\theta\partial\theta^T}\right), \quad J_n(\theta) = -\sum_{i=1}^{n} \frac{\partial^2 \log f(y_i \mid \theta)}{\partial\theta\partial\theta^T}.$$

Work on the Log-Scale

It is generally preferable to work on the log scale when evaluating densities or likelihood functions, as this improves numerical stability.

Example: Let $y_i \in \{0, 1\}$ and $x_i \in \mathbb{R}^p$. Consider a probit model

$$P(y_i = 1) = \pi_i = \Phi(\beta^T x_i), \quad i = 1, \ldots, n,$$

where Φ denotes the CDF of $N(0, 1)$. The MLE of β is

$$\begin{aligned}\hat{\beta} &= \arg\max_{\beta} f(\beta) = \arg\max_{\beta} \prod_{i=1}^{n} \pi_i^{y_i}(1 - \pi_i)^{1-y_i} \\ &= \arg\max_{\beta} \sum_{i=1}^{n} \Big\{ y_i \log \pi_i + (1 - y_i) \log(1 - \pi_i) \Big\}.\end{aligned}$$

Hence, $\hat{\beta}$ is the root of $\partial \log f(\beta) / \partial\beta = 0$.

Example, Continued

Evaluating the following is numerically unstable for large values of x,

$$\phi(x)/[1-\Phi(x)]$$

where ϕ and Φ denote the density and CDF of $N(0,1)$.

```
dnorm(x) / (1 - pnorm(x)) # 2.373216 when x = 2; NaN when x = 80
exp(dnorm(x, log = T) - pnorm(x, lower = F, log = T))  # 80.0125
```

When x is extremely large, even computations on the log scale may fail. In such cases, use the approximation $\phi(x)/[1-\Phi(x)] \approx x$, which is justified by Gordon's inequality:

$$x \le \frac{\phi(x)}{1-\Phi(x)} \le x + \frac{1}{x}, \qquad \text{for } x > 0.$$

Question: How about $\phi(x)/\Phi(x)$ when x is extremely small?

R Functions for Optimization

R provides two functions for general-purpose optimization.

- For univariate minimization

```
optimize(f, interval, ...,
         lower = min(interval), upper = max(interval),
         maximum = FALSE,
         tol = .Machine$double.eps^0.25)
```

- For multivariate minimization,

```
optim(par, fn, gr = NULL, ...,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B",
                 "SANN", "Brent"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)
```

Set `control = list(fnscale = -1)` for maximization.

Statistical Learning

The observations are $\{(y_i, x_i), i = 1, \ldots, n\}$. Let $\tilde{y} = g(x \mid \theta)$ be a prediction of y, where θ is unknown. Then

$$\hat{\theta} = \min_{\theta} \sum_{i=1}^{n} L(y_i, g(x_i \mid \theta))$$

where L is a loss function

- least squares: $L(y, \tilde{y}) = \|y - \tilde{y}\|_2^2$
- least absolute deviation: $L(y, \tilde{y}) = |y - \tilde{y}|$
- misclassification error: $L(y, \tilde{y}) = I(y \neq \tilde{y})$
- negative log-likelihood:

Bayesian Inference

Let $y_1, \ldots, y_n \sim^{iid} f(y \mid \theta)$ and the prior distribution of θ be $p(\theta)$. Then the posterior distribution of θ is

$$f(\theta \mid y_1, \ldots, y_n) = \frac{f(y_1, \ldots, y_n \mid \theta)p(\theta)}{f(y_1, \ldots, y_n)} \propto \prod_{i=1}^{n} f(y_i \mid \theta)p(\theta)$$

If we can generate samples $\theta^1, \ldots, \theta^m$ from $f(\theta \mid y_1, \ldots, y_n)$, the population characteristics of θ can be approximated by their sample counterparts,

$$E[g(\theta) \mid y] \approx \frac{1}{m}\sum_{j=1}^{m} g(\theta^j), \qquad P(\theta < a \mid y) \approx \frac{1}{m}\sum_{j=1}^{m} I(\theta^j < a)$$

Topics

Givens and Hoeting (2013). *Computational Statistics*. Wiley.

- Numerical linear algebra
- Optimization and solving nonlinear equations
  - univariate and multivariate problems
  - combinatorial optimization
  - EM algorithm
- Integration and simulation
  - numerical integration
  - simulation and Monte Carlo integration
  - Markov chain Monte Carlo (MCMC)
  - Hamiltonian Monte Carlo (HMC)
- Selected topics
  - bootstrap
  - deep learning (neural network, XGBoost)

Top 10 Algorithms

The January/February 2000 issue of *Computing in Science & Engineering* discussed the Top 10 Algorithms of the 20th century.

- Metroplis algorithm for Monte Carlo
- simplex method for linear programming
- Krylov subspace iteration methods
- the decompositional approach to matrix computations
- the Fortran optimizing compiler
- QR algorithm for computing eigenvalues
- quicksort algorithm
- fast Fourier transform
- integer relation detection
- fast multipole method

Computer Numbers $\neq$ Real Numbers

True or False?

```
1 + 1 + 1 == 3
0.1 + 0.1 + 0.1 == 0.3
0.1 + 0.1 + 0.1 > 0.3
```

```
print(0.1, digits = 20)
print(0.3, digits = 20)
```

Will the code stop?

```
x = 9.007199254740992e+15
while(x + 1 != x)  x = x + 1        # x = 9007199254740992 = 2^53
```

True or False?

```
(1e+20) + 1 == (1e+20)
```

Integers - Fixed-Point Numbers

For a 32-bit integer,

- the first bit: sign, 0 for positive and 1 for negative
- the remaining 31-bits: integer in binary representation

```
log2( .Machine$integer.max + 1)  # = 31
2147483647L + 1L                 # NA with warning message
2147483647L + 1                  # 2147483648
```

The range of integers is $[-2^{31}, 2^{31}-1]$. Overflow occurs if the arithmetic operation leads to a result beyond the range.

Real Numbers - Floating-Point Numbers

In a 64-bits system,

- 1 bit for sign
- 52 bits for significand
- 11 bits for exponent

The range of values is roughly between 10^{-318} to 10^{318} with 16 decimal places of precision.

- The computer number of a real number x is the nearest floating-point number that can be represented by the computer.
- Floating-point numbers are not uniform within its range. There are same number of points in $[2^i, 2^{i+1}]$ as in $[2^{i+1}, 2^{i+2}]$.

Machine Epsilon

$\varepsilon_{\min}$ and $\varepsilon_{\max}$ are the distances between 1 and the compute numbers immediately smaller and larger than 1, respectively.

```
.Machine$double.eps           # 2.220446e-16
.Machine$double.neg.eps       # 1.110223e-16

1 + .Machine$double.eps * 0.5 == 1              # TRUE
1 + .Machine$double.eps * 0.5000001 == 1        # FALSE

1 - .Machine$double.neg.eps * 0.5 == 1          # TRUE
1 - .Machine$double.neg.eps * 0.500001 == 1     # FALSE

2 / .Machine$double.eps       # 9.007199e+15
```

Tolerance

We may treat $X = Y$ if for a small positive value $\delta > 0$,

- the absolute tolerance

$$|X - Y| < \delta.$$

- the relative tolerance

$$\frac{|X - Y|}{\min\{|X|, |Y|\}} < \delta.$$

Some choices of δ,

- a small number such as 10^{-4}, 10^{-6}, 10^{-8}, or others
- `sqrt(.Machine$double.eps)` which is `1.490116e-08`
- Rarely choose `.Machine$double.eps`

Overflow and Underflow

- Underflow occurs when numbers near zero are rounded to zero.
- Overflow occurs when numbers with large magnitude are approximated as ∞ or $-\infty$.

For example, calculate

$$\frac{e^a}{e^a + e^a} = \frac{1}{2}$$

Underflow occurs when a approaches $-\infty$ and overflow occurs when a approaches ∞.

```
exp(-1e+10) / (exp(-1e+10) + exp(-1e+10))    # NaN, underflow
exp(1e+10) / (exp(1e+10) + exp(1e+10))       # NaN, overflow
```

Computing Softmax

We can computing softmax as follows,

$$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - m}}{\sum_j e^{x_j - m}}, \quad m = \max_j x_j$$

```
x = c(1000, 1001, 999)
exp(x) / sum(exp(x))       # NaN NaN NaN

softmax = function(x) {
  m = max(x)
  exp(x - m) / sum(exp(x - m))
}

softmax(x)          # 0.24472847 0.66524096 0.09003057
```

Functions

The `matrixStats` package provides some useful functions,

- `logSumExp()` compute $\log(e^a + e^b)$
- `log1pexp()` compute $\log(1 + e^x)$

```
library(matrixStats)

ls = logSumExp(x)
exp(x - ls)              # 0.24472847 0.66524096 0.09003057
```

Notice that

$$\frac{e^{x_i}}{\sum_j e^{x_j}} = \exp\left(x_i - \log\left(\sum_j e^{x_j}\right)\right)$$

Calculation When Close to Zero

Compute $1 - \cos(x)$ for small x,

$$1 - \cos(x) = 2\sin^2(x/2)$$

```
x = 1e-8
a = 1 - cos(x)                          # = 0
b = 2 * sin(x/2) * sin(x/2)             # = 5e-17

library(Rmpfr)
y = mpfr("1e-8", precBits = 200)        # 200-bit precision
m = 1 - cos(y)
n = 2 * sin(y/2) * sin(y/2)
```

R provides some built-in functions that are accurate for small x.

- `log1p()` compute $\log(1 + x)$
- `expm1()` compute $e^x - 1$

Solve Quadratic Equations

The roots for quadratic equation $ax^2 + bx + c = 0$ are

$$x = -\frac{b \pm \sqrt{b^2 - 4ac}}{2a}$$

It is better to compute the roots as

$$x_1 = -\frac{b + \text{sign}(b)\sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{c}{ax_1}$$

```
a = 1; b = -1e+6; d = 1;
delta = b * b - 4 * a * d;
x  = -(b + c(-1, 1) * sqrt(delta)) / (2 * a)
y1 = -(b + sign(b) * sqrt(delta)) / (2 * a)
y  = c(y1, d / (a * y1))

a * x^2 + b * x + d      # = 0.000000e+00 -7.614492e-06
a * y^2 + b * y + d      # = 0 0
```

Some Comments on Writing Codes

- Proper comments in your codes.
- Good programming style with indentation, space, alignment, etc.
- Vectorization in your codes.
- Avoid loop as much as possible.

Make your code

- workable
- efficient and fast
- error handling

Vectorization

Most R functions are designed to handle vectors as well as scalars. Avoid loops as much as possible in your R codes.

```
x = c(1, 2, 3) + c(1, 2, 3)        # good
for(i in 1:3) x[i] = i + i         # bad
```

Calculate the following

$$\sum_{i=1}^{1000} \sin\left(\frac{i\pi}{1000}\right)$$

```
x = 0;
for(i in 1:1000) x = x + sin(i * pi / 1000)       # bad
y = sum(sin((1:1000) * pi / 1000))                # good
```

Example: Kernel Density Estimate

Suppose that $x_1, \ldots, x_n$ are iid with density $f(x)$. The kernel density estimate at x_0 is

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\Big(\frac{x_0 - x_i}{h}\Big),$$

where $h > 0$ is called bandwidth and K is the kernel, which is usually selected as a density function.

```
## x0 is a scalar
KDEv0 = function(x, h, x0)
{
  n = length(x); yhat = 0.0;
  for(i in 1:n) yhat = yhat + dnorm(x[i] - x0, sd = h);
  yhat / n;
}
```

```
## x0 is a scalar
KDEv1 = function(x, h, x0)
{
  mean(dnorm(x - x0, sd = h));
}
```

```
## x0 is a vector
KDEv2 = function(x, h, x0)
{
  colMeans( dnorm(outer(x, x0, "-"), sd = h) );
}
```

Functions in R

- `colMeans()`, `colSums()`, `rowMeans()`, `rowSums()`
- `sweep()`
- `apply()`
- `sapply()`
- `aggregate()`

Using an apply function is not really vectorization. apply functions are actually wrappers for for loops.

```
x = matrix(rnorm(10 * 5), nrow = 10)
x + (1:10)                    # add 1:10 to each column
sweep(x, 2, 1:5, "+")         # add 1:5 to each row
apply(x, 2, sd)               # apply sd() to each column
```

Pre-allocating Memory

R is bad at continually re-sizing objects, because it makes an extra copy of these objects each time. If you have a loop that creates a vector or list, don't append to the vector or list. Instead, make an empty object of the correct size first, then fill in its elements.

```
# first approach.
alist = list();
for(idx in 1:100) alist = append(alist, idx);

# second approach
alist = list();
for(idx in 1:100) alist[[length(alist) + 1]] = idx;

# third approach
alist = vector("list", 100);
for(idx in 1:100) alist[[idx]] = idx;
```

How to Make R Faster

Some easy ways to speed up R:

- Get a better computer
- HPC (high performance computing)

More ways to speed up R:

- Vectorization
- Better algorithms
- Parallel computing
- R interface to C/Fortran