R 中的统计模拟

wang

2019-08-06

献给……

呃，爱谁谁吧

目录

表格

插图

前言

关于 R 的基础语法, 可以在网上或者书籍中学习.

Github[1]

W3Cschool[2]

Advanced R[3]

这里只是总结一些统计模拟中遇到的问题, 以及实用的技巧.

致谢

这个页面的建立基于 **knitr** (Xie, 2015) 和 **bookdown** (Xie, 2019)。以下是我的 R 进程信息:

```
sessionInfo()
```

```
## R version 3.6.0 (2019-04-26)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS Mojave 10.14.6
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/lib
```

[1] https://github.com/yanping/r-spring-camp/blob/master/1-introduction.md

[2] https://www.w3cschool.cn/r/r_overview.html

[3] https://adv-r.hadley.nz

```
## LAPACK: /Library/Frameworks/R.framework/Versions/3.6/Resources/lib/libRlapa
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils
## [6] datasets  methods   base
##
## other attached packages:
## [1] doParallel_1.0.14 iterators_1.0.10
## [3] foreach_1.4.4     MASS_7.3-51.4
## [5] quadprog_1.5-7
##
## loaded via a namespace (and not attached):
##  [1] beeswarm_0.2.3     tidyselect_0.2.5
##  [3] xfun_0.7           Rook_1.1-1
##  [5] purrr_0.3.2        colorspace_1.4-1
##  [7] vctrs_0.2.0        htmltools_0.3.6
##  [9] viridisLite_0.3.0  yaml_2.2.0
## [11] XML_3.98-1.20      rlang_0.4.0
## [13] pillar_1.4.2       glue_1.3.1
## [15] RColorBrewer_1.1-2 plyr_1.8.4
## [17] stringr_1.4.0      munsell_0.5.0
## [19] gtable_0.3.0       visNetwork_2.0.7
## [21] htmlwidgets_1.3    bench_1.0.2.9000
## [23] codetools_0.2-16   evaluate_0.14
## [25] knitr_1.23         vipor_0.4.5
## [27] DiagrammeR_1.0.1   profmem_0.5.0
## [29] highr_0.8          Rcpp_1.0.1
## [31] xtable_1.8-4       readr_1.3.1
## [33] backports_1.1.4    scales_1.0.0
## [35] jsonlite_1.6       rgexf_0.15.3
## [37] gridExtra_2.3      brew_1.0-6
## [39] ggplot2_3.1.1      hms_0.5.0
```

```
## [41] digest_0.6.19      stringi_1.4.3
## [43] bookdown_0.11      dplyr_0.8.1
## [45] grid_3.6.0         influenceR_0.1.0
## [47] tools_3.6.0        magrittr_1.5
## [49] lazyeval_0.2.2     tibble_2.1.3
## [51] crayon_1.3.4       tidyr_0.8.3
## [53] pkgconfig_2.0.2    zeallot_0.1.0
## [55] downloader_0.4     ggbeeswarm_0.6.0
## [57] assertthat_0.2.1   rmarkdown_1.13
## [59] rstudioapi_0.10    viridis_0.5.1
## [61] R6_2.4.0           igraph_1.2.4.1
## [63] compiler_3.6.0
```

作者简介

统计学学生.

主要用 R 和 tex.

第一章　常用函数及常见错误

1.1　产生数据

在进行模拟时, 我们经常会需要生成数据. 这里以正态分布为例, 说明如何产生数据. 在 R 中, 每种分布都会有以下 4 个函数:

- 概率密度函数: dnorm(x, mean = 0, sd = 1, log = FALSE)
- 累计分布函数: pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
- 分位数函数: qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
- 随机数产生: rnorm(n, mean = 0, sd = 1):

其中前 3 个函数都支持向量输入, 即计算一组取值的概率密度、累计分布、分位数. 最后一个函数常用来生成数据,n 即产生数据的个数. 如果需要生成多维正态分布, 需要调用 MASS 包中的 *mvrnorm(n = 1, mu, Sigma, tol = 1e-6, empirical = FALSE, EISPACK = FALSE), 其中 Sigma 是指定的协方差矩阵.

1.2　定义运算符

空间模型 (SAR) 中, 会出现对角块矩阵

$$\mathbf{W} = \begin{pmatrix} \mathbf{M} & & & \\ & \mathbf{M} & & \\ & & \ddots & \\ & & & \mathbf{M} \end{pmatrix}$$

M 作为权重矩阵, 这种形式的矩阵可以利用克罗内克积 (Kronecker product, 符号为 $\otimes$) 在 R 中很方便的产生, 命令是%x%. 可以写成

$$\mathbf{W} = \mathbf{I} \otimes \mathbf{M}.$$

```
diag(3)%x%matrix(1:6,2,3)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    3    5    0    0    0    0    0    0
## [2,]    2    4    6    0    0    0    0    0    0
## [3,]    0    0    0    1    3    5    0    0    0
## [4,]    0    0    0    2    4    6    0    0    0
## [5,]    0    0    0    0    0    0    1    3    5
## [6,]    0    0    0    0    0    0    2    4    6
```

这是借助自定义运算符实现的. 自定义运算符是一种特殊的函数, 当参数只有两个变量时, 可以进行定义. 用法如下:

```
'%myop%'<-function(a,b){a^b+b^a}
2%myop%3
```

```
## [1] 17
```

利用自定义运算符, 可以实现很方便的功能.R 中矩阵乘法 (%*%)、Kronecker 乘积 (%x%) 都是这样实现的. 另外还有整除 (%/%) 和取余 (%%)

```
9%/%4
```

```
## [1] 2
```

```
13%%3
```

```
## [1] 1
```

1.3 自动纠错

当我们输入的命令不规范时,R 会自动纠正, 以保证程序正常运行.

比如看下面的例子:

```
1:4 - 1:2
```

```
## [1] 0 0 2 2
```

```
1:5-1:2
```

```
## Warning in 1:5 - 1:2: longer object length is not a
## multiple of shorter object length
```

```
## [1] 0 0 2 2 4
```

当运算的向量长度不一致时,R 会自动重复短的向量, 使之长度与另外的向量长度相同进行运算. 但是当长度是整数倍当时候, 不会有任何提示.

再看下面当例子:

```
matrix(1:9,3,3)*(1:3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    4   10   16
## [3,]    9   18   27
```

本来想计算矩阵与向量的乘积, 但是错误使用了 *, 未使用矩阵乘法%*%,R 也可以计算返回一个矩阵, 不会有 warning.

当条件为向量是, 只会判断第一个值:

```
if( 1 <= 1:3) print("真")  else print("假")
```

```
## Warning in if (1 <= 1:3) print("真") else print("假"):
```

```
## the condition has length > 1 and only the first element
## will be used
```

```
## [1] "真"
```

如果条件为向量, 应该使用 all 或者 any 函数:

```
all(1 <= 1:3)
```

```
## [1] TRUE
```

```
all(2 <= 1:3)
```

```
## [1] FALSE
```

所有都真的时候返回 TRUE.

```
any(4 <= 1:3)
```

```
## [1] FALSE
```

```
any(2 <= 1:3)
```

```
## [1] TRUE
```

只要有一个为真就返回 TRUE.

1.4 predict 函数

当我们拟合好一个模型时, 下一步要做的就是评价模型好坏或者对新数据预测. 这都需要将新的输入值带入模型中计算, 得到预测值. 区别只是有没有真值对比. 对于简单模型, 我们当然可以直接提取系数, 自己计算预测值, 但是当模型复杂时 (比如时间序列模型), 就不太容易操作.

R 借助泛型函数[1], 编写模型都会提供 summary、predict、plot 等函数方便调用. 但是在学习过程中发现经常会不小心错误使用, 特此单独说明一下. 下面以线性模型为例, 先看正确的使用方法:

```
n=100;p=3;beta=c(1,2,3);
X = matrix(rnorm(n*p),n,p)
Y = X%*%beta + rnorm(n)
trainlist = sample(1:n,70)
regData = data.frame(Y,X)
fitmodel = lm(Y~.,data=regData[trainlist,])
pe = predict(fitmodel,newdata=regData[-trainlist,])
sum((pe-regData[-trainlist,1])^2)/length(pe)
```

```
## [1] 1.133
```

关键点:

所有数据存在一个数据框中 通过下标控制训练集和测试集的数据

错误程序 1:

```
n=100;p=3;beta=c(1,2,3);
X = matrix(rnorm(n*p),n,p)
Y = X%*%beta + rnorm(n)
fitmodel = lm(Y~X)
X2 = matrix(rnorm(n*p),n,p)
Y2 = X2%*%beta + rnorm(n)
#predict(fitmodel,newdata = X2)数据格式错误,不能执行
pe = predict(fitmodel,newdata = data.frame(X2))
sum((pe-Y2)^2)/length(Y2)
```

```
## [1] 28.09
```

这个程序能明显看出问题, 误差不应该这么大, 但是程序没有任何 warning.

[1]程序在 这里[2] 查看, 其中最后被/***R */夹住的部分是 R 程序, 每次加载后会自动执行, 方便调试.

错误程序 2:

```
n=100;p=3;beta=c(1,2,3);
X = matrix(rnorm(n*p),n,p)
Y = X%*%beta + rnorm(n)
fitmodel = lm(Y~X)
X2 = matrix(rnorm(0.2*n*p),0.2*n,p)
Y2 = X2%*%beta + rnorm(0.2*n)
pe = predict(fitmodel,newdata = data.frame(X2))
```

```
## Warning: 'newdata' had 20 rows but variables found have
## 100 rows
```

```
length(pe)
```

```
## [1] 100
```

修改测试数据的条数, 使之与训练集数据量不同, 可以发现 warning. 提示我们数据行数不一样. 并且我们测试集合 X2 是 20 行, 但是预测值 pe 返回的是 100 个值. 问题在于 predict 函数使用不正确.

我们用 all 指令查看:

```
all(predict(fitmodel)==predict(fitmodel,newdata = data.frame(X2)))
```

```
## Warning: 'newdata' had 20 rows but variables found have
## 100 rows
```

```
## [1] TRUE
```

就是说我们输入的参数没起到作用.

查看函数说明[3]

[3]垃圾回收（英语：Garbage Collection，缩写为 GC），在计算机科学中是一种自动的存储器管理机制。当一个计算机上的动态存储器不再需要时，就应该予以释放，以让出存储器，这种存储器资源管理，称为垃圾回收。

```
?predict.lm
```

参数	说明
object	Object of class inheriting from “lm”
newdata	An optional data frame in which to look for variables with which to predict. If omitted, the fitted values are used.
...	

newdata 不是必要的参数, 当缺失时候会使用拟合模型的数据. 至于为什么输入的数据不能正确识别, 因为名字不一样.R 中参数的传递都是通过名字, 我们在拟合 lm 时候, 解释变量的名字叫 ‘X’, 所以传入 ‘X2’ 不会识别到. 如果把 ‘X2’ 改名成 ‘X’, 即可正确预测[4], 比如看下面的程序:

```
n=100;p=3;beta=c(1,2,3);
X = matrix(rnorm(n*p),n,p)
Y = X%*%beta + rnorm(n)
fitmodel = lm(Y~X)
X = matrix(rnorm(0.2*n*p),0.2*n,p)
Y2 = X%*%beta + rnorm(0.2*n)
pe = predict(fitmodel,newdata = data.frame(X))
sum((Y2-pe)^2)/length(pe)
```

```
## [1] 0.5423
```

通过以上例子, 想说明当程序的结果和预期不一致时. 当然可能是我们的方法不对, 但是也有可能是调用函数的方式出了问题. 比如线性规划求解的 lp 函数, 默认在正半轴求解[5].

[4] 后续重新整理成 Rmd 格式.

[5] 省略了常数系数.

1.5 保存结果

我们重复了 M 次实验 (运行几小时或者几天), 计算了几个指标. 但是后续通过阅读其他文献或者老师的建议, 需要计算一个新的指标. 这时候, 就可以打开之前保存好的运行结果, 只需要进行分析画图即可, 不需要重新运行程序[6].

```
getwd()
setwd()
save.image()
```

getwd,setwd 分别用于获取、设置当前的工作目录. 有时候我们保存了结果, 但是不知道存到哪里, 可以通过 getwd 查看当前的工作目录. 当然, 最好是在程序执行前, 手动设置好工作目录.

save.image 用于保存工作空间, 可以借助对于字符串操作函数 paste 等设置文件名.

1.6 结果输出

R 只是我们模拟使用的工具, 模拟结果需要以图表的形式在文章中展现. 对于图片, 只要单独保存成文件, 在文章中插入即可. 对于表格, 如果不借助工具, 会很耗时. 这里我们通过 xtable 包中的函数, 可以将表格数据转换成 LaTeX 内的表格形式.

这里顺便介绍一下 R 中 package 的安装和加载. 不管是在 R GUI 中, 还是在 Rstudio 中, 都可以通过点选菜单进行安装. 如果通过命令安装, 如下

```
install.packages("xtable")
```

安装后, 并不能直接使用. 需要通过 library 命令加载后才能使用. 一个 package 中包含很多函数和数据, 有时候我们只需要使用其中的一个函

[6] 目前只有单次二项分布的程序正确. 完整的程序可以从这里[7]下载.

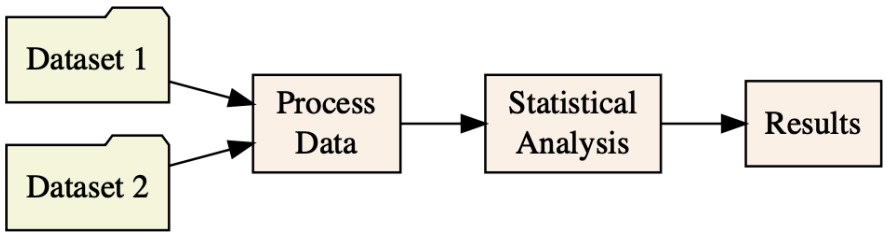
数, 不需要加载整个包. 这时候可以通过下面这样, 直接调用某个包中的函数,xtable 有一些参数可以设置输出的格式, 比如下面指定了小数部分位数:

```
xtable::xtable(matrix(rnorm(15),3,5),digits=5)
```

```
## % latex table generated in R 3.6.0 by xtable 1.8-4 package
## % Tue Aug  6 09:01:36 2019
## \begin{table}[ht]
## \centering
## \begin{tabular}{rrrrrr}
##   \hline
##  & 1 & 2 & 3 & 4 & 5 \\
##   \hline
## 1 & 1.42443 & -1.38918 & 1.66400 & -0.19721 & 0.28056 \\
##   2 & 0.02490 & 0.82681 & -0.27839 & -0.98789 & -0.06587 \\
##   3 & -0.23097 & 0.30374 & 0.29035 & 0.08272 & -0.00393 \\
##    \hline
## \end{tabular}
## \end{table}
```

这只是最基础的表格,LaTeX 中定制表头[8]可以在这查看.

1.7 模拟流程



[8] https://github.com/Ri0016/table-update-tex

第二章　参数估计

对于参数估计的模拟, 主要分成三步:

1. 生成数据;
2. 估计参数;
3. 评价估计好坏.

对于生成数据, 在阅读文献时, 文章中会明确给出如何产生. 我们不必多费周折. 在设计我们自己的模拟实验时, 多借鉴其他文献中的例子. 一方面方便和其他文献进行比较, 另一方面, 可以避免落入陷阱[1].

对于评价估计好坏, 有限维离散参数一般采取 $\left\|\hat{\boldsymbol{\theta}}-\boldsymbol{\theta}_0\right\|$, 无穷维函数一般采取 $\int\left(\hat{m}(x)-m_0(x)\right)^2 dx$.

估计量通过解估计方程得到, 根据方程的形式, 分为 M 估计量和 Z 估计量, 下面分别说明.

2.1　M 估计

定义: 极大化 (或极小化) 目标函数得到参数估计值.

$$M_n(\theta)=\frac{1}{n}\sum_{i=1}^{n} m_\theta\left(X_i\right)$$

$$\hat{\theta}=\arg\max_{\theta\in\Theta} M_n(\theta)$$

[1] 程序在 这里[2] 查看, 其中最后被/***R */夹住的部分是 R 程序, 每次加载后会自动执行, 方便调试.

其中 $m_\theta(X_i)$ 为已知函数. 特别的, 如果 $M_n(\theta)$ 可导,M 估计量和 Z 估计量有等价形式.

下面以线性模型为例:$\boldsymbol{Y} = \boldsymbol{X}^T\boldsymbol{\theta} + \boldsymbol{\varepsilon}$

考虑最小二乘估计, 取 $m_\theta(\boldsymbol{X}_i) = -\left(Y_i - \boldsymbol{X}_i^T\boldsymbol{\theta}\right)^2$[3], 则

$$\begin{aligned} M_n(\theta) &= -\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \boldsymbol{X}_i^T\boldsymbol{\theta}\right)^2 \\ &= -\frac{1}{n}\left(\boldsymbol{Y} - \boldsymbol{X}^T\boldsymbol{\theta}\right)^T\left(\boldsymbol{Y} - \boldsymbol{X}^T\boldsymbol{\theta}\right) \\ &= -\frac{1}{n}\left(\boldsymbol{\theta}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{\theta} - 2\boldsymbol{Y}^T\boldsymbol{X}^T\boldsymbol{\theta} + \boldsymbol{Y}^T\boldsymbol{Y}\right) \end{aligned}$$

其中最后一项是与 $\boldsymbol{\theta}$ 无关的常数项, 可以不考虑, 进而可以整理成如下的二次规划问题:

利用 quadprog 包中的函数可以求解

```
library(quadprog)
n=100;p=3;beta=c(1,-2,3);
X = matrix(rnorm(n*p),n,p)
Y = X%*%beta + rnorm(n)
lm(Y~X+0)$coef == solve.QP(Dmat = t(X)%*%X/n, dvec = (t(Y)%*%X)/n, Amat = matr
```

```
##    X1    X2    X3
## FALSE FALSE  TRUE
```

可以看到结果显示不相等, 但是如果我们打印出来显示:

```
lm(Y~X+0)$coef
```

```
##       X1       X2       X3
##   0.8925 -2.1106   3.0365
```

[3]垃圾回收（英语：Garbage Collection，缩写为 GC），在计算机科学中是一种自动的存储器管理机制。当一个计算机上的动态存储器不再需要时，就应该予以释放，以让出存储器，这种存储器资源管理，称为垃圾回收。

```
solve.QP(Dmat = t(X)%*%X/n, dvec = (t(Y)%*%X)/n, Amat = matrix(0,p,p))$so
```

```
## [1]  0.8925 -2.1106  3.0365
```

可以看到结果是一致的. 这是由于计算机存储数字精度引起的. 比如我们查看

```
sum(abs(lm(Y~X+0)$coef -
solve.QP(Dmat = t(X)%*%X/n, dvec = (t(Y)%*%X)/n, Amat = matrix(0,p,p))$so
```

```
## [1] 1.998e-15
```

另外, 我们可以用 all.equal 这个函数设置容忍的误差值, 判断近似相等:

```
all.equal(unname(lm(Y~X+0)$coef),
solve.QP(Dmat = t(X)%*%X/n, dvec = (t(Y)%*%X)/n, Amat = matrix(0,p,p))$so
,tolerance=1e-10)
```

```
## [1] TRUE
```

```
all.equal(unname(lm(Y~X+0)$coef),
solve.QP(Dmat = t(X)%*%X/n, dvec = (t(Y)%*%X)/n, Amat = matrix(0,p,p))$so
,tolerance=1e-20)
```

```
## [1] "Mean relative difference: 6.654e-16"
```

下面考虑稍复杂的情况, 取 $m_\theta(\boldsymbol{X}_i) = \rho_\tau\left(y_i - \boldsymbol{X}_i^T\boldsymbol{\theta}\right)$, 其中 $\rho_\tau(t) = t\left(\tau - I_{\{t<0\}}\right)$. 这称为分位数回归, 详细的推导过程可以在分位数回归总结[4]查看[5].

这里缺少一个 M 估计迭代求解的例子

[4] https://github.com/dujiangbjut/dujiangbjut.github.io/tree/master/ /

[5] 后续重新整理成 Rmd 格式.

2.2 Z 估计

定义: 解一个等于 0 的方程得到参数估计.

$$\Psi_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\psi_\theta\left(X_i\right) = 0,$$

其中 $\psi_\theta\left(X_i\right)$ 为已知函数.

$\hat{\theta}$ 为 $\Psi_n(\theta) = 0$ 的解.

回到线性模型最小二乘的例子, 由于目标函数存在导数, 可以转化成一个 Z 估计量. 取 $\psi_\theta\left(X_i\right) = m'_\theta\left(X_i\right) = 2\boldsymbol{X}_i^T\left(Y_i - \boldsymbol{X}_i^T\boldsymbol{\theta}\right)$

则[6]

$$\Psi_n(\theta) = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{X}_i^T\left(Y_i - \boldsymbol{X}_i^T\boldsymbol{\theta}\right) = \boldsymbol{X}\boldsymbol{X}^T\boldsymbol{\theta} - \boldsymbol{X}\boldsymbol{Y} = 0,$$

进而得到参数估计值为:$\hat{\boldsymbol{\theta}} = \left(\boldsymbol{X}\boldsymbol{X}^T\right)^{-1}\boldsymbol{X}\boldsymbol{Y}$.

通过下面的程序验证:

```
library(quadprog)
all.equal(solve.QP(Dmat = t(X)%*%X/n, dvec = (t(Y)%*%X)/n, Amat = matrix(0,p,p
```

```
## [1] TRUE
```

下面是一个迭代求解 Z 估计量的例子[7].

补充哪一篇参考文献

```
source("code/glm.R")
n=500
m1=2
m2=2
m3=2
m=m1+m2+m3
```

[6] 省略了常数系数.

[7] 目前只有单次二项分布的程序正确. 完整的程序可以从这里[8]下载.

```r
beta=c(rep(-1,m1),rep(0,m2),rep(1,m3))
X=runif(n*m,-1,1)
X=matrix(X,n,m)
eta=X%*%beta
mu=1/(1+exp(-eta))

# Y~B(1,p)
Y=runif(n)
Y[Y>=mu]=0
Y[Y>0]=1
glm(Y~X+0,family=binomial(link="logit"))
```

```
##
## Call:  glm(formula = Y ~ X + 0, family = binomial(link = "logit"))
##
## Coefficients:
##      X1       X2       X3       X4       X5       X6
## -1.1039  -1.0538   0.1477  -0.0958   0.7766   1.1833
##
## Degrees of Freedom: 500 Total (i.e. Null);  494 Residual
## Null Deviance:       693
## Residual Deviance: 569   AIC: 581
```

```r
myglm(Y,X,distribution  = "binom")
```

```
## $theta
##           [,1]
## [1,] -0.87757
## [2,] -0.83540
## [3,]  0.11336
## [4,] -0.06956
## [5,]  0.61555
## [6,]  0.93599
##
```

```
## $steps
## [1] 2
```

可以看到和真值相差不大, 但是很快收敛.

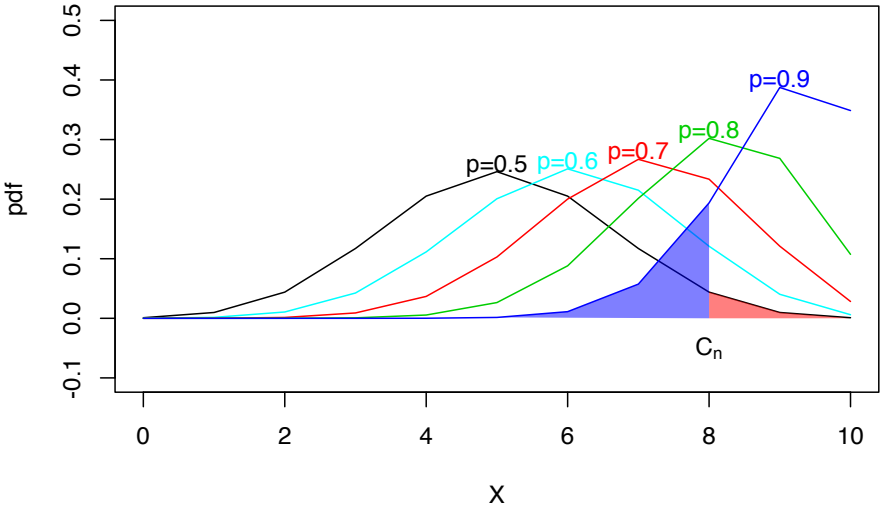
第三章　假设检验

假设检验部分模拟相对简单, 只需要在估计出参数值后, 按照检验统计量的形式正确计算即可. 主要难点在于构造检验统计量以及给出检验统计量的渐近分布.

常用的办法是通过自助法 (bootstrap) 估计分布.

我们以最简单的二项分布为例, 说明一些假设检验中的概念.

X	p=0.5	p=0.6	p=0.7	p=0.8	p=0.9
0	0.0010	0.0001	0.0000	0.0000	0.0000
1	0.0098	0.0016	0.0001	0.0000	0.0000
2	0.0439	0.0106	0.0014	0.0001	0.0000
3	0.1172	0.0425	0.0090	0.0008	0.0000
4	0.2051	0.1115	0.0368	0.0055	0.0001
5	0.2461	0.2007	0.1029	0.0264	0.0015
6	0.2051	0.2508	0.2001	0.0881	0.0112
7	0.1172	0.2150	0.2668	0.2013	0.0574
8	0.0439	0.1209	0.2335	0.3020	0.1937
9	0.0098	0.0403	0.1211	0.2684	0.3874
10	0.0010	0.0060	0.0282	0.1074	0.3487



显著水平 = 第一类错误 $=\alpha$: 为图中红色区域的面积

第二类错误 $=\beta$: 为图中蓝色区域的面积

功效 = 势 =power$=1-\beta$: 为图中蓝色曲线下空白面积

检验的相合性:

1. 在 H_0 下, 拒绝概率 (size) 收敛到 α.
2. 在 H_1 下, 拒绝概率 (power) 收敛到 1.

第四章　进阶技巧

程序的可读性和执行效率是两个很重要的要求. 这一章主要介绍如何提高这两点.

4.1　apply 函数族

apply 并不能提高执行效率, 只能使代码简洁易读.

详细的介绍可以在 apply 函数族介绍[1]查看, 其中给了一些简单的例子. 下面演示一下稍微复杂的用法.

对矩阵按列进行操作, 每列的奇数项求和, 偶数项求和:

```
m <- matrix(1:12,4,3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
apply(m, 2, tapply,rep(1:2,2), sum)
```

```
##   [,1] [,2] [,3]
```

[1] http://blog.fens.me/r-apply/

```
## 1    4   12   20
## 2    6   14   22
```

对矩阵按列进行操作, 每列前两项求和, 后两项求和:

```
apply(m, 2, tapply,rep(1:2,each=2), sum)
```

```
##   [,1] [,2] [,3]
## 1    3   11   19
## 2    7   15   23
```

下面看一个 3 维情况的例子.

```
a <- array(1:24,dim = c(2,3,4))
a
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
```

```
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

固定 1 个维度, 对另外 2 个维度求和:

```
apply(a,1,sum)
```

```
## [1] 144 156
```

这样看起来不太直观, 我们利用 aperm 函数调整数组的维度顺序:

```
aperm(a,c(2,3,1))
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    3    9   15   21
## [3,]    5   11   17   23
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]    2    8   14   20
## [2,]    4   10   16   22
## [3,]    6   12   18   24
```

固定 2 个维度, 对 1 个维度求和:

```
apply(a,c(2,3),sum)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    3   15   27   39
## [2,]    7   19   31   43
## [3,]   11   23   35   47
```

固定 2 个维度, 对 1 个维度分组求和:

```
apply(a,c(1,2),tapply,rep(c(-1,-2),2), sum)
```

```
## , , 1
##
##    [,1] [,2]
## -2   26   28
## -1   14   16
##
## , , 2
##
##    [,1] [,2]
## -2   30   32
## -1   18   20
##
## , , 3
##
##    [,1] [,2]
## -2   34   36
## -1   22   24
```

对第三个维度奇数项、偶数项分别求和 (奇数项是-1 组, 偶数项是-2 组).

4.2 并行

单次模拟时间越长, 重复次数越多, 并行得到的提升越明显.

R 中有很多并行包, 可以在 不同并行包比较[2]查看对比.

这里介绍的 foreach 是比较友好的一个包.

下面以线性模型估计系数为例, 给出示例代码.

[2] https://yulongniu.bionutshell.org/blog/2014/06/25/parallel-package/

sim_single 可以在 单次估计程序[3] 查看. 其中 SLP 参数用于增加单次模拟的计算量 (运行时间).

4.2.1 低重复次数, 低计算量

```
source("code/parallel-demo.R")
library("foreach")
library("doParallel")
beta0 = c(1,-2,3)
N = c(50,100,200)
distribution= c(rnorm,rcauchy)
SIM = 500
tstart=Sys.time()
cl <- makeCluster(detectCores())
registerDoParallel(cl)
result <- foreach(n=N,.combine = rbind) %:%
  foreach(dst=distribution,.combine = c) %:%
  foreach(i=1:SIM,.combine = '+',
          .packages = c("MASS") )%dopar%{
            sim_single(n,beta0,SLP=FALSE,mydist = dst)
          }
stopImplicitCluster()
t.end=Sys.time()
t.end-tstart
```

```
## Time difference of 2.34 secs
```

```
result/SIM
```

```
##            [,1]   [,2]  [,3]    [,4]   [,5]   [,6]
## result.1 0.9980 -2.000 3.002 -1.5225  5.281 33.300
## result.2 1.0001 -1.997 2.998 -0.8967  1.138  2.556
```

[3] %22code/parallel-demo.R%22

```
## result.3 0.9978 -1.999 3.007  0.6673 -1.630  3.195
```

把申请 cluster 的命令去掉,%dopar% 换成%do% 程序就会按串行执行

```
source("code/parallel-demo.R")
library("foreach")
library("doParallel")
beta0 =  c(1,-2,3)
N = c(50,100,200)
distribution= c(rnorm,rcauchy)
SIM = 500
tstart=Sys.time()
#cl <- makeCluster(detectCores())
#registerDoParallel(cl)
result <- foreach(n=N,.combine = rbind) %:%
  foreach(dst=distribution,.combine = c) %:%
  foreach(i=1:SIM,.combine = '+',
          .packages = c("MASS") )%do%{
            sim_single(n,beta0,SLP=FALSE,mydist = dst)
          }
#stopImplicitCluster()
t.end=Sys.time()
t.end-tstart
```

```
## Time difference of 2.633 secs
```

```
result/SIM
```

```
##            [,1]   [,2]  [,3]   [,4]   [,5]    [,6]
## result.1 0.9871 -2.006 2.994  5.461 -2.015 -0.8637
## result.2 1.0030 -1.993 3.006 -1.129 -1.252 -1.0508
## result.3 0.9953 -1.998 2.999  1.527 -2.218  3.2188
```

4.2.2 高重复次数, 低计算量

增加到 1000 次.

并行:

```
source("/Users/wang/Documents/GitHub/Simulation-in-R/code/parallel-demo.R
library("foreach")
library("doParallel")
beta0 = c(1,-2,3)
N = c(50,100,200)
distribution= c(rnorm,rcauchy)
SIM = 1000
tstart=Sys.time()
cl <- makeCluster(detectCores())
registerDoParallel(cl)
result <- foreach(n=N,.combine = rbind) %:%
  foreach(dst=distribution,.combine = c) %:%
  foreach(i=1:SIM,.combine = '+',
          .packages = c("MASS") )%dopar%{
            sim_single(n,beta0,SLP=FALSE,mydist = dst)
          }
stopImplicitCluster()
t.end=Sys.time()
t.end-tstart
```

```
## Time difference of 3.668 secs
```

```
result/SIM
```

```
##           [,1]   [,2]  [,3]    [,4]    [,5]   [,6]
## result.1 1.001 -1.999 3.001  1.8729 -0.7882 10.724
## result.2 1.001 -1.999 2.997 -0.6922 -2.1665  3.070
## result.3 1.000 -2.002 3.000  0.2200 -0.7819  1.951
```

串行:

```
source("code/parallel-demo.R")
library("foreach")
library("doParallel")
beta0 =  c(1,-2,3)
N = c(50,100,200)
distribution= c(rnorm,rcauchy)
SIM = 1000
tstart=Sys.time()
#cl <- makeCluster(detectCores())
#registerDoParallel(cl)
result <- foreach(n=N,.combine = rbind) %:%
  foreach(dst=distribution,.combine = c) %:%
  foreach(i=1:SIM,.combine = '+',
          .packages = c("MASS") )%do%{
            sim_single(n,beta0,SLP=FALSE,mydist = dst)
          }
#stopImplicitCluster()
t.end=Sys.time()
t.end-tstart
```

```
## Time difference of 5.494 secs
```

```
result/SIM
```

```
##            [,1]   [,2]  [,3]   [,4]   [,5]  [,6]
## result.1 1.0015 -2.001 2.993 0.4662 -6.437 3.672
## result.2 1.0007 -1.999 3.006 0.4559 -1.154 3.249
## result.3 0.9987 -1.999 3.000 2.7252 -1.533 3.364
```

4.2.3 低重复次数, 高计算量

增加每次模拟的计算量, 延长单次模拟的时间.

并行:

```
source("/Users/wang/Documents/GitHub/Simulation-in-R/code/parallel-demo.R
library("foreach")
library("doParallel")
beta0 = c(1,-2,3)
N = c(50,100,200)
distribution= c(rnorm,rcauchy)
SIM = 500
tstart=Sys.time()
cl <- makeCluster(detectCores())
registerDoParallel(cl)
result <- foreach(n=N,.combine = rbind) %:%
  foreach(dst=distribution,.combine = c) %:%
  foreach(i=1:SIM,.combine = '+',
          .packages = c("MASS") )%dopar%{
            sim_single(n,beta0,SLP=TRUE,mydist = dst)
          }
stopImplicitCluster()
t.end=Sys.time()
t.end-tstart
```

```
## Time difference of 10.73 secs
```

```
result/SIM
```

```
##            [,1]   [,2]  [,3]   [,4]     [,5]   [,6]
## result.1 1.0053 -2.005 3.000 3.3844  0.07196 0.9722
## result.2 0.9993 -1.999 3.004 0.4133 -1.99415 3.4178
## result.3 0.9992 -2.001 3.000 1.4560 -2.34520 3.9121
```

串行:

```
source("code/parallel-demo.R")
library("foreach")
library("doParallel")
```

```
beta0 =  c(1,-2,3)
N = c(50,100,200)
distribution= c(rnorm,rcauchy)
SIM = 500
tstart=Sys.time()
#cl <- makeCluster(detectCores())
#registerDoParallel(cl)
result <- foreach(n=N,.combine = rbind) %:%
  foreach(dst=distribution,.combine = c) %:%
  foreach(i=1:SIM,.combine = '+',
          .packages = c("MASS") )%do%{
            sim_single(n,beta0,SLP=TRUE,mydist = dst)
          }
#stopImplicitCluster()
t.end=Sys.time()
t.end-tstart
```

```
## Time difference of 40.73 secs
```

```
result/SIM
```

```
##            [,1]   [,2]  [,3]    [,4]   [,5]   [,6]
## result.1 0.9967 -1.989 3.001 -0.9001 -2.721 6.8869
## result.2 1.0005 -2.006 3.008  0.9264 -2.611 3.5339
## result.3 0.9963 -1.997 3.002  0.2332 -2.145 0.7733
```

4.2.4 模型选择

除了蒙特卡洛模拟之外, 另外一个很适合并行的应用是模型选择. 下面的例子用 AIC 对线性模型进行选择:

```
AIClm <- function(Y,X,ind){#单次计算AIC,ind为纳入模型的变量下标
  AIC(lm(Y~X[,ind]))
```

```
}
gen_ind <- function(bt, index) bt*index#生成候选模型下标

n = 100
p = 18
m0 = p%/%3
p0 = sample(1:p,m0)
beta = rep(0,p)
beta[p0] = 1:2
X <- matrix(rnorm(n*p),n,p)
Y <- X%*%beta + rnorm(n)

##生成所有候选模型的下标
pl <- 1:(2^p - 1)
mt <- matrix(0,2^p-1,p)
for (i in pl) {
  mt[i,]=rev(as.integer(intToBits(i)[1:(p)] ))
}
index = 1:p
lst = t(apply(mt, 1, gen_ind,index=index))

#并行
library("foreach")
library("doParallel")
tstart=Sys.time()
cl <- makeCluster(detectCores())
registerDoParallel(cl)
foreach(i=1:(2^p-1),.combine = c) %dopar%{
  AIClm(Y,X,lst[i,])
}->result
stopImplicitCluster()
t.end=Sys.time()
t.end-tstart
```

```
## Time difference of 1.495 mins
```

```
which(lst[which.min(result),]!=0)
```

```
## [1]  2  4  6  9 10 12 15 18
```

```
which(beta!=0)
```

```
## [1]  2  4  6  9 10 15
```

```
#串行
tstart=Sys.time()
cl <- makeCluster(detectCores())
registerDoParallel(cl)
foreach(i=1:(2^p-1),.combine = c) %do%{
  AIClm(Y,X,lst[i,])
}->result
stopImplicitCluster()
t.end=Sys.time()
t.end-tstart
```

```
## Time difference of 3.478 mins
```

```
which(lst[which.min(result),]!=0)
```

```
## [1]  2  4  6  9 10 12 15 18
```

```
which(beta!=0)
```

```
## [1]  2  4  6  9 10 15
```

4.3 Rcpp

Rcpp 提供了 R 与 C++ 的无缝接口, 可以很方便的在 R 中调用编写的 C++ 程序.

Rcpp 文档[4]

如何改写 R 程序[5]

Rcpp 已提供的分布函数[6]

可以通过 cppFunction 直接在 R 中编写:

```
Rcpp::cppFunction('int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')
add
```

```
## function (x, y, z)
## .Call(<pointer: 0x10af678f0>, x, y, z)
```

```
add(1, 2, 3)
```

```
## [1] 6
```

但是这种方式在编写 (没有语法高亮)、调试 (定位编译报错行号) 时都有不便. 推荐单独编写 cpp 文件, 通过 sourceCpp 加载.

下面以矩阵按列求和为例, 比较 col_mean(自己编写的 C++ 函数)[7]、colMeans(R base 中提供的注重速度的函数)、mean_R(在 R 中用编写的函数) 以及 apply 的运行速度.

---

[4] https://teuder.github.io/rcpp4everyone_en/

[5] https://adv-r.hadley.nz/rcpp.html

[6] https://teuder.github.io/rcpp4everyone_en/220_dpqr_functions.html#list-of-probability-distribution-functions

[7] 程序在 这里[8] 查看, 其中最后被/***R */夹住的部分是 R 程序, 每次加载后会自动执行, 方便调试.

```
Rcpp::sourceCpp('code/Rcpp-demo.cpp')
```

```
##
## > mean_R <- function(X) {
## +     n = dim(X)[1]
## +     m = dim(X)[2]
## +     cm <- rep(0, m)
## +     for (i in 1:n) {
## +         cm = cm + X[i, ]
## +     }
## +    .... [TRUNCATED]
```

```
m = 200
n = 100
X <- matrix(rnorm(m*n),m,n)
col_mean(X) -> l1
mean_R(X) -> l2
all.equal(l1,l2)
```

```
## [1] TRUE
```

```
colMeans(X) -> l3
all.equal(l1,l3)
```

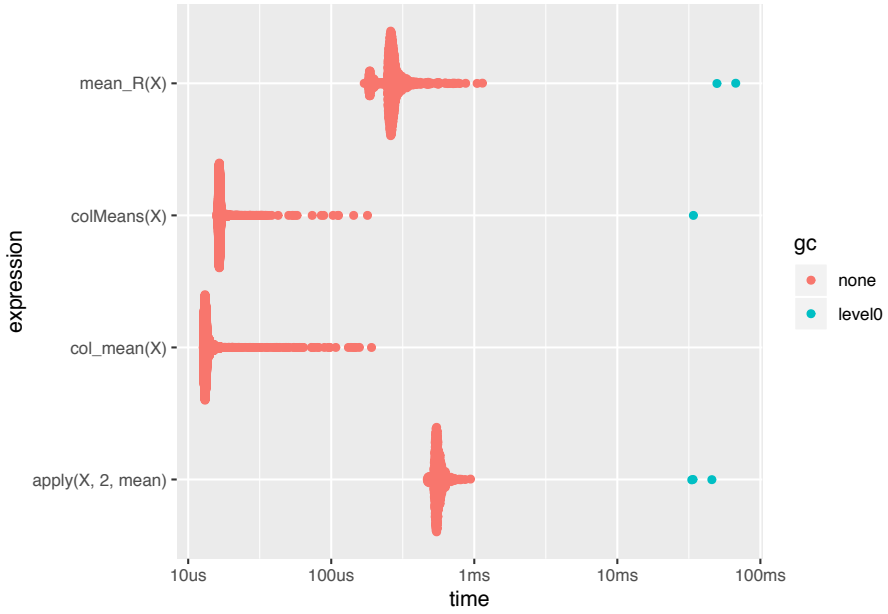
```
## [1] TRUE
```

```
apply(X, 2, mean) -> l4
all.equal(l1,l4)
```

```
## [1] TRUE
```

```
bench::mark(
  col_mean(X),
```

```
  colMeans(X),
  mean_R(X),
  apply(X, 2, mean),
  check = FALSE,relative = TRUE
)->results
ggplot2::autoplot(results)
```



对比结果如图所示, 其中 gc[9]是一个关于内存使用的指标, 越低越好.

apply 和在 R 中用循环编写函数速度差不多, 用 C++ 编写的函数明显比其他快, 甚至比 base 库中的函数还要快. 不过, 当我们增加矩阵的大小时, 就会发现不一样的结果:

```
Rcpp::sourceCpp('code/Rcpp-demo.cpp')
```

```
##
## > mean_R <- function(X) {
```

[9]垃圾回收（英语：Garbage Collection，缩写为 GC），在计算机科学中是一种自动的存储器管理机制。当一个计算机上的动态存储器不再需要时，就应该予以释放，以让出存储器，这种存储器资源管理，称为垃圾回收。

```
## +     n = dim(X)[1]
## +     m = dim(X)[2]
## +     cm <- rep(0, m)
## +     for (i in 1:n) {
## +         cm = cm + X[i, ]
## +     }
## +    .... [TRUNCATED]
```

```
m = 2000
n = 1000
X <- matrix(rnorm(m*n),m,n)
col_mean(X) -> l1
mean_R(X) -> l2
all.equal(l1,l2)
```

```
## [1] TRUE
```

```
colMeans(X) -> l3
all.equal(l1,l3)
```

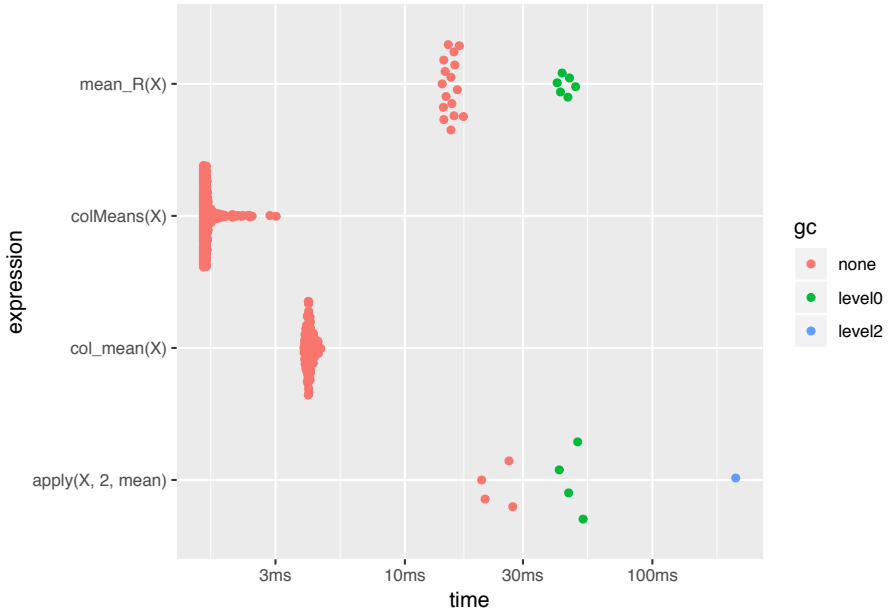
```
## [1] TRUE
```

```
apply(X, 2, mean) -> l4
all.equal(l1,l4)
```

```
## [1] TRUE
```

```
bench::mark(
  col_mean(X),
  colMeans(X),
  mean_R(X),
  apply(X, 2, mean),
  check = FALSE,relative = TRUE
)->results
```

```
ggplot2::autoplot(results)
```



可以看到, 还是 base 库中提供的函数速度最快.

4.3.1 RcppParallel

C++ 并行库

官方文档[10]

这里提供一个 RcppParallel 的例子: 核估计[11]

下面是三个关于线性代数的库,https://gist.github.com/wolfv/ca3ac2b24e1daf70f85eac18ec7b1b8f 这个例子的测试结果表明 xtensor 最快

[10] https://rcppcore.github.io/RcppParallel/index.html

[11] https://github.com/Ri0016/kernelCpp

4.3.2 RcppArmadillo

线性代数库 官方文档[12]

4.3.3 RcppEigen

线性代数库 (更快一点, 但是不友好) 官方文档[13]

4.3.4 xtensor

GitHub 地址[14]

[12] https://cran.r-project.org/web/packages/RcppArmadillo/RcppArmadillo.pdf
[13] https://cran.r-project.org/web/packages/RcppEigen/RcppEigen.pdf
[14] https://github.com/QuantStack/xtensor-r

附录 A　余音绕梁

呐，到这里朕的书差不多写完了，但还有几句话要交待，所以开个附录，再啰嗦几句，各位客官稍安勿躁、扶稳坐好。

参考文献

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2019). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.11.

索引