前言

随着国内经济迅速发展，统计分析在各个领域中扮演着举足轻重的角色。举例说，2008 年在北京举行的奥运会，有大量数据需要处理以作市场分析与预测。要分析数据，便要应用统计软件。因现时的计算机科技与日俱增，不同的计算机软件层出不穷，让人眼花撩乱。这本小册子只有一个目的，就是扼要地介绍一个免费而专业的统计计算机语言及软件平台——R。

早于廿年前，专业统计界已经广泛应用另一统计软件 S-PLUS。S-PLUS 为收费软件平台，其计算机语言称为 S，由于这软件十分昂贵，因此未能普及于教育界。直至 90 年代末，跟 S 语言极相近的 R 面世，而最重要是 R 的软件平台是免费的，这大大增加了它的普及性。虽然 R 为免费软件，但它拥有很多其它收费软件如 SAS 或 SPSS 没有的优点：

- 它可以处理不同类型的数据，如数字或非数字的数据也可一并处理，应用层面增广，举例说，市场数据往往存在大量的非数字数据，例如性别、籍贯等等。假若要把非数字数据分开处理，便费时误事。而 R 却能一并处理，十分方便；
- R 为独立程序，用者可以自行编写所需的程序，不像一般软件，规范严谨，缺乏弹性；

- R 拥有高质素的图像工具，这对统计分析提供了很大的帮助。再者，R 的图像工具弹性很高，用者可加进自己的要求，绘出所需的图像。R 的另一特点为拥有七彩缤纷的绘图能力，增加统计的乐趣；
- 大部份常用与非常用的统计工具已被写成 R 程序，用者可按需要去加减，从而进行统计分析。再者，相关的 R 网页已存有大量写成的程序，读者只要在相关网页下载便能免费使用。而一般收费软件的平台却价钱昂贵，一般人难以支付；
- R 也可与其它计算机语言如 C ++ 及 FORTRAN 合并，很多先进的统计概念可以运算出来。它的应用层面除了统计，还遍布医疗、金融经济、商业等等。

基于上述种种优点，R 在近几年间快速普及起来，在欧美十分通行。现在北京大学和中国人民大学都已经开始运用 R 教授统计分析课程。虽然统计分析在中文为主的社会中已十分普及，但有关 R 的中文课本却寥寥可数。这是笔者写这本书的主要动机之一，希望能藉此推广 R 的应用。

全书共分九章，第一章为基本入门的简单指令；第二章介绍数据及对象的类型；第三章介绍各类统计分布及如何在 R 进行模拟实验；第四章介绍有关 R 程序编写；第五章是关于读取及整理数据。第六及第七章为关于 R 的统计模型；第六章主要是常用的线性统计模型如线性回归、方差分析及广义线性模型；而第七章是关于一些常用多元统计方法如聚类分析、分类树及人工神经网络等；第八章是有关 R 的造像功能；最后一章则关于最优化方法。

在编写过程中，笔者以一般读者为对象。除了第六章及第七章关于统计模型外，并没有假设读者具备深厚的统计或计算知识。本书其中一大特点是以实例来解释 R 的秘诀，读者只要按部就班，根据书中的实例一步步地练习，应很容易地掌握 R 的特征及要点。换句话说，假如读者能边读边练习，把例子重复，便能很有效地学会怎样利用 R 作统计分析与推断。

本书所用的数据，大部份都是 R 内置数据。可以依照书内的指

令，直接在 R 中读取。只有几个很小的数据档案不是内置的。读者可到以下网站下载或自行输入亦不会有困难。一般来说，假若读者能花上数小时于每章内，不用数星期便能读毕全书。应用 R 便能驾轻就熟，应付自如。当然，假若读者完成这书后有兴趣了解更深入的统计或 R 概念，读者大可参考其它书籍，而这书便成读者入门的良伴。

在编写本书过程中，得到人民大学吴喜之教授提供宝贵建议，大大改进本书的内容及编排，在此作者衷心感谢吴教授的支持及协助。作者亦感谢中国香港特别行政区研究资助局基金赞助。当然，本书内之漏洞错误皆为作者之责任。

数据档案下载网站：http://www.sta.cuhk.edu.hk/books/Rdata/

R

目录

R

第一章

R 入门

这一章主要介绍 R 的基本使用方法，包括开启及关闭 R、输入数字、向量和一些简单指令及内置函数的应用。

1.1 下载及安装 R

R 是免费软件，读者可以根据以下程序于网上下载：

1. 进入 http://www.r-project.org/；

2. 点击 CRAN，然后选择最近的镜子网站(mirror)；

3. 选择 Windows (95 and later)，假设读者使用微软窗口；

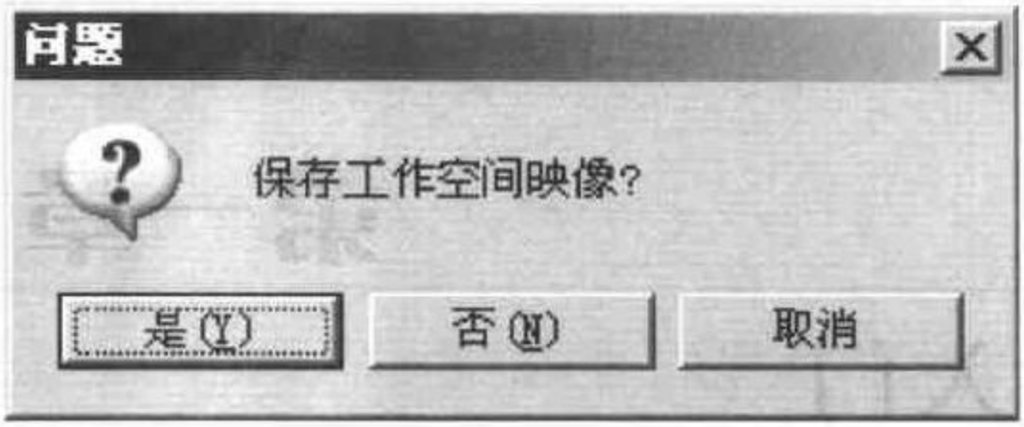
4. 选择 base，然后点击 rw2011.exe 开始下载至硬盘（档案大约有 25 兆字节）。

5. 下载后双击 rw2001.exe 进行安装。

1.2 启动和退出 R 环境

启动和退出 R 程序的方法和其它微软窗口软件一样，在程序集里按出 R 选项便可进入 R 程序；点右上角的框或输入 q() 就能退出。在退出 R 时会出现以下的选项框：

使用者可选择保存或不保存工作空间映像。如果选择保存，则这次在 R 出现的数据对象及自设函数等都会保存。（关于 R 数据对象及自设函数等本书将详细介绍）。下一次启动 R 时这些数据对象及自设函数等都会自动加载。如果选择不保存，则这些数据对象及



自设函数等都会消失。至于保存与否,则视使用者喜好而定。

启动 R 后,使用者可以看见以下文字:

```
R : Copyright 2005, The R Foundation for Statistical Computing
Version 2.2.0 (2005-10-06 r35749)
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

>
```

在 R 窗口中出现“ > ”这符号,就是表示 R 等待使用者在这里输入指令。

1.3 输入指令

使用者只需在 > 后输入指令,然后按 enter 就可以执行指令。举例说,输入 2 + 3 后按 enter,R 就会显示答案:

```
> 2 + 3
[1] 5
```

四则运算的次序跟数学的法则相同，乘号、除号及指数符号(Exponentiation)分别以 * 、/ 及 ^ 来表示。

函数的格式可以分为“有自变量”和“无自变量”两类。如果函数 functionname 是没有自变量的，那么只需输入 functionname( )便可。但假若它是需要自变量 arg1，arg2 的话，那么使用者便要输入 functionname(arg1，arg2)。如果只输入 functionname(没有括号)，R 便会把它的定义显示出来。使用者必须注意一点，在 R 中，大小写是有分别的，所以 functionname( )和 Functionname( )代表着两个不同的函数。

使用者可参考以下例子，平方根(square root)和绝对值(absolute value)都是需要输入一个自变量的：

```
> sqrt(9)
[1] 3

>abs( -5)
[1] 5
```

这里的 [1] 代表着所得答案的第一个元素。这种表示方式在表达一串数字所组成的向量时是很重要的。下列例子中第 36 个数值是 1000：

```
[1]    0.73   1.46   2.19   2.92   3.65   4.38   5.11   5.84    6.57
[10]   7.30   8.03   8.76   9.49  10.22  10.95  11.68  12.41   13.14
[19]  13.87  14.60  15.33  16.06  16.79  17.52  18.25  18.98   19.71
[28]  20.44  21.17  21.90  22.63  23.36  24.09  24.82  25.55 1000.00
[37]   1.04   1.30   1.56   1.82   2.08   2.34    2.6   2.86    3.12
[46]   3.38   3.64   3.90   4.16   4.42   4.68   4.94
```

使用者如输入不完整的指令时(例如：没有输入关括号)，符号 + 便会出现，它表示 R 正等待使用者输入余下的指令，例如：

```
> sqrt(16
+
```

使用者只要输入“)”就可以完成指令:

```
> sqrt(16
+ )
[1] 4
```

1.4 R 原始码的使用(R Source code)

使用者除了可每次输入一个指令外,还可以选择把一组程序编写成指令文件,并在 R 里执行。一般来说,指令文件可以把多个指令放在一起,以便易于修改,如计算仿真模型等。但对于处理一些需要互动输入及探索性的数据分析,则用处不大。使用者可利用文本编辑器来编写指令文件,并把所有指令输入文件内。(注意:在档尾段必须留有空白行,否则 R 可能不能正常执行最后一行指令。)

使用者先用记事本在某数据夹(例如 C:\ test)建立及编写一个文字模式的 R 原始码档案(例如 myfile. r)。在 R 的选单中选取 file - > change dir. . . ,更改预设的目录为 C:\ test。然后输入指令:source(“myfile. r”,echo = T)。R 就会执行在 myfile. r 档案中的每句指令。自变量 echo = T 是要显示 R 执行的指令。

1.5 R 在线说明

R 的使用说明只提供个别功能的用法和定义,并无提供整体的功能指南。使用者最初会感到困难,但一旦熟悉了,便会发现这种方式能有效地说明功能的用法、所需自变量及例子等。例如,若想找出绝对值(absolute value)的应用方法,只需输入 help(abs) 即可。这个窗口提供了一个接口给使用者从不同的题目中找寻适合的功能使用说明。还有,在 R 的选单中选取 help,里面有很多说明档案。在此推荐 Html help 中的 An Introduction to R,特别适合初学人士。

1.6 常用功能

R 会把使用者所输入的全部指令储存，使用者可以利用上、下光标键来选取以前输入的指令。选取适当指令后，可用输入键重新执行此指令；亦可用左、右光标键来更改指令。应用光标键来更改及执行以前输入的指令是十分方便快捷的。还有，在 R 的选单下面有很多不同的小图像，包括加载 R 原始码、加载及储存影像文件、剪贴及复制和停止目前运算等。

1.7 变量与赋值

使用者可以用“=”或“<-”来将数值赋给一个变量。在旧版中只可使用“<-”，至于用哪个符号则视使用者喜好而定。任何英文字母、数字、“-”及“.”都可作为变量名称。但是第一个字必须是英文字母。除此之外，R 不容许变量名称中有空格或“-”，如 Water Depth 就必须改为 WaterDepth 或 Water. Depth，x-1 可改为 x_1。如果要给变量 x 赋予数值 9，只需输入：

```
> x = 9    或    > x <- 9
```

使用者可能已注意到赋值后是不会显示出它的数值的。如使用者想确定赋值是否成功，可以输入该变量名称看看。当 x 有其数值时，就可以使用它作运算：

```
> sqrt(x)
[1] 3
> y = (5 * ( x + 2 )) - 3
> y
[1] 52
```

这些运算不会影响 x 的数值。倘若想重新给 x 赋值，可参考以下例子：

```
> sqrt(x)
[1] 3
```

```
> x
[1] 9
> x = sqrt(x)
> x
[1] 3
```

另外,使用者也可赋予一个字符串,如:

```
> y = "hello"
> y
[1] hello
```

现在变量 x 和 y 都储存在 R 的工作空间(workspace)中,使用者可输入 objects() 来检查目标中对象的清单。要删除不需要的对象,如 x,可用 remove("x")或 rm(x)。

问题:怎样可以把 x 和 y 的数值对调呢?

1.8 向量

上述所用的例子都是标量的,但在统计学里,多数数据都是以一组来表达的。R 中,使用者能以向量形式来输入一组数字。举例来说,在一个重复实验中得出 10 个结果如下:

2, 4.6, 1, 3.7, 5.9, 4.0, 6.7, 2.8, 1.4, 3.1

使用者可以利用向量形式把这组数据储存在一个变量中:

```
> observations = c(2, 4.6, 1, 3.7, 5.9, 4.0, 6.7, 2.8, 1.4, 3.1)
```

在这里 observations 是一个包含了 10 个数值的向量,而 c()则指示 R 在括号中的数值是以向量形式输入。而再输入 observations 时,则会显示变量的内容:

```
> observations
[1] 2.0 4.6 1.0 3.7 5.9 4.0 6.7 2.8 1.4 3.1
```

向量的运算与标量的运算是相同的。例如使用者可将 observations 的单位由英吋转换成厘米:

```
> 2.54 * observations
[1] 5.080 11.684 2.540 9.398 14.986 10.160 17.018 7.112 3.556
7.874
```

注意:变量 observations 中的每个数值都乘以 2.54。这就是向量化运算(vectorized operation),是 R 语言中一大特色,亦大大简化了 R 的程序编写工作。还有,在以上的例子中,observations 仍然保留原有的数值。

```
> mean(observations)
[1] 3.52
> var(observations)
[1] 3.450667
```

函数 mean( )和 var( )计算变量的平均值和方差。注意:只有当所有的自变量皆是向量的情况下,函数 mean( )和 var( )才能正常运作。下面第一例子计算 1,3,5 的平均值,但却得到 1。而第二例子中所用的才是正确的方法:

```
> mean(1, 3, 5)
[1] 1
> mean(c(1, 3, 5))
[1] 3
```

问题:怎样才可使向量 observations 标准化(normalized)令其平均值和方差分别为 0 和 1 呢?

1.9 从向量中选取子集

使用者可在向量名称后加上中括号 [ ] 以选择向量的子集:

```
> observations[3]
[1] 1
> observations[5:7]
[1] 5.9 4.0 6.7
> observations[c(1, 2, 7, 1)]
```

[1] 2.0 4.6 6.7 2.0

在上述例子中,R 分别传回了向量 observations 里的第 3 个,第 5 至第 7 个,以及第 1, 2, 7 和重复第 1 个元素。若在中括号中输入“-n”,则隐藏向量中的第 n 个元素。例如:

```
> observations[ -1 ]
[1] 4.6 1.0 3.7 5.9 4.0 6.7 2.8 1.4 3.1
```

此外,R 还提供了基本逻辑功能“>, <, >=, <=, ==, !=”(分别为“大于”、“小于”、“大于或等于”、“小于或等于”、“等于”及“不等于”),方便使用者从向量中选择适当的子集。举例说,若想找出 observations 中所有数值大于 4 的元素,使用者可输入:

```
> observations[ observations > 4]
[1] 4.6 5.9 6.7
```

1.10 R 例子

虽然上述例子全都是以人手输入数据的,其实 R 也可以读取外置的数据档案,详情将会在第四章讨论。现在,我们可以先利用 R 预先储存的数据作讨论。事实上 R 已预先储存不少有名的数据,使用者可以键入 data( ) 来观看所有档案。我们就以其中一个档案“trees”来作示范。

这档案包括了 31 棵樱桃树数据。我们首先用 data(trees) 去读取此档案。然后可用 names(trees) 及 dim(trees) 去显示档案中的变量及样本数目:

```
> names(trees)
[1] "Girth" "Height" "Volume"
> dim(trees)
[1] 31 3
```

这档案包含三个变量:树身周长(Girth),高度(Height)及可用木材之体积(Volume);亦有 31 个样本。(dim(trees) 的第一个数字是样本数目;第二个是变量数目,在第二章内将会详细介绍 dim)。我

们可用以下来显示它们的数值：

```
> trees $ Girth
[1] 8.3 8.6 8.8 10.5 10.7 10.8 11.0 11.0 11.1 11.2 11.3 11.4 11.4
[14] 11.7 12.0 12.9 12.9 13.3 13.7 13.8 14.0 14.2 14.5 16.0 16.3 17.3
[27] 17.5 17.9 18.0 18.0 20.6

> trees $ Height
[1] 70 65 63 72 81 83 66 75 80 75 79 76 76 69 75 74 85 86 71 64 78 80
[23] 74 72 77 81 82 80 80 80 87

> trees $ Volume
[1] 10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 24.2 21.0 21.4
[14] 21.3 19.1 22.2 33.8 27.4 25.7 24.9 34.5 31.7 36.3 38.3 42.6 55.4
[27] 55.7 58.3 51.5 51.0 77.0
```

假若想计算每棵树的大概体积，可先将 trees $ Girth 之平方乘以向量 trees $ Height 除以 4π，然后储存在 vol：

```
> vol = trees $ Girth^2 * trees $ Height/(4 * pi)
```

读者亦可以利用函数 summary( ) 得出 vol 的统计概要：

```
> summary(vol)
Min. 1st Qu. Median Mean 3rd Qu. Max.
382.6 750.2 969.9 1141.0 1375.0 2938.0
```

stem-and-leaf plot 功能可有效显示出数据的分布情况：

```
The decimal point is 3 digit(s) to the right of the |
0 | 444
0 | 667778888889
1 | 00112223
1 | 569
2 | 0011
2 | 9
```

综合来说，大部分树木的体积都在 2,000 立方英尺以下。树木的体积愈大，是否可用木材的体积也愈大呢？要回答这问题，我们首

先看看 trees $ Volume 的统计概要：

```
> summary(trees $ Volume)
   Min. 1st Qu.  Median Mean 3rd Qu.  Max.
   10.20 19.40 24.20 30.17 37.30 77.00
```

然后选取体积大于 2,000 立方英尺树木，看看它们的可用木材是否较大：

```
> trees $ Volume[vol > 2000]
[1] 58.3 51.5 51.0 77.0
```

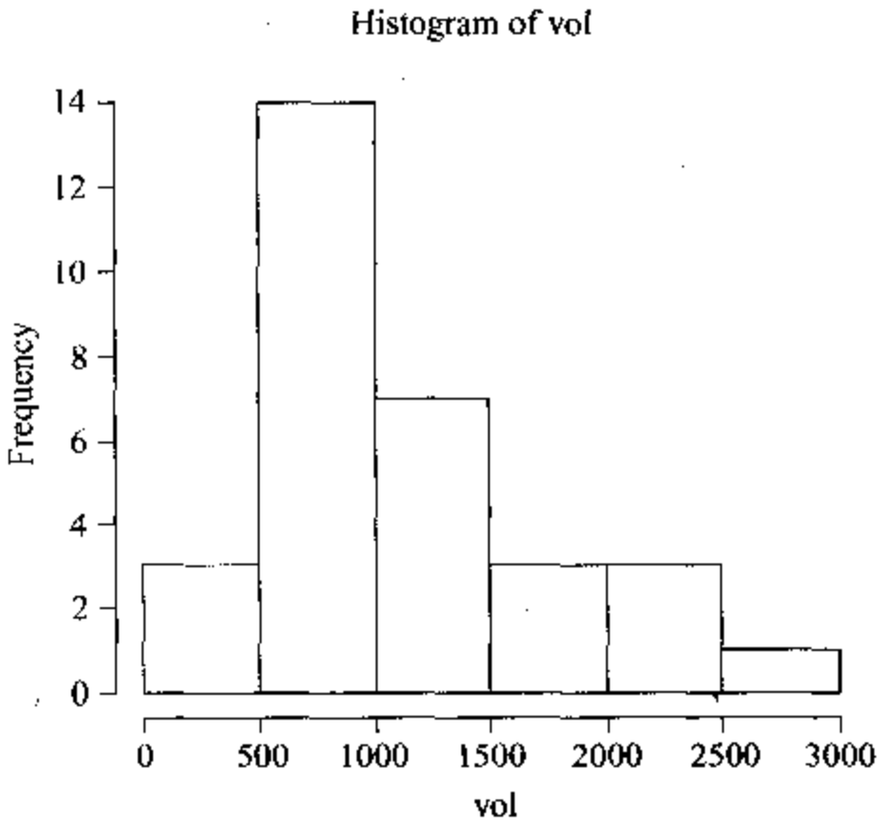
很明显，树木的体积愈大，可用木材的体积亦愈大。

问题：请试用 stem – and – leaf plot 来展示出所有可用木材体积少于 300 树木的体积。

1.11 R 图像解说

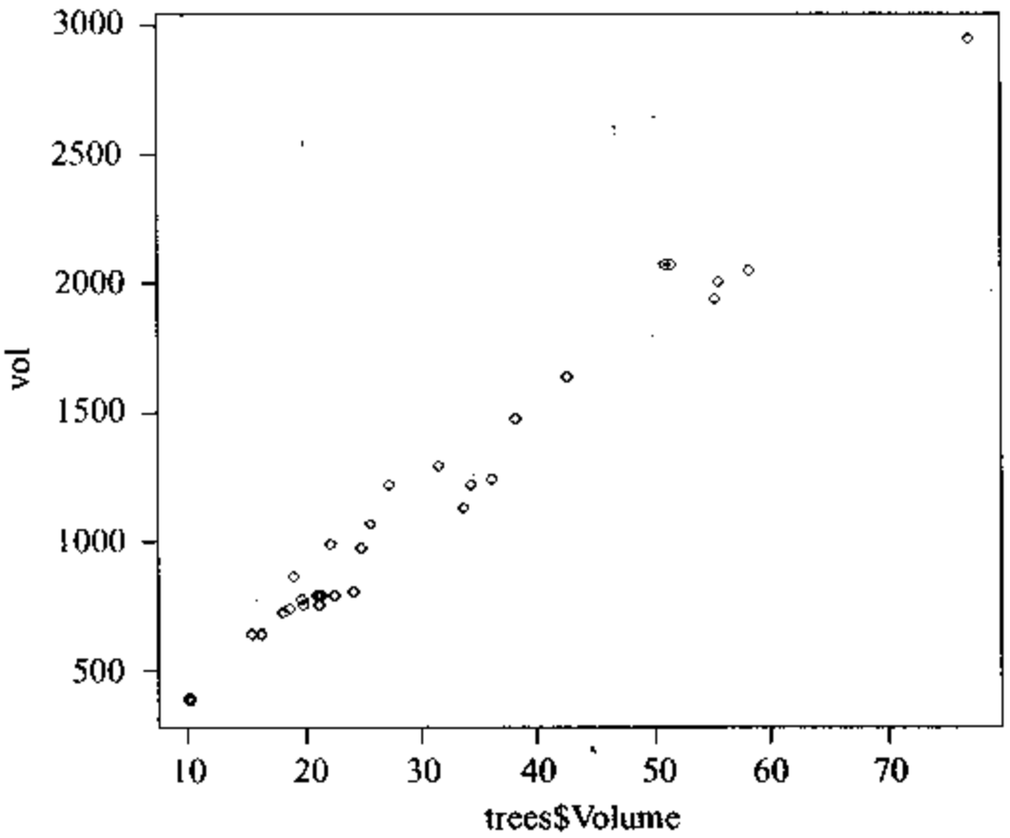
有时利用图像解说更能清楚表达出数据的特质。例如想画出树木体积的直方图，方法如下：

```
> hist (vol, col = 0)
```



R 的图像颜色是由图像窗口中的选项清单来控制，因此使用者可根据自己的意愿随意修改。若想以图像方式来表达树木体积和可用木材的关系，可在 R 输入指令：

```
> plot(trees $ Volume,vol)
```



这样，trees $ Volume 就会显示在图像中的 x 轴，而 vol 则会显示于图像中的 y 轴。图像中也显示出它们有明显的关系。

问题：请找出 vol 与 trees $ Girth，或 vol 与 trees $ Height 的关系，请以图像分析说明。

注意：若只输入一个向量，函数 plot 会把向量数值显示在 y 轴，而数值的次序则会显示在 x 轴。使用者可利用这方法来作残差分析（residual analysis），观察余量的标差是否一致。在完成所有图像分析后，输入 q( ) 便可关闭图形窗口及退出 R。

第二章

对象与数据类型

2.1 引言

R 把对象分为多种类型，如向量、矩阵、因子(factor)、清单(list)及数据框(data frame)等。部分 R 函数会根据不同的对象作出不同的处理。譬如，函数 plot 会随输入的数据类型的不同而得出不同类型的图像：输入数字向量，会得到散点图(scatter plot)；输入一组因子，就会得到盒形图(box plot)；输入回归模型的结果，则会得到一系列的图像分析(在 R 里，大部分复杂的分析功能，如回归模型，都会得到一系列的结果)。这种弹性被称为“对象导向式程序编写”(object-oriented programming)。

所有对象都包含数值及属性(attributes)。数值是指当输入对象名称后所显示的数值，而属性则可分为三类：对于向量的长度，对于矩阵的维(dimensions)及对于数据框的行和列。以下将会介绍各种最常见的对象及其属性。

2.2 标量(Scalar)与向量

R 中最简单的对象就是标量，它可以是由数字或字符串所组成。在 R 的语法中，表达文字时都必须加上引号“”。例如：“x”代表字母 x，但 x 则代表对象 x。同一道理，“1”代表符号，而 1 则是一个数字。

向量是由一组一维的标量所构成的，所以标量也可说是长度为

一单位的向量。除此之外，这些标量必须是同一类型（数值或文字）。若一向量同时包含了数值和文字，R 会把该向量视为文字处理。如：

```
> blah = c(1, 2, 3, "d")
> blah
[1] "1" "2" "3" "d"
```

在这例子中，R 把数字 1、2、3 都视为文字 "1"、"2"、"3"，使向量 blah 的所有成员变成同一类型。

使用者可利用中括号来查看或变更向量内特定元素的内容（请参考第一章）：

```
> blah [1]
[1] "1"
> blah [c (1, 2, 3)]
[1] "1" "2" "3"
> blah [4]
[1] "d"
> blah [4] = "zip"
[1] "1" "2" "3" "zip"
```

向量的属性为长度。利用功能 length 便可查看或变更向量的长度：

```
> length(blah)
[1] 4
> length(blah) = 5
> blah
[1] "1" "2" "3" "zip" "NA"
```

当向量 blah 的长度被改为 5 时，"NA" 就会新增成为其第五元素。在 R 中，NA 是代表遗漏值（missing value）。

问题：如果减少向量的长度，会得出怎样的结果？

2.3 类型检视与转换

在不知对象类型时，使用者可利用一系列由 is. 串连的功能去检视，R 便会用 TRUE 以表示正确或 FALSE 以表示错误，如：

```
> is. vector( blah )
[1] TRUE
> is. character( blah )
[1] TRUE
> is. numeric( blah )
[1] FALSE
```

这就像 R 回答了以下 3 条问题：

blah 是向量吗？（正确）

blah 是由字符串组成的吗？（正确）

blah 是由数值组成的吗？（错误）

R 除了可作类型检视外，也可以利用 as. 变更对象的类型：

```
> as. numeric( "1" )
[1] 1
> as. numeric( "hello" )
[1] NA
Warning messages:
NAs introduced by coercion 1 missing value generated coercing from character to numeric in: as. numeric( "hello" )
```

由于"hello" 并不属于数值类型。因此 as. numeric("hello") 只会显示出 NA。

2.4 因子（Factor）

因子是 R 中一项非常有用的工具。它是由一组代表不同程度的元素所组成的向量，而这些程度的相对影响是未知的。

假设在一个医学实验中有 10 个病人用 3 个分别名为安慰药

(placebo)、medicine A 和 medicine B 的治疗法。前 3 个病人接受安慰药;第四至第七个病人接受 medicine A;最后 3 个病人则接受 medicine B。而回应变量(response variable)就是病情的康复程度。数据中包含两个向量:病人所接受的疗法(疗法向量)和他们所需的康复时间。

使用者可能已注意到疗法向量是不能以数值编码的。因为数字本身具有次序和数值的比较,所以并不能用于代表名称的疗法向量上。假设我们分别以数字 0、1、2 代表疗法 placebo、medicine A 及 medicine B。这会暗示着疗法 medicine B 的好与坏是 medicine A 的两倍。但实际上,当实验还没完成,这是没有根据的。因此,疗法向量应以文字来输入并转化为因子:

```
> medicine = c(rep("placebo", 3), rep("A",4), rep("B",3))
> medicine
[1] "Placebo" "Placebo" "Placebo" "A" "A" "A" "A" "B" "B" "B"
> medicine = as. factor(medicine)
> medicine
[1] Placebo Placebo Placebo A A A B B B
```

注意:函数 rep(what, n)是指把自变量 what 重复 n 次的缩写。以下的函数 level( ) 可以显示因子的水平:

```
> levels(medicine)
[1] "A" "B" "Placebo"
```

以上的疗法向量是一个有 3 种水平的因子,但是水平间是没有特定次序的。若有需要,使用者可以利用"次序因子"(ordered factor)去表达水平间的等级。譬如说,一间市场研究公司正进行一部新电影受欢迎程度的调查,它所得的数据可能包括观众对电影的评价:

```
> responses
[1]   Great  Good  Good  Poor  Great  Fair  Awful  Fair  Good
[10]   Fair  Poor  Great  Fair  Good  Poor   Fair  Fair  Great
```

```
> levels (responses)
[1]  "Awful"  "Fair"  "Good"  "Great"  "Poor"
```

很明显,因子 responses 有五个水平,而水平间的比较是不依字母次序的。因此,使用者必须以函数 ordered 来编排恰当的次序:

```
> responses = ordered (responses, levels = c ("Awful", "Poor", "Fair", "Good", "Great"))
> responses
[1]  Great Good Good Poor Great Fair Awful Fair Good
[10]  Fair Poor Great Fair Good Poor Fair Fair Great
Awful < Poor < Fair < Good < Great
```

R 这时会明白因子 responses 是有次序的。Awful 代表最差,而 Great 则代表最好。

忠告:使用者并不能藉更改 levels 的次序来改变数据的次序,应该利用上述的 ordered 函数。另外,只有当程度的名称需作改动时,使用者才应更改程度的属性。例如,我们可以将"A"及"B"的名称分别改为"Medicine A"及"Medicine B"。

```
> medicine
[1] Placebo Placebo Placebo A A A A B B B
> levels (medicine)
[1] "A" "B" "Placebo"
> levels (medicine) = c("Medicne A", "Medicine B", "Placebo")
> medicine
[1] Placebo Placebo Placebo Medicine A Medicine A Medicine A
[7] Medicine A Medicine B MedicineB Medicine B
```

问题:请自制一个新的向量 moreresponses,它必须由 responses best movie ever 和 worst movie ever 所组成,把它设定为次序因子,并以 worst movie ever 为较低水平。在向量 responses 中加入向量 moreresponses。最后,以 worst movie ever 为最低水平(在 Awful 之下)及 best movie ever 为最高(在 Great 之上),重新输入次序。若有需要,可更改

levels 上的名称。

2.5 矩阵(Matrix)

矩阵和向量有点相似,但它是二维的。输入矩阵如同输入向量,只需加上它的二维数据。例如:

```
> matrix(c(1: 9), nrow =3, ncol =3, byrow = T)
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

上述例子中,自变量"byrow =T"提示 R 以一行行来排列矩阵。当矩阵的第一行被填满时,余下的数字将填补下一行,如此类推,直至完成矩阵。byrow 的默认值是"F",因此,若不输入自变量"byrow =T",则会被视作以下列方式组成矩阵:

```
> matrix(c(1: 9), nrow =3, ncol =3)
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

注意:自变量的写法并非一成不变,nrow 及 ncol 可以简化为 nr 及 nc。我们只需输入行数或列数:matrix(c(1: 9),nr =3) 或 matrix(c(1: 9),nc =3),亦可得到同样结果。我们甚至可以进一步简化为 matrix(c(1: 9),3,3) 或 matrix(c(1: 9),3)。简写自然较为方便,但亦较难明白。使用者应在两者之间取得平衡。

由于矩阵和向量的紧密关系,它们也可分别以 as. vector 及 as. matrix 互相转换。as. matrix 能使一个向量变成 n x 1 的矩阵,而 as. vector 则能把矩阵变成为向量(从左至右一列列的读取)。矩阵拥有两个属性:"二维数据"和"行和列的名称"。

```
> bulls = matrix(c("Ron", "Michael", "Scottie", "Dennis",
```

"Luc", "Steve", "Toni", "Jud", "John", "Bill"), nrow = 2, ncol = 5, byrow = T)

```
> bulls
     [,1]    [,2]      [,3]      [,4]      [,5]
[1,] "Ron"   "Michael" "Scottie" "Dennis"
[2,] "Steve" "Toni"    "Jud"     "John"
```

函数 dim 能显示矩阵的二维数据:

```
> dim (bulls)
[1] 2 5
```

使用者可利用函数 dimnames 来查看或更改行和列的名称。注意,二维名称必须以清单方式输入。(参照 2.6)

```
> dimnames(bulls)
NULL
> dimnames(bulls) = list(c("Starters", "Bench"), c("PG", "SG", "SF", "PF", "C"))
> bulls
          PG      SG        SF        PF      C
Starters "Ron"   "Michael" "Scottie" "Dennis" "Luc"
Bench    "Steve" "Toni"    "Jud"     "John"   "Bill"
> dimnames(bulls)
[[1]]:
[1] "Starters"  "Bench"

[[2]]:
[1]  "PG"  "SG"  "SF"  "PF"  "C"
```

2.6 清单(List)

清单比向量有更大效用,因它能储存多种类型的数据在一起。倘若向量是一组标量的排列,那么清单则可视为一组不同形态对象的排列。例如,它可把一组文字向量,一组数值矩阵和一组数值向量

编排在一起。

```
> blah = list(c("1", "two", "three", "4"), matrix(c(1, 2, 3,
4, 5, 6), nrow=2, ncol=3), c(1, 7, 17))
> blah
[[1]]:
[1] "1" "two" "three" "4"
[[2]]:
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6


[[3]]:
[1]  1  7  17
```

当拣选清单内的元素时,方法跟向量或矩阵有所不同,需提供两组自变量(分别以[ ]加载)。第一组自变量(需用双中括号)表示拣选清单内的哪些对象,而第二组自变量(单中括号)则表示哪些对象内被拣选的元素。若不输入第二组自变量,整个对象将会被显示出来。

```
> blah[[1]]
[1] "1"  "two"  "three"  "4"
> blah[[1]][3]
[1] "three"
> blah[[2]][1:2, 2]
[1] 3 4
```

清单中的对象也可以被命名的:

```
> names(blah) = c("some.characters", "a.matrix", "some.numbers")
> blah
$some.characters:
[1] "1"  "two"  "three"  "4"
```

```
$ a. matrix:
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

$ some. numbers:
[1]    1    7    17
```

一旦清单已被命名,便可直接拣选其对象出来:

```
> blah $ some. characters[4]
[1] "4"
> blah [[1]] [4]
[1] "4"
```

2.7 数据框(Data Frame)

到目前为止,使用者可能已能认识到 R 对象的类型有很多相似之处。数据框可以是多维的,而且有行和列可被命名等性质。在数据框内,亦如数值向量(或因子)一般,空白输入是以 NA 来表示。

利用指令 data. frame,可将矩阵转化成数据框(在 R 中并没有函数 as. frame 的)。同样,利用 as. matrix,便可将数据帧转化成矩阵。但使用者必须记住一点:矩阵和向量一样,只能拥有一种数据类型,而数据框却能同时拥有多种。若数据框内同时含有文字,当数据框被转化成矩阵时,所有元素都会转变成文字。

2.8 例题:数据框和因子的应用

以下的数据取自 1979 年 New York Choral Society 内每一位歌唱家的身高(以最接近的整数英吋作单位)。研究目的在于测试身高与音域间的关系。歌唱家首先会被分类为男低音(Bass),男高音(Tenor),女低音(Alto)及女高音(Soprano)。

Soprano	Alto	Tenor	Bass
64	65	69	72
62	62	72	70
66	68	71	72
65	67	66	69
60	67	76	73
61	63	74	71
65	67	71	72
66	66	66	68
65	63	68	68
63	72	67	71
67	62	70	66
65	61	65	68
62	66	72	71
65	64	70	73
68	60	68	73
65	61	73	70
63	66	66	68
65	66	68	70
62	66	67	75
65	62	64	68
66	70		71
62	65		70
65	64		74
63	63		70
65	65		75
66	69		75
65	61		69
62	66		72
65	65		71
66	61		70
65	63		71
61	64		68
65	67		70
66	66		75
65	68		72
62			66
			72
			70
			69

2.8.1 读取和组织数据

首先假设数据储存在 c:\example\singers.csv。csv 代表逗号分隔,是一种电子表格(如 EXCEL)常用的格式。R 的 read.csv 函数可以读取此种格式的档案。但档案中的四列数据(Bass,Tenor,Alto,Soprano)长度不同。因此,用 read.csv 读取后变成数据框,空格则用 NA 填补。为方便起见,我们首先用选单中 file -> change dir...将预设数据夹转为 c:\example,然后输入

```
> d = read.csv("singers.csv")
> attributes(d)
$ names
[1] "Soprano" "Alto" "Tenor" "Bass"
$ class
[1] "data.frame"
$ row.names
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12" "13" "14"
[15] "15" "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28"
[29] "29" "30" "31" "32" "33" "34" "35" "36" "37" "38" "39"
```

d 的属性(attributes)有 names,class 及 row.names。R 虽已读取了所有数据,但这些数据包括 NA,排列方式并不适用于分析。我们要重组数据成为两列的矩阵。首先,我们将 names(d)储存,然后将 d 转成矩阵,再转成向量。

```
> part = names(d)
> part
[1] "Soprano" "Alto" "Tenor" "Bass"
> d = as.matrix(d)
> d = as.vector(d)
> d
```

```
 [1]  64 62 66 65 60 61 65 66 65 63 67 65 62 65 68 65 63 65 62 65 66
[22]  62 65 63 65 66 65 62 65 66 65 61 65 66 65 62 NA NA NA 65 62 68
[43]  67 67 63 67 66 63 72 62 61 66 64 60 61 66 66 66 62 70 65 64 63
[64]  65 69 61 66 65 61 63 64 67 66 68 NA NA NA NA 69 72 71 66 76 74
[85]  71 66 68 67 70 65 72 70 68 73 66 68 67 64 NA NA NA NA NA NA NA
[106] NA NA NA NA NA NA NA NA NA NA NA NA 72 70 72 69 73 71 72 68 68
[127] 71 66 68 71 73 73 70 68 70 75 68 71 70 74 70 75 75 69 72 71 70
[148] 71 68 70 75 72 66 72 70 69
```

此向量长度为156;首39个数字为Soprano的身高;随后39个数字为Alto的身高;如此类推。向量中亦包含NA。将文字向量part相应扩充,首39个元素为"Soprano",接着为"Alto",如此类推。然后将part和d合成为两列矩阵temp。

```
> part = rep(part, each = 39)
> temp = matrix(c(d, part), nc = 2)
```

要留意一点,在这矩阵中,由于part为文字向量,因此,d也会被转化为文字。现在,我们选取在temp中不是NA的行,然后把它转化为数据框:

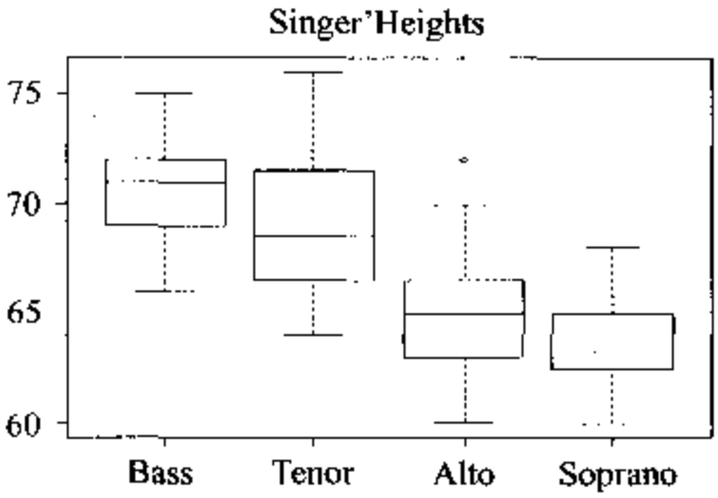
```
> temp = temp[! is. na(temp[,1]),]
> singers = data. frame(temp)
> names(singers) = c("Height", "Part")
> singers $ Height = as. numeric(Height)
> singers $ Part = ordered(as. factor(singers $ Part), levels = c("Bass","Tenor", "Alto", "Soprano"))
```

最后,singers就转成我们想要的数据框。

2.8.2 数据分析

由于singers已是个数据框,我们可以用attach来挂上此数据框。首先,利用以下程序代码造出一些盒形图:

```
> attach(singers)
> plot((Part, Height)
> title(main = "Singers' Heights")
```



要注意,如 part 的内容没有被编码成次序因子,盒形图便会依据字母次序来输出。另外,若 part 的内容也没有被设定成因子,那么,R便会输出点图(dot plots)而不是盒图。

图像显示,音域较低的歌唱家,身高普遍都较高。当然,其中一个原因,是由于男性唱男高音或男低音,而女性唱女高音或女低音所致。所以,我们较有兴趣知道的是,男高音与男低音,以及女高音与女低音之间的分别。

```
> t. test ( Height [ Part  = = "Soprano" ], Height [ Part  = = "Alto" ],
var. eq = T)

        Two Sample t-test

data: Height[ Part  = = "Soprano" ] and Height[ Part  = = "Alto" ]
t  =  -1. 1289, df  =  69, p - value  =  0. 2628
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1. 7591030 0. 4876745
sample estimates:
mean of x mean of y
64. 25000 64. 88571

> t. test(Height[ Part  = = "Tenor" ], Height[ Part  = = "Bass" ],var. eq
= T)

        Two Sample t-test
```

```
data: Height[Part == "Tenor"] and Height[Part == "Bass" G]
t = -2.1297, df = 57, p-value = 0.03753
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-3.04220979 -0.09368765
sample estimates:
mean of x mean of y
69.15000 70.71795
```

这些 t-test 的结果显示女高音与女低音之间的身高并没有明显的差异,但男低音的身高明显比男高音的身高高。

第三章

统计分布及模拟

3.1 引言

电子计算机在统计学上有重大影响及应用,模拟方法是其中最重要应用之一。本章会介绍如何用 R 编写简单模拟实验(simulation experiment)及应用相关的统计分布函数(statistical distribution)。我们假设读者具备一些基本的统计分布知识。

3.2 sample 函数

sample() 函数可用来产生随机排列或随机样本。以下的指令是产生从 1 到 10 的随机排列及随机样本:

```
> sample(10)
[1] 1 5 4 3 8 2 10 9 6 7

> sample(10,replace = T)
[1] 4 8 2 9 2 9 4 3 7 7
>
```

第二句指令加多了自变量 replace = T,因此每个数字都有机会被抽中多次,而每个数字被抽中机会均等 (= 1/10) 。我们也可以用 sample 来产生任何离散分布的随机样本。例如:

```
> sample(c( -1,0,1),size = 20,prob = c(1/4,1/2,1/4),replace = T)
[1] -1 1 0 0 0 0 -1 0 -1 0 0 0 0 0 -1 0 -1 1 1 1
```

以上指令产生了20个离散分布的随机样本,抽中 -1, 0, 及 1 的机会分别是1/4,1/2 及 1/4。读者可能注意到当输入以上指令时会得到不同结果。原因在于随机数产生器是应用随机种子(random seed)产生伪随机数据 (pseudo random number)。使用者在不同时间执行指令都会有不同随机种子。当然,我们可以用 set. seed( ) 函数来设定随机种子。当随机种子是一样时,所产生的伪随机数据也相同。例如:

```
> set. seed(12345)
> sample(10)
[1] 8 10 7 9 3 1 2 4 6 5
> sample(10)
[1] 1 2 6 10 3 8 9 4 7 5
> sample(10)
[1] 5 3 8 10 4 2 9 7 1 6
```

我们开始时设定了随机种子为 12345,然后产生不同的随机排列。原因是当每次执行产生随机数后,随机种子亦会依从某特定的数学公式改变。现在我们尝试将随机种子再设定回 12345,就会产生相同的随机排列:

```
> set. seed(12345)
> sample(10)
[1] 8 10 7 9 3 1 2 4 6 5
> sample(10)
[1] 1 2 6 10 3 8 9 4 7 5
```

使用者设定随机种子,每次产生相同的随机数据会对编写程序及除错有很大帮助。现在我们就应用 sample 来模拟随机步行(random walk) 。假设我们进行掷银赌博;如果掷到人像就赢一元,掷到文字就输一元。这就是一种简单的随机步行,我们可以下面统计模型表示:

$$w(t+1) = w(t) + r(t),\ t = 0, 1, \ldots, n.$$

r(t) 等于 1 或 -1 的机会是 1/2。w(t) 就是在时间 t 累积的金额。假设最初金额 w(0) =0,我们就用以下指令仿真这随机步行:

```
> set. seed(13579)
> r = sample(c( -1,1),size = 100,replace = T,prob = c(1/2,1/2))
> r = c(0,r)
> r
 [1]   0  1 -1  1 -1 -1  1  1 -1  1 -1 -1 -1 -1 -1 -1  1 -1  1  1 -1 -1 -1  1
[26]   1  1 -1  1 -1  1 -1  1  1 -1 -1  1  1 -1  1 -1  1  1  1  1 -1  1 -1
[51]   1  1 -1  1  1 -1  1 -1  1  1 -1 -1 -1 -1  1 -1 -1  1 -1  1  1  1  1 -1 -1
[76]  -1 -1 -1  1 -1 -1 -1  1 -1  1 -1  1  1 -1  1 -1  1  1 -1  1  1  1 -1  1
[101] -1
> w = cumsum(r)
> w
 [1]  0  1  0  1  0  1  2  3  2  1  2  1  0 -1 -2 -3 -4 -3 -4 -3 -2 -3
[23] -4 -5 -4 -3 -2 -1 -2 -1 -2 -1 -2 -1  0 -1 -2 -1  0 -1  0  1  0  1
[45]  2  3  4  3  4  3  4  5  4  5  6  5  6  5  6  7  6  5  4  3  4  3
[67]  2  3  2  3  4  5  6  5  4  3  2  1  2  1  0 -1  0 -1  0 -1  0  1
[89]  2  1  2  1  2  3  2  3  4  5  4  5  4
```

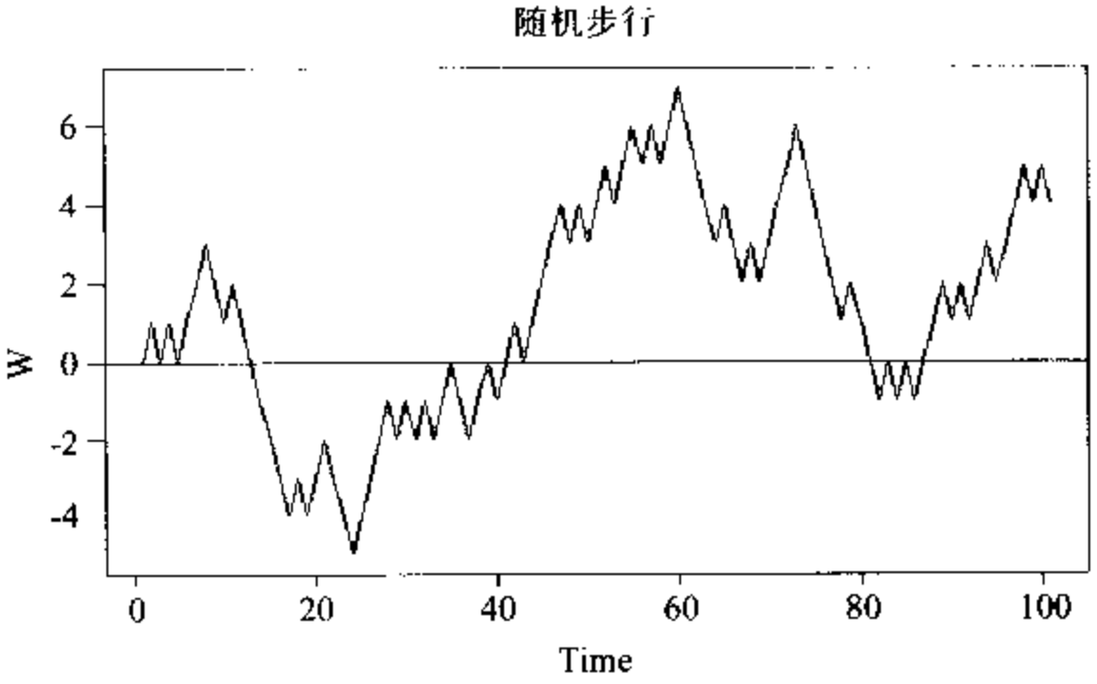
以上指令 r = c(0,r) 是将 w(0) =0 加在 r 前面,而 w = cumsum(r) 是计算 r 的累积和(cumulative sum)。现在让我们绘制这随机步行:

```
> w = as. ts(w)
> plot(w,main = "随机步行")
> abline(h =0)
```

注意:我们先要将 w 转成时间序列,然后用 plot( ) 函数来绘图。abline(h =0) 是在图中加上 y =0 的水平线。有关 R 绘制统计图表将会在第七章详细介绍。

3.3 统计分布

R 内置了很多常用的统计分布函数。我们可以用这些函数来产生各类型的随机样本。在介绍这些函数之前,先要说明关于 R 统计



分布的四个基本项目：

1. 概率密度函数（Probability density function，pdf）
2. 累积分布函数（Cumulative distribution function，cdf）
3. 分位数（Quantile）
4. 伪随机数（Pseudo random number）

R 的命名方法是分别用 d，p，q，r 去表示这四个项目。例如 dnorm，pnorm，qnorm 及 rnorm 分别代表正态分布的这四个基本项目。

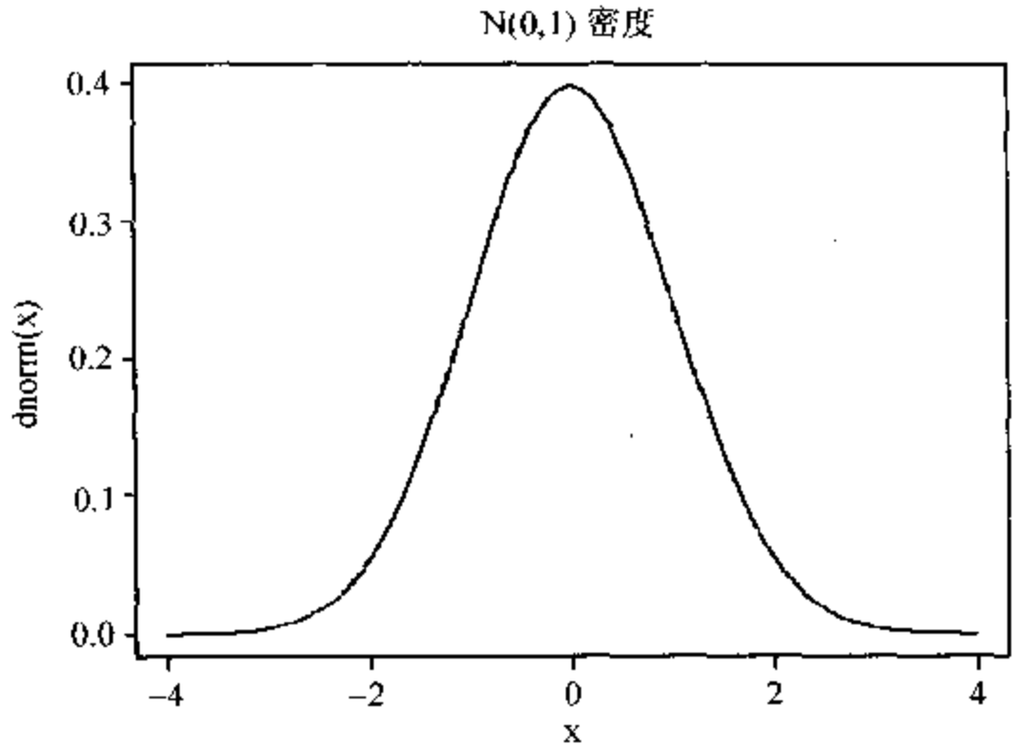
我们用以下指令绘制正态分布的概率密度函数（著名的钟形曲线）。注意，在 plot 函数加入自变量 type = "l" 是表示用线（line）来绘制。

```
> x = seq( -4,4,0.1)
> plot(x,dnorm(x),type = "l",main = "N(0,1) 密度")
```

应用 pnorm( ) 可以计算标准正态随机变量 Z < 1.96 的概率：

```
> pnorm(1.96)
[1] 0.9750021
```

现在再举一例。假设随机变量 T 有自由度为 5 的 t 分布，(T ~ t(5))，我们可以用以下指令计算 T > 2.3 的概率：




```
> 1 - pt(2.3,5)
[1] 0.03488623
```

注意:我们也可以用指令 pt(2.3,5,lower.tail = F) 得到相同答案。这概率在假设检验(Hypothesis Testing)称为右尾概率(right - tail probability),亦是单侧检验(one - sided test)的 p - 值。如果这 p - 值小于检验水平(通常是 0.05),则我们就拒绝接受零假设(Null Hypothesis)。

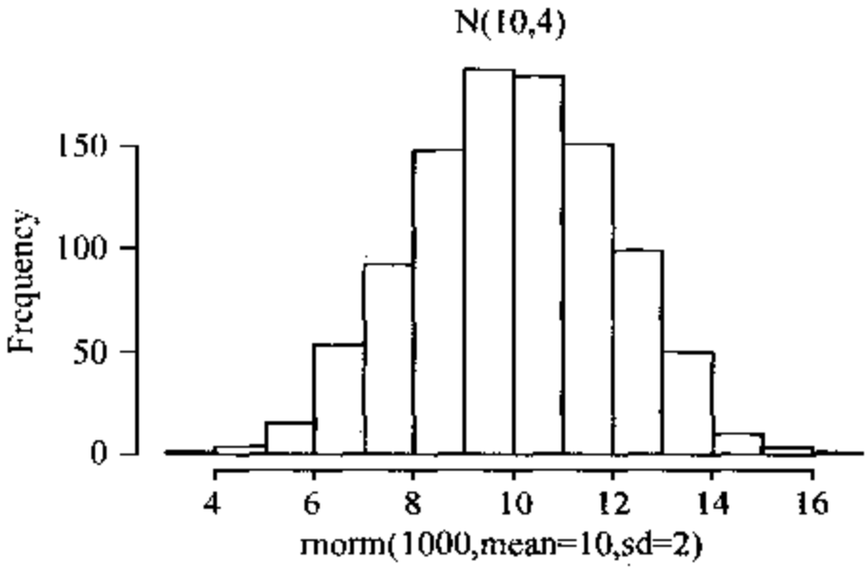
qnorm() 可以用来计算标准正态分布的分位数:

```
> q = c(0.025,0.05,0.5,0.95,0.975)
> qnorm(q)
[1] -1.959964 -1.644854 0.000000 1.644854 1.959964
```

首先定义概率向量 q,然后计算相应的分位数(这些都是假设检验常用的分位数)。

我们尝试用 rnorm() 去产生 1000 个 N(10,4) 的随机数据,然后绘制直方图(histogram)。

```
> hist(rnorm(1000,mean = 10,sd = 2),main = "N(10,4)")
```

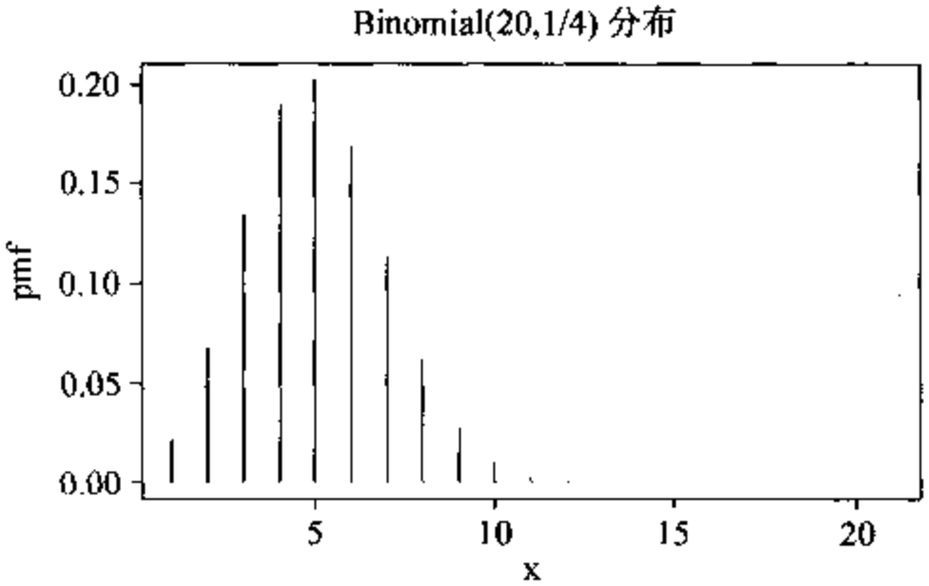


我们又尝试用二项式分布为例。二项式分布的密度函数为：

$$f(x)=\binom{n}{x}p^x(1-p)^{n-x},x=0,\ldots,n$$

以下指令是绘制 n = 20，p = 1/4 的二项式分布密度函数：

```
> x = seq(0:20)
> pmf = dbinom( ,size = 20,prob = 1/4)
> plot(x,pmf ,type = "h",main = "Binomial(20,1/4) 分布")
```



dbinom( )传回二项式分布的密度，自变量 size 及 prob 分别是指 n 和 p。在 plot 函数中 type = "h"代表用棒形图绘制，适用于离散分布。

pbinom( )函数可以计算二项式分布的累积概率。设 X 有以上二项式分布（Binomial(20,1/4)），以下指令可计算 X >8 的概率(尾部概率 tail probability)：

```
> 1 - pbinom(8,size = 20,prob = 1/4)
[1] 0.04092517
```

qbinom( )函数可以找出二项式分布的分位数。以下指令计算上述二项式分布的 1/10, 2/10, . . . , 9/10 的分位数：

```
> q = seq(0.1,0.9,0.1)
> qbinom(q,size = 20,prob = 1/4)
[1] 3 3 4 4 5 5 6 7 8
```

从以上的例子中,我们知道怎么用 d, p, q 及 r 去代表统计分布（如 norm, binom）的四个基本专案。下列表格包括了 R 内置常用统计分布。注意,每个分布都有自变量,有些有默认值（如 N(0,1)）；有些则无（如 binom)。

分布	R 名称	自变量
贝塔 Beta	beta	shape1, shape2, ncp
二项式 Binomial	binom	size, prob
柯西 Cauchy	cauchy	location, scale
卡方 Chi-square	chisq	df, ncp
指数 Exponential	exp	rate
F	f	df1, df2, ncp
伽玛 Gamma	gamma	shape, scale
几何 Geometric	geom	prob
超几何 Hypergeometric	hyper	m, n, k
对数正态 Log-normal	lnorm	meanlog, sdlog
逻辑斯谛 Logistic	logis	location, scale
负二项式 Negative binomial	nbinom	size, prob

续表

正态 Normal	norm	mean, sd
泊松 Poisson	pois	lambda
Student's t	t	df, ncp
均匀 Uniform	unif	min, max
威布尔 Weibull	weibull	shape, scale
威尔科克森 Wilcoxon	wilcox	m, n

3.4 中心极限定理 (Central Limit Theorem)

这一节示范何以应用模拟资料去说明一个在统计学上很重要的定理——中心极限定理。首先让我们回顾这定理。假设 $X_1, \ldots, X_n$ 为随机样本,而其总体方差 (population variance) 为有限时;则当 n 趋向无限大时,其样本均值 (sample mean) $\overline{X} = (1/n) \sum_{i=1}^{n} X_i$ 之抽样分布 (sampling distribution) 趋向正态分布。

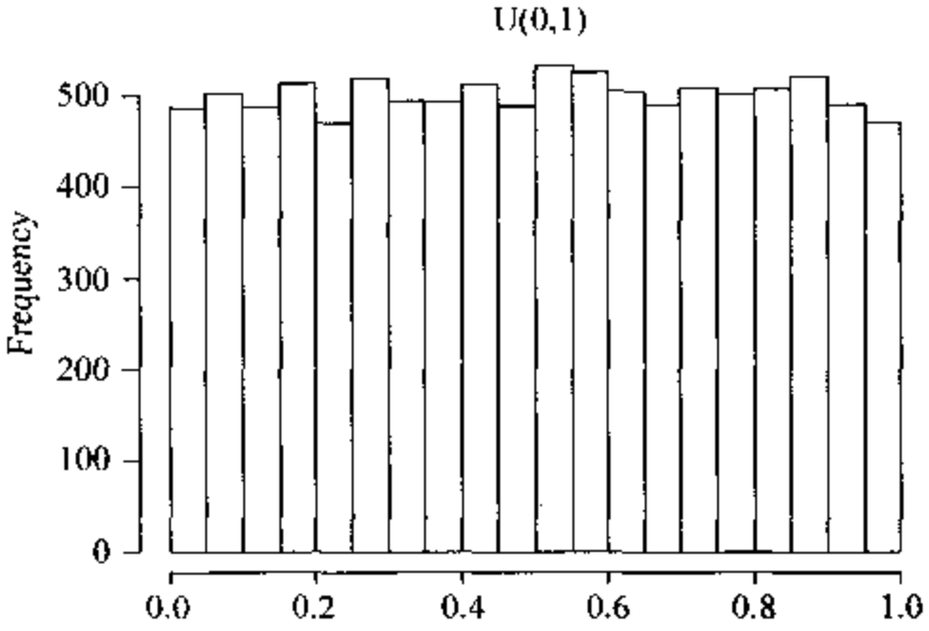
这定理的重要性在于适用于绝大部分的统计分布 (有限总体方差的分布)。在上一节表格的统计分布 (除了柯西分布外) 都适用。现在就让我们产生 10000 个独立的均匀分布 Uniform(0,1) 随机数据。

```
> set. seed(12345)
> u = runif(10000)
> summary(u)
    Min.      1st    Qu. Median      Mean      3rd      Qu.      Max.
5. 187e -05 2. 537e -01 5. 046e -01 5. 007e -01 7. 492e -01 1. 000e +00
> var(u)
[1] 0. 08240007
> hist(r,main = "U(0,1)")
```

这随机样本 r 的样本平均值及中位数都接近 0.5;而第一四分位 (first quantile) 及第三四分位 (third quantile) 亦接近 0.25 及 0.75。样

本方差亦接近总体方差（1/12 =0.0833）。u 的直方图十分合理：

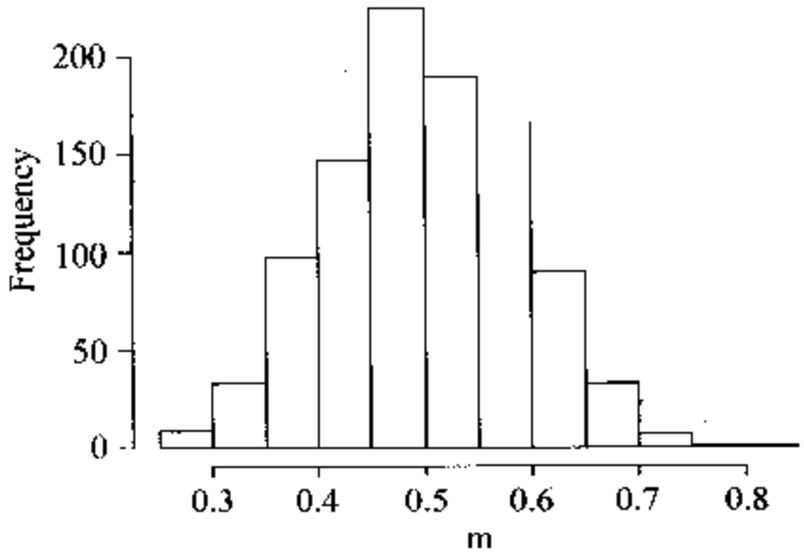
现在，我们将 u 转成一个 1000x10 的矩阵。在矩阵里的每一行都代表一个样本大小为 10 的随机样本 $X_1,\ldots,X_{10}$。然后计算每一行的样本平均值 $\bar{X}$。由于这矩阵有 1000 行；因此我们就有 1000 个 $\bar{X}$。这 1000 个 $\bar{X}$ 就可以用作估计 $\bar{X}$ 的抽样分布。注意：在计算每一行的样本平均值 时，我们应用 R 的内置函数 apply()。




```
> u = matrix(u, nc = 10)
> m = apply(u, 1, mean)
> summary(m)
   Min.    1st Qu.    Median    Mean    3rd Qu.    Max.
0.2620    0.4405    0.4982    0.5007    0.5665    0.8304
> var(m)
[1] 0.087678896
```

在 apply 的自变量 1 是指矩阵的第一维数，亦即是行。mean 就是我们要应用的函数。因此，apply（u，1，mean）就是要计算矩阵 u 内每一行的平均数。（在 4.5 节内亦会再介绍这 apply 函数）。根据统计理论，$\bar{X}$ 的抽样分布的中位数及平均数都接近 0.5，而样本方差则接近总体方差十分之一（0.00833）。最后，我们可以绘制 m 的直方图如下：

```
> hist(m, main = "样本平均(n = 10)的抽样分布")
```



问题:试重复上述之模拟实验;不过用 (i) 柯西(Cauchy)分布及 (ii) 自由度为5 的t 分布代替均匀分布,看看中心极限定理是否成立。

第四章

程序编写

4.1 引言

相信读者们已开始熟悉 R 的指令及某些内置函数。其实 R 也可以让使用者自设函数。本章将进一步介绍 R 的程序编写。

4.2 函数的编写

R 中不但有多个预设的函数，如平均数、对数(log)、线性模型(lm)等，而且也容许使用者建立及执行自定的函数。

在建立一个函数时，使用者必须让 R 知道该函数需要读取的自变量数目及它的功能。以下一系列指令将建立出一个包含了两个变量，名为 my. function 的函数。当完成执行这函数时，最后一行程序代码所代表的数值会被传回，这就是 value. to. return 的数值。

```
> my. function = function(arg1, arg2)
{
    ...
    value. to. return
}
```

大括号 { } 中所包含的是函数的执行部分。function 这个指令预期只会执行一个指令，因此，如要执行多重指令，便需将那些指令写于大

括号 { } 中。虽然 R 已有内置的标准差函数 sd( ), 但我们也可以自己写一个:

```
> stdev = function(x)
+ {
+       sqrt(var(x))
+ }
```

(在第一章提及过,当输入不完整的指令时, R 便会显示“+”号等待使用者输入完整的指令。)由于函数 function 只有一个指令,在这个情况下没有必要用大括号 { }。让我们建立一些向量来测试函数 stdev,看看它是如何运作的:

```
> z = c(1, 2, .45, -1.2, 2.3)
> stdev(z)
[1] 1.395708
```

函数就如向量和数据框一样,都是一种 R 对象。因此,其它对象的操作方法,都适用于函数。

```
> is.function (stdev)
[1] T
> my.stdev = stdev
```

函数内的变量赋值只在函数内有效。意思是当函数结束时,该变量便会恢复原来的值。试看以下例子,变量 test 的值在函数内转变了,但当函数结束时,test 的值便恢复成原来的值。函数 changer 没有任何自变量,只把 test 的值变成“changed”, 并传回 test 的值。

```
> test = "unchanged"
> changer = function( )
+ {
+       test = "changed"
+       test
+ }
```

```
> changer( )
[1] "changed"
> test
[1] "unchanged"
```

由于 changer 包含了两个指令，所以需在这组指令前后加上大括号。此外，需留意函数可以完全没有自变量及在 changer 的最后一行中，test 的值是这函数在执行时所传回的值。

有时函数中变量赋值被限制在局部的位置，这或许是件好事，因为不会对数据造成任何意外的损坏。但另一方面，这样会令编辑数据变得繁复。假若使用者真的要利用函数 changer 去变更 test 的值，可用以下做法：

```
> test
[1] "unchanged"
> test = changer( )
> test
[1] "changed"
```

对于复杂的函数来说，如能在其中加上解释其功能的批注，对使用者了解该函数的运作有莫大的帮助。程序上任何一行出现在 # 符号之后的指令都不会被执行，但每当显示函数时会出现在屏幕上。即是说，在每一行批注前都必须加上符号 #。另外，如果使用者只输入函数的名称，R 会显示该函数的定义。例如：

```
> stdev = function(x)
+ {
+     # This function calculates the standard deviation
+     sqrt(var(x))
+ }
> stdev
function(x
{
```

```
    # This function calculates the standard deviation
    sqrt(var(x))
}
```

4.3 函数的编辑

R 提供了一个 edit 函数，让使用者可在一个特定的窗口中更改自设函数，使用者便不用在每次更改函数时重打一次。fix 函数会使用窗口中的预设编辑器（记事本）。

```
> fix(stdev)
```

输入上述指令后，便会出现一个记事本的窗口，让使用者更改 stdev 之定义。当更改完毕后关闭窗口，新的定义便会自动储存。R 还有内置函数 edit，可以用来编辑 R 原始码的档案：

```
>edit(file = "myfile. r")
```

忠告：在编写函数及 R 原始码档案时，读者应尽量使用批注及缩排（indent）。这样可使函数及原始码更容易阅读及修改。绝大部分的程序编写员都会遵从这些习惯。

4.4 循环和逻辑

到目前为止，读者应该懂得编写一些简易的函数去执行一系列指令。但假如要执行一些比较繁复的程序，应如何是好呢？

当我们重复执行一些程序时，可利用指令 for 来作循环（loop）处理，其程序语法为：for(指标变量 in 执行范围){}。指标变量会依次取得执行范围中的每一个值。例如：以下的循环是用来计算从 1 加到 10 的总和。

```
> total = 0
> for(i in 1: 10)
+{
+       total = total + i
+ }
```

```
> total
[1] 55
```

也可以应用于较繁复的执行范围：

```
> total = 0
> for(i in c(1, 2, 4, 5, 9))
+     total = total + i
> total
[1] 21
```

以下的例子，进一步说明更多 for 循环的实际应用。函数 stretch 能读入一个向量，然后把向量中每个元素复制同等的次数。（函数首先建立一个空的向量，然后把每一个元素复制到其中。）有时候，这对于重组数据的格式编排有莫大的帮助。函数的重点在于使用了函数 rep(v, n)。rep 会建立一个向量，它储存了首个自变量 v，合共 n 次。

```
> stretch = function(vec, num)
+ {
+     output = c( )
+     for(i in 1: length(vec))
+         output = append(output, rep(vec[i], num))
          output
+ }
> stretch(c("big bird", "oscar", "snuffleupagus"), 4)
 [1]     "big bird"     "big bird"     "big bird"         "big bird"
 [5]       "oscar"        "oscar"        "oscar"            "oscar"
 [9]  "snuffleupagus"  "snuffleupagus"  "snuffleupagus"
[12]  "snuffleupagus"
```

注意，函数 rep 可把数据延长至不同的方式：

```
> rep(c("big bird", "oscar", "snuffleupagus"), 4)
 [1] "big bird"          "oscar"  "snuffleupagus"  "big bird"
 [5]   "oscar"  "snuffleupagus"        "big bird"     "oscar"
```

[9] "snuffleupagus" "big bird" "oscar"
[12] "snuffleupagus"

if 的语法为 if (条件式) { } else { }。条件式可以包括任何具有符号 <, >, ==, &, |(or)等的逻辑句子。如条件式正确,就会执行第一个大括号中的指令。如条件式不正确,则执行 else 部分之指令(else 部分可有可无)。例如:

```
> x = 1
> if(x < 1) {"x is less than one"} else{"x is greater than or equal to one"}
[1] "x is greater than or equal to one"
```

while 循环可以处理所有 for 循环能处理的事项,而且可控制较多普遍的循环。其语法为:while(条件式) { }。以下函数计算少于 y 的 x 之最大整数幂(其值必须大于0)。首两个 if 是用来确保 x 和 y 的值是合理的。随后的两个指示是用来把变量 i 及 total 的值初始化。while 循环便是这函数的重点所在:

```
> max.power = function(x, y)
+{
+    if (x < y & x > 0)
+        if (x > 1) {
+          i = 1
+          total = x
+          while (total * x < y) {
+            i = i + 1
+            total = total * x
+            }
+          i
+        }
+        else "Infinity"
+    else NULL
```

```
+ }
```

例子中的首两个 if 确保 x 与 y 的值在程序中是合理的。while 叙述中的变量 total 持续乘上 x 的值，并计算其执行的次数。最后传回 i 的值、"Infinity"或 NULL。

```
> max. power(2, 100)
[1] 6
> max. power(2, 1)
NULL
> max. power(.5, 100)
[1] "Infinity"
> max. power( -3, 100)
NULL
```

问题：你们会如何改写 max. power，使其在程序输出 NULL 的两个情况下（当第 1 自变量大于第 2 自变量或第 1 自变量是负数），会传回更加详细的讯息？

除了上述 for 及 while 迴圈外，R 还有 repeat 迴圈。repeat 跟 while 相似，不过 repeat 迴圈中要加入 break，用以逃离 repeat 迴圈。否则的话，程序就会困在无限迴圈（infinite loop）中。以下例子就是用 repeat 迴圈将 max. power 函数重写。

```
> max. power = function(x, y)
+ {
+     if (x < y & x > 0)
+         if (x > 1) {
+           i = 1
+           total = x
+           repeat {
+             if (total * x >= y) break
+             i = i + 1
+             total = total * x
```

```
+                 }
+                 i
+             }
+             else "Infinity"
+     else NULL
+ }
```

在 repeat 迴圈中,如果 total * x > = y 的话,就要用 break 来逃离迴圈。

4.5 Apply 函数

在 R 执行循环时是较花时间及资源的。当进行大量数据运算及大型模拟实验时,应尽量使用向量化运算而避免使用循环。以下的例子说明循环是可以避免的,指令 apply 会比循环更有用。在 R 的预设数据库中,有一个很有名的数据档案:iris。它包含了 150 个样本,记录了三类鸢尾花的 4 项特性。如要计算每项特性的平均值,用迴圈的做法是:

```
> get.mean = function(x)
+ {
+     report = rep(0, 4)
+     for (i in 1: 4)
+         report[i] = mean(x[, i])
+     report
+ }
>data(iris)
> get.mean(iris)
[1] 5.843333    3.057333    3.758000    1.199333
```

另有一种较快捷而不需任何循环的做法是:

```
> apply (iris[ ,1: 4], 2, mean)
Sepal L.      Sepal W.      Petal L.      Petal W.
5.843333      3.057333      3.758000      1.199333
```

以上的指令是计算在矩阵 iris 中每列的平均值。在 apply 的自变量 2 是指矩阵的第二维数,亦即是列。mean 就是我们要应用的函数。因此,apply (iris[,1: 4], 2, mean)就是要计算矩阵 iris[,1: 4] 内每一列的平均数。

4.6 防错

使用者或许想在函数中加入防错功能,R 就具备了一些这方面的应用函数:stop 及 warning。函数 stop 会传回错误讯息及终止执行函数;而函数 warning 则在传回错误讯息之后,仍然继续执行函数。

以下标准差函数的运作与以前相同,但当自变量不是向量时则传回错误讯息(is. vector 前的感叹号'(!) 是否定的意思,所以当 x 不是向量时便会出现错误讯息)。

```
> stdev = function(x)
+ {
+     # stdev with error handling
+     if (! is. vector(x))
+         stop ("The argument to stdev must be a vector")
+     else sqrt (var(x))
+ }
```

再举另一例,试看之前用来测试标准差函数的向量 z,以及 z 被转换成矩阵后的情况:

```
> stdev (z)
[1] 1.395708
> z = as. matrix(z)
> stdev (z)
Error in stdev (z):The argument to stdev must be a vector
```

R 会一字不漏地传回错误讯息,因此编写错误讯息时,必须确保错误讯息简洁易明。

问题:试编写 is. sym( ) 函数,首先检查输入矩阵是否正方,然后检

查矩阵对称。输出 True,False 或适当的错误讯息。

4.7 除错函数

R 有内置除错函数 debug( ) 帮助我们找出问题所在。debug 是个互动的除错器,使我们可以逐步执行函数内的指令及检查变量的数值。这些功能对除错工作有莫大帮助。我们可以输入 help(debug)去查看这函数的说明及指令。现在就让我们用在 4.4 节中的 max. power 来试试 debug。如果在正常情况下输入 max. power(2,9),R 会输出正确答案 3。现在输入 debug(max. power),函数 max. power 就进入除错模式(debug mode)。当输入 max. power(2,9)时,就会出现以下情况:

```
> debug(max. power)
> max. power(2,9)
debugging in: max. power(2, 9)
debug: {
    if (x < y & x > 0)
      if (x > 1) {
        i = 1
        total = x
        repeat {
          if (total * x >= y)
            break
          i = i + 1
          total = total * x
        }
        i
      }
    else "Infinity"
    else NULL
}
Browse[1] >
```

首先,R 会显示 max. power 函数的定义,然后就出现 Browse[1] > 符号。使用者可输入 n 或 next (或甚至可直接按输入键),R 就会执行下一行指令。使用者也可以在任何时候输入变量 (例如 x, i, total 等),R 就会显示变量当时的数值。使用者亦可输入 c 去继续执行指令,直至最后及退出 max. power。此时,函数 max. power 仍然在除错模式。使用者要输入 undebug(max. power)才可使 max. power 恢复正常模式。

在以下一节中,我们将会示范如果应用 debug 来帮助除错。

4.8 例子

在本节中,我们尝试透过编写函数 by. quantiles 为例子,逐步展示编写程序及除错的过程。

注意:在本节程序中,为方便作解说,会出现多个故意造成的错误。

首先,函数 by. quantiles 可以把向量 x 分成任何数目 n 的分位数 (quantiles)。假如 x 是 c(.42, .15, .27, .35, .38, .41, .1),那么 by. quantiles(x,2) 及 by. quantiles(x,4),分别会传回向量 (2, 1, 1, 1, 2, 2, 2, 1) 及 (4, 1, 2, 2, 3, 3, 4, 1)。这些数字代表在 x 内相应的元素是属于第几个分位。

R 有内置函数 quantile,它能读取向量形式的数据及概率切点 (probability breakpoints)向量(二分位数时是 c(0, .5, 1);四分位数时是 c(0, .25, .5, .75, 1)等)及传回数据的分位数切点。quantile 函数就是 by. quantiles 核心所在。它的语法及内容如下:

```
> by. quantiles = function(x, n)
{
    产生 n 分位数的概率切点向量
    利用函数 quantile 来产生真正的切点
    根据所得的切点,把数据 x 细分为 n 分位数。
    传回结果
}
```

首先,我们逐步编写 by. quantiles;我们需明白如何产生概率切点。概率切点是一个向量,其起始为 0,终结为 1。把数据分成两部分的向

量含有三个元素 (0, .5, 1),而把数据分为四部分的向量含有五个元素(0, .25, .5, .75, 1),所以我们可得出把数据分为 n 部分的向量会含有 n + 1 个元素。我们可利用 rep 函数来建立这个向量。我们最好分开来测试函数的每一部分,直到我们确定所有部分都相互协调。首先我们输入:

```
> by. quantiles = function(n, x)
+ {
+       pbreaks = rep(0, n + 1)
+       pbreaks [n +1] = 1
+       pbreaks
+ }
```

现在让我们测试以下例子:

```
> by. quantiles (c(1, 2, 3, 4), 2)
Error in rep. int:rep( ) only defined for length(times) = = 1 or length
(x):
Rep. int (1: xlen, times)
...
```

以上例子中,第一个自变量应为 x,而第二个自变量应为 n 才对。更正这个错误后再执行,便会输出“[1] 0 0 1”,到目前为止,仍看似正确无误。现在,我们需设法找出向量中间所包含的数值,其中一个方法是利用 for 循环。我们只需知道向量的首个及最尾的数值,然后逐一加入中间的数值。在二分位数的例子中,向量以 1/2 为计算单位:(0, 0.5, 1);而在四分位数的例子中,则以 1/4 为计算单位:(0, .25, .5, .75, 1)。因此在 n 分位数的情况下,向量应以 1/n 为计算单位。

利用 fix(by. quantiles)函数来变更 by. quantiles 如下:

```
function (x, n)
{
      pbreaks = rep(0, n +1)
      pbreaks [n +1] = 1
```

```
    for(i in 2 : (n-1))
        pbreaks[i] = 1/n
    pbreaks
}
```

现在我们又测试以下例子：

```
> by. quantiles (c(1, 2, 3, 4), 2)
[1] 0.5    0.5    1.0
```

这看来并不正确，因第一个数值应是0。我们再输入不同的测试程序，好让我们对于问题所在有更清晰的了解：

```
> by. quantiles (c(1, 2, 3, 4), 4)
[1] 0.00 0.25 0.25 0.00 1.0
```

这里有两个问题：一是0 出现于1 之前，而数字并没有递增，原来是由于我们在 for 循环中用了 n-1 而不是 n。向量含有 n+1 个元素，因此第二最后元素应为 n。另一个问题是来自 pbreaks[i] = 1/n 一行。我们是想在这方程式的右边得到出现某处的 i 之数值。

更正 for 循环的执行范围应可解决第一个问题。同时，让我们尝试用 i/n 来代替 1/n 来解决第二个问题。

```
> by. quantiles (c(1, 2, 3, 4), 2)
[1]   0   1   1
> by. quantiles (c(1, 2, 3, 4), 4)
[1]   0.00   0.50   0.75   1.00   1.00
```

应该用 (i -1)/n 才对！更正这错误后，函数的这部分便可正常运作：

```
> by. quantiles
function (x, n)
{
    pbreaks = rep(0, n+1)
    pbreaks [n+1] = 1
```

```
    for(i in 2:n)
        pbreaks[i] = (i - 1)/n
    pbreaks
}
> by.quantiles(c(1, 2, 3, 4), 4)
[1]  0.00  0.25  0.50  0.75  1.00
```

下一步是利用函数 quantile 来计算数据中每个分位数的切点。输入 help(quantile) 可找到这函数的用法。为避免每次测试程序时都要重新输入数据,我们把数据储存为 sample。

```
> sample = c(.42, .15, .25, .27, .35, .38, .41, .1)
```

我们加入函数 quantile 来测试 sample。当 n=4 时,所传回的值应类似 (.1, .2, .3, .4, .5)。

```
> by.quantiles
function(x, n)
{
    pbreaks = rep(0, n+1)
    pbreaks[n+1] = 1
    for(i in 2:n)
        pbreaks[i] = (i - 1)/n
    dbreaks = quantile(x, probs = pbreaks)
    dbreaks
}
> by.quantiles(sample, 4)
    0%   25%   50%   75%   100%
   0.1 0.225  0.31 0.3875  0.42
```

这看似正确。下一步要利用 dbreaks 把数据以分位数写入。我们需要做的是核查 x 的每个元素是否在某两个切点之间,若是便归纳于相对应的分位。

开始时,我们先建立 n+1 个 0,然后利用 for 循环把它们逐个加

入。实际上，我们需要两个 for 循环：一个用来读取数据内的每个元素，另一个则把每个元素与分位数逐一比较。这即是说，就算该元素已被安置到正确的分位数组别，它亦会继续与其它分位数比较。其实也可编写一个较快的循环，在安置元素到正确的分位数组别之后，便终止运作。但编写这个函数较为复杂。

这两个 for 循环（一个根据 i，另一个根据 j）会含有一个 if 叙述，用以确定某元素是否属于第 j 个分位数组别。如果条件正确，该元素对应的 j 的值便会被记录。

```
> by. quantiles
function (x, n)
{
    pbreaks = rep(0, n+1)
    pbreaks [n+1] = 1
    for(i in 2:  n)
        pbreaks[i] = (i - 1)/n
    dbreaks = quantile (x, probs = pbreaks)
    data. by. quantiles = rep(0, n+1)
    for(i in 1:  (n+1))
        for (j in 1:  n)
            if (x[i] >= dbreaks[j] & x[i] < dbreaks[j+1])
            data. by. quantiles[i] = j
    data. by. quantiles
}1 > by. quantiles(sample, 4)
[1] 0 1 2 2 3
```

为何结果不是与原来的数据一样长呢？让我们利用 debug 来找寻 for 循环中的错处所在。（而且，为何 0 会出现于此？）

```
> debug(by. quantiles)
> by. quantiles(sample, 4)
debugging in:  by. quantiles(sample, 4)
debug: |
```

```
    pbreaks = rep(0, n + 1)
    pbreaks[n + 1] = 1
    for (i in 2: n) pbreaks[i] = (i - 1)/n
    dbreaks = quantile(x, probs = pbreaks)
    data. by. quantiles = rep(0, n + 1)
    for (i in 1: (n + 1)) for (j in 1: n) if (x[i] > = dbreaks[j] &
      x[i] < dbreaks[j + 1])
      data. by. quantiles[i] = j
    data. by. quantiles
}
Browse[1] > n
debug: pbreaks = rep(0, n + 1)
Browse[1] > n
debug: pbreaks[n + 1] = 1
Browse[1] > n
debug: for (i in 2: n) pbreaks[i] = (i - 1)/nlBrowse[1] > n
debug: dbreaks = quantile(x, probs = pbreaks)
Browse[1] > n
debug: data. by. quantiles = rep(0, n + 1)
Browse[1] > n
debug: for (i in 1: (n + 1)) for (j in 1: n) if (x[i] > = dbreaks[j]
&
    x[i] < dbreaks[j + 1]) data. by. quantiles[i] = j
Browse[1] > x
[1] 0. 42 0. 15 0. 25 0. 27 0. 35 0. 38 0. 41 0. 10
Browse[1] > dbreaks
    0%    25%    50%    75%    100%
0. 1000 0. 2250 0. 3100 0. 3875 0. 4200
Browse[1] > data. by. quantiles
[1] 0 0 0 0 0
Browse[1] > Q
```

向量 data. by. quantiles 的长度应与 x 相同，而不是与 dbreaks 的长度相同。这代表 i 应由 1 至 length(x)。更改这错误之后，结果看来稍为好些，但仍未能正常运作：

```
> by. quantiles (sample, 4)
[1] 0 1 2 2 3 3 4 1
```

让我们再次利用 debug 来查看错在何处。输出的结果应是(4, 1, 2, 2, 3, 3, 4, 1)，那么为何函数只会产生 0 而不是 4 呢？或许是由于输出结果从 0 开始吧。那么为何函数保留 0 而不产生 4 呢？

```
> by. quantiles(sample,4)
debugging in: by. quantiles(sample, 4)
debug: {
    pbreaks = rep(0, n + 1)
    pbreaks[n + 1] = 1
    for (i in 2: n) pbreaks[i] = (i - 1)/n
    dbreaks = quantile(x, probs = pbreaks)
    data. by. quantiles = rep(0, n + 1)
    for (i in 1: length(x)) for (j in 1: n) if (x[i] > = dbreaks[j]
&
      x[i] < dbreaks[j + 1])
      data. by. quantiles[i] = j
    data. by. quantiles
}
Browse[1] >
debug: pbreaks = rep(0, n + 1)
Browse[1] >
debug: pbreaks[n + 1] = 1
Browse[1] >
debug: for (i in 2: n) pbreaks[i] = (i - 1)/n
Browse[1] >
```

```
debug: dbreaks = quantile(x, probs = pbreaks)
Browse[1]>
debug: data.by.quantiles = rep(0, n + 1)
Browse[1]>
debug: for (i in 1: length(x)) for (j in 1: n) if (x[i] >= dbreaks
[j] &
    x[i] < dbreaks[j + 1]) data.by.quantiles[i] = j
Browse[1]> x[1]
[1] 0.42
Browse[1]> dbreaks[4]
   75%
0.3875
Browse[1]> dbreaks[5]
100%
0.42
```

从结果看来,第一个元素是大于或等于第75个百分位数,但不是少于第100个百分位数。为何会出错呢?原来测试应针对x[i]是否小于或等于(<=) dbreaks[j+1],而不是绝对小于(<) dbreaks[j+1]。把这错处更正后,函数便可如常运作:

```
function (x, n)
{
    pbreaks = rep (0, n+1)
    pbreaks [n+1] = 1
    for(i in 2 : n)
        pbreaks[i] = (i - 1)/n
    dbreaks = quantile (x, probs = pbreaks)
    data.by.quantiles = rep(0, length(x))
    for(i in 1 : length(x))
      for(j in 1 : n)
        if (x[i] >= dbreaks[j] & x[i] <= dbreaks[j+1])
```

```
            data. by. quantiles[ i ]  =  j
        data. by. quantiles
}
> by. quantiles( sample ,4 )
[1] 4 1 2 2 3 3 4 1
```

第五章

读取及整理数据

5.1 引言

在第二章里已示范过用 read. csv( )来读取逗号分隔（csv）的档案。本章会介绍各类型的数据档案、如何读取这些档案及整理数据的指令。

首先我们要明白储蓄数据档案的各种类型。差不多每种软件都有自己特定储蓄数据的格式，但也有两种既简单而又通用的格式：逗号分隔档案及文本文件。一般电子表格程序都可用这些格式储存数据。用这些格式储蓄可能会令数据失去一些电子表格特定的属性，但对于储蓄数据作数据分析来说，这些格式是最适合不过。这两种格式比较简单而且可以用一般编辑软件（如记事本）来输入及更改数据。

假设我们有某间公司客户的数据档案：cus1. csv 和 cus1. txt，分别是逗号分隔档案及文本文件格式，安放在 c:\Rdata 的资料夹内。

cus1. csv

```
name,sex,age
Alice,f,23
Boris,m,30
Carlo,f,19
David,m,21
Elisa,f,18
Flora,f,22
George,m,29
Helen,f,25
Ivan,m,31
Jack,m,27
Kevin,m,25
```

cus1. txt

name	sex	age
Alice	f	23
Boris	m	30
Carlo	f	19
David	m	21
Elisa	f	18
Flora	f	22
George	m	29
Helen	f	25
Ivan	m	31
Jack	m	27
Kevin	m	25

这些数据共有三列,分别代表姓名、性别及年龄,第一行是标签。csv 格式用逗号分隔每项数据,而 txt 则用空格分隔。

5.2 read. csv 指令

首先我们在 R 选单的 file - > change dir... 更改预设目录为 c:\Rdata,然后用以下 read. csv 指令读取数据:

```
> d = read. csv( "cus1. csv" )
> d
```

	name	sex	age
1	Alice	f	23
2	Boris	m	30
3	Carlo	f	19
4	David	m	21
5	Elisa	f	18
6	Flora	f	22
7	George	m	29
8	Helen	f	25

```
9     Ivan     m    31
10    Jack     m    27
11    Kevin    m    25
> names(d)
[1] "name" "sex" "age"
> mode(d)
[1] "list"
> dim(d)
[1] 11  3
```

由于 csv 档内的数据是用逗号分隔开,因此 R 可以知道每行有多少个数据。指令 names( ) 显示对象中的卷标 (label);mode( ) 显示对象的类型;dim(d) 显示对象维数的数据。由于 d 是清单,我们可以用 d $ age 去代表第三列的数据。

```
> d $ age
[1] 23 30 19 21 18 22 29 25 31 27 25
```

顺带一提,读者可用 help(read. csv) 去了解更多关于这指令讯息。特别是 header 的预设为 T,亦即是说,read. csv 预设第一行为标签。假如数据档案是无标签,使用者可加入 header = F 去表示档案无标签;第一行已是数据。此时,R 会自动加入 V1, V2, V3 等去代表第一列,第二列,第三列等。

问题:假如我们用 d = read. csv("cus1. csv",header = F) 去读取数据,d 会有什么特别之处? 又假设用 d = read. csv("cus1. txt") 去读取文字文件 cus1. txt,d 又有什么特别之处?

5.3 read. table 指令

read. table( ) 可以读取文字文件。文字文件内的数据是用空格隔开,因此 R 亦可以知道每行有多少个数据。还有,在 read. table( ) 中,header 的默认值为 F。因此,如果文字文件第一行是标签 (如 cus1. txt),记得加入自变量 header = T。

```
> d = read. table("cus1. txt",header = T)
> mode(d)
[1] "list"
> names(d)
[1] "name" "sex" "age"
> dim(d)
[1] 11  3
```

以上用 read. table 去读取文字文件 cus1. txt 跟用 read. csv 去读取 cus1. csv 完全一样。读者也可用 help(read. table) 去了解更多关于此指令的讯息。在 read. table 内有很多自变量,其中一项是 sep = ""。这是用来表示在档案内用来分隔数据的符号。明白此道理,我们甚至可以用 read. table 来读取 csv 档:

```
> d = read. table("cus1. csv",header = T,sep = ",")
```

档案的标签是可以更改的。以下指令将原本的卷标转成大写:

```
> names(d) = c("NAME","SEX","AGE")
> names(d)
[1] "NAME" "SEX" "AGE"
```

问题:假如我们只用 d = read. table("cus1. txt") 而忘记了加入 header = T, d 会有什么特别之处? 又假设用 d = read. table("cus1. csv")去读取文字文件 cus1. csv,d 又有什么特别之处?

5.4 scan 指令

无论用 read. csv 或 read. table 读取数据,数据必须是完整而无遗漏,数据及每一行数据的数目都一样。如果档案有遗漏,数据或每行数据的数目不一样(如第二章里面 singer. csv),csv 档(用 read. csv 读取)就会自动在空位填上 NA;txt 档(用 read. table 读取)就出现错误讯息。假设我们有 singers. txt(数据跟第二章里面 singer. csv 一样,只不过档案格式是 txt 而非 csv)。

```
> d = read. table("singers. txt",header = T)
```

```
Error in scan(file = file, what = what, sep = sep, quote = quote,
dec = dec, :
    line 21 did not have 4 elements
```

以上的错误讯息是指档案并非完整或每行的数目不一样。在这种情况下,我们应改用 scan 读取:

```
> height = scan("singers. txt",skip = T)
Read 130 items
> height
[1]  64 65 69 72 62 62 72 70 66 68 71 72 65 67 66 69 60 67 76 73 61 63 74
[24]  71 65 67 71 72 66 66 66 68 65 63 68 68 63 72 67 71 67 62 70 66 65 61
[70]  66 68 70 62 66 67 75 65 62 64 68 66 70 71 62 65 70 65 64 74 63 63 70
[93]  65 65 75 66 69 75 65 61 69 62 66 72 65 65 71 66 61 70 65 63 71 61 64
[116]  68 65 67 70 66 66 75 65 68 72 62 66 72 70 69
```

用 scan 指令可以成功地读取这些数据。skip = T 是指第一行是标签,我们要跳过第一行而从第二行开始读取数据(scan 是没有 header = T 这自变量的)。数据是逐行读入;height 是 130 位歌唱家的高度;最初 36 个是 Soprano(女高音);紧接的 35 个是 Alto(女低音);再紧接的 20 个是 Tenor(男高音);最后的 39 个是 Bass(男低音)。知道这些数据后,我们可以制造一个字符串向量去代表歌唱家的声部,然后用 cbind 合并及转为数据框:

```
> part = c(rep("Soprano",36),rep("Alto",35),rep("Tenor",20),
rep("Bass",39))
> singers = cbind(height,part)
> singers = data. frame(singers)
> singers $ height = height
```

5.5 清单及数据框的连系

若每次都要表示数据内的变量,都需要用符号 $。特别是当数据文件内有很多标签时,很快便会令人感到厌烦。但如果只输入变量名称,R 又不知应到哪里寻找。例如:

```
> d = read. csv( "cus1. csv" )
> d $ age
[1] 23 30 19 21 18 22 29 25 31 27 25
> age
Error: Object "age" not found
```

要解决这问题,我们可以将清单或数据框连系(attach)。连系之后,我们就可以用标签去表示变量;而无需每次都要输入清单或数据框名称及符号 $。例如:

```
> attach(d)
> age
[1] 23 30 19 21 18 22 29 25 31 27 25
```

使用者也可以用 detach( ) 去解除这连系。虽然对象的连系已被解除,但对象本身仍然存在。例如:

```
> detach(d)
> age
Error: Object "age" not found
> d $ age
[1] 23 30 19 21 18 22 29 25 31 27 25
```

5.6 数据整理

读取数据之后,我们很多时候要整理数据。在本节中,我们通过以下简单的例子去说明整理数据的指令。假设我们有以下两间公司客户数据的 csv 档案:

cus1. csv

```
name,sex,age
Alice,f,23
Boris,m,30
Carlo,f,19
David,m,21
Elisa,f,18
Flora,f,22
George,m,29
Helen,f,25
Ivan,m,31
Jack,m,27
Kevin,m,25
```

cus2. csv

```
name,sex,age
Alice,f,23
Alex,m,30
Boris,m,30
Betty,f,24
David,m,21
Doris,f,25
Henry,m,26
John,m,20
Peter,m,27
```

第一个数据档 cus1. csv 是我们曾在 5.1 节见过的客户数据；而 cus2. csv 的客户数据有些是在 cus1. csv 出现，亦有些是新的客户数据。我们就用这两个的数据档来说明整理数据的指令。虽然这两个数据档甚为简单，但以下介绍的指令同样可以应用到大型的数据库。首先我们用 read. csv 读取这两个数据档：

```
> d1 = read. csv("cus1. csv")
> d2 = read. csv("cus2. csv")
```

5.6.1 数据选取

假设我们想在 d1 内选取年龄小于 25 岁的客户数据，我们可以用以下指令：

```
> d1[d1 $ age < 25,]
   name    sex   age
1  Alice    f     23
3  Carlo    f     19
4  David    m     21
5   Elisa   f     18
6  Flora    f     22
```

我们通过研究指令的每一部分,可以更清楚了解指令的运作。首先 d1 $ age<25 会传回以下的逻辑向量:

```
> d1 $ age<25
[1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[10] FALSE FALSE
```

这逻辑向量代表 d1 $ age<25 的真假值。然后 d1[d1 $ age<25,] 根据这逻辑向量选取 d1 的数据。注意:在方括号内的逗号前面是行指标(row index),是代表选取哪--行;逗号后面是列指标(column index),是代表选取哪--列。如上述例子中,行指标是 d1 $ age<25 的逻辑向量,因此 R 会选取适合的行。而列指标留空,是代表所有列(name,sex,age)都选取。

问题:如何选取在 d1 的男性客户 name 及 age 的数据?

5.6.2 排序及秩(Sorting and Ranking)

假设想将 d2 的数据根据 age 由小至大排序,我们应怎么办?首先,R 的内置函数 sort() 只能将向量作简单排序(如 sort(d2 $ age)),但不能达到想要的结果。我们需要用 order() 函数去完成。首先指令 order(d2 $ age)是传回数字在整个向量里的排序:

```
> order(d2 $ age)
[1] 8 5 1 4 6 7 9 2 3
```

这就是说,第八行(John,m,20)最年轻;紧接是第五行(David,m,21)等等。我们就可以依照这排序选取记录:

```
> d2[order(d2 $ age),]
  name  sex  age
8 John    m   20
5 David   m   21
1 Alice   f   23
4 Betty   f   24
6 Doris   f   25
7 Henry   m   26
```

9 Peter m 27
2 Alex m 30
3 Boris m 30

跟 order 有密切关系的是秩（rank）。秩在非自变量统计中是不可或缺的。rank( ) 回传每个数字在整个向量中的秩。例如：

```
> rank(d2 $ age)
[1] 3.0 8.5 8.5 4.0 2.0 5.0 6.0 1.0 7.0
```

注意：第二及第三行的年龄相同，原本的秩 8 及 9 就变成 8.5。

问题：如何将 d1 的数据根据 age 由大至小排序？（提示：可用 help(order) 查看有没有其它自变量可用。）又如何将 d1 的数据用以下的方式排序：首先前面是女性，紧接是男性；然后分别在女及男的组别中以年龄由小至大排序？

5.6.3 配对（Matching）

在 d1 及 d2 里有些客户记录是相同。我们如何找出这些记录？答案就是应用 R 的内置二元算子（binary operator）“%in%”。x %in% y 就是转回逻辑向量，表示在 x 中的元素是否亦在 y 内出现。例如：

```
> 1:7 %in% 5:10
[1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

很明显，在 1 至 7 这七个数字中，只有最后三个会在向量 5:10 之中出现。我们利用以下指令，就可找出在 d1 而同时又在 d2 出现的记录：

```
> d1[d1 $ name %in% d2 $ name,]
   name sex age
1 Alice   f  23
2 Boris   m  30
4 David   m  21
```

注意：x %in% y 跟 y %in% x 不一样。

问题:如何找出在 d1 中但不在 d2 出现的客户记录?

5.6.4 重复记录 (Duplicated Records)

现在用 d = rbind(d1,d2) 将 d1 及 d2 合并成为 d,在 d 里有重复记录。如何找出这些重复记录? 内置函数 duplicated() 传回对象中重复记录的逻辑向量。我们就可利用以下指令找出重复记录:

```
> d[duplicated(-d $ name),]
    name   sex   age
12  Alice    f    23
31  Boris    m    30
51  David    m    21
```

问题:如何去除在 d 中的重复记录 (de - duplicate)?

5.7 应用

以上简单的例子只用作说明整理数据的指令,而这些指令一样可以应用到大型数据库。在数据挖掘 (Data Mining) 中有巨型的数据库。我们通常在数据库随机抽取或分割成训练数据 (training dataset) 及测试数据 (testing dataset)。为方便说明,我们产生 5000 个标准正态随机数字;然后转成一个 500x10 的数据矩阵。再从这数据矩阵随机抽取 400 行作为训练数据;而余下的 100 行作为测试数据。

```
> set. seed(12345)
> d = matrix(rnorm(5000),nc = 10)
> rl = sample(1:500,400)
> dl = d[rl,]
> dim(dl)
[1] 400  10
> d2 = d[ -rl,]
> dim(d2)
[1] 100  10
```

以上的指令已经介绍过,读者应该可以明白。d 是原本的数据库;rl

是从 1 至 500 中随机抽取 400 个整数；d1 是训练数据而 d2 是测试数据。唯一要稍加说明的是指令 d2 = d[-r1,]。它从 d 选取没有在 r1 出现的整数；亦既是抽取 d1 之后余下的 100 行。

5.8 遗漏数值（Missing Values）及完整记录（Complete Cases）

在日常遇到的真实数据中，有很多包含遗漏数值或不完整记录。前面已提及 R 以 NA 代表遗漏数值。但进行统计分析时，有很多统计方法是不容许有遗漏数值。最简单的方法是只选取完整记录；在每一行中，但凡有遗漏数值就整行不要。（这方法虽然简单，但有可能得到偏差样本（bias sample）。事实上有很多统计研究是针对如何有效处理遗漏数值；当然这些不在本书讨论范围。）

R 有内置函数 complete. cases( ) 去选取完整记录。承接上一节的例子，我们首先用 replace( ) 函数，将在 d 中的大于 2 数字变成 NA；然后示范用 complete. cases( ) 选出完整记录：

```
> d3 = replace(d,d > 2,NA)
> d4 = d3[complete. cases(d3),]
> dim(d4)
[1] 402  10
```

注意：complete. cases( ) 只传回逻辑向量，我们要利用它来选取完整记录。从以上的例子知道，在原本的 500 行中有 98 行有遗漏数值；因此完整记录 d4 只要 402 行。

问题：我们如何知道在全部 5000 个数字中有多少个 NA？NA 的数目又是否合理？尝试计算 d 及 d4 每列的平均数，比较一下它们有什么分别。

第六章

统计模型(一)

6.1 引言

这章主要介绍在R环境下怎样建立统计模型,例如线性回归模型(linear regression model)、方差分析(Analysis of Variance)、逻辑斯谛回归(logistic regression)等。假设读者已熟习R的基本运作并且对统计模型有一定的概念。

6.2 模型公式

所有在R内的统计模型公式都是由自变量(predictor/predictors)和因变量(response)所组成的,而格式是:因变量 ~ 自变量。符号 ~ 代表左边的数值被右边的模型来解释。在众自变量之间应以"+"来分隔,而交互作用(interaction)则以"*"代替。R的模型预设是包含截距项(intercept)。如要取消截距项,只要加入-1为其中一个自变量即可。

6.3 回归模型

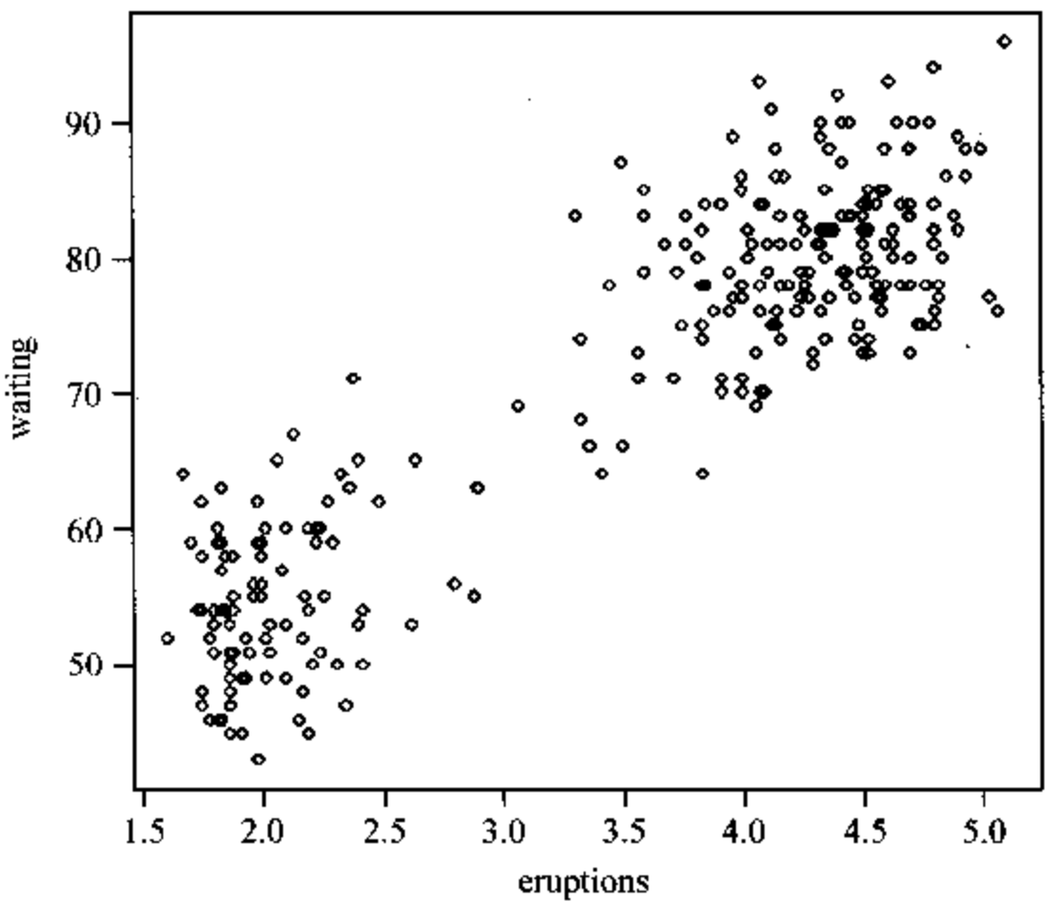
在R内置档案中,档案faithful载有一个地质学家于1978年8月与1979年8月在黄石公园旅游景点"Old Faithful"间歇泉所记录的喷发数据。

```
> data(faithful)
```

```
> attach(faithful)
> names(faithful)
[1] "eruptions" "waiting"
```

在这档案内包括两组向量,它们分别是泉水持续时间(eruptions)(以分钟计)及喷发相隔时间(waiting)(以分钟计)。

```
> plot(eruptions,waiting)
```



从图中可以观察出喷发相隔时间和喷发持续时间正相关(positively correlated),这代表若今次喷发持续时间愈长,那么下一次喷发会相距愈远。

我们先从单自变量的回归模型开始说明。模型 waiting ~ eruptions 代表着“用 eruptions 去解释 waiting”。函数 lm 是用来建立线性回归模型。R 的指令如下:

```
> lm(waiting ~ eruptions)
Call:
```

```
lm(formula = waiting ~ eruptions)
Coefficients:
(Intercept)      eruptions
  33.47            10.73
```

函数 lm 拟合一个线性回归模型：waiting = 33.47 + 10.73 eruptions，并建立了一个属于线性模型的对象，并传回各个自变量系数和其它不同的资料。因此使用者应该加以储存。

```
> faithful.lm = lm(waiting ~ eruptions)
```

利用函数 summary( ) 可以得出以下概要统计数据：

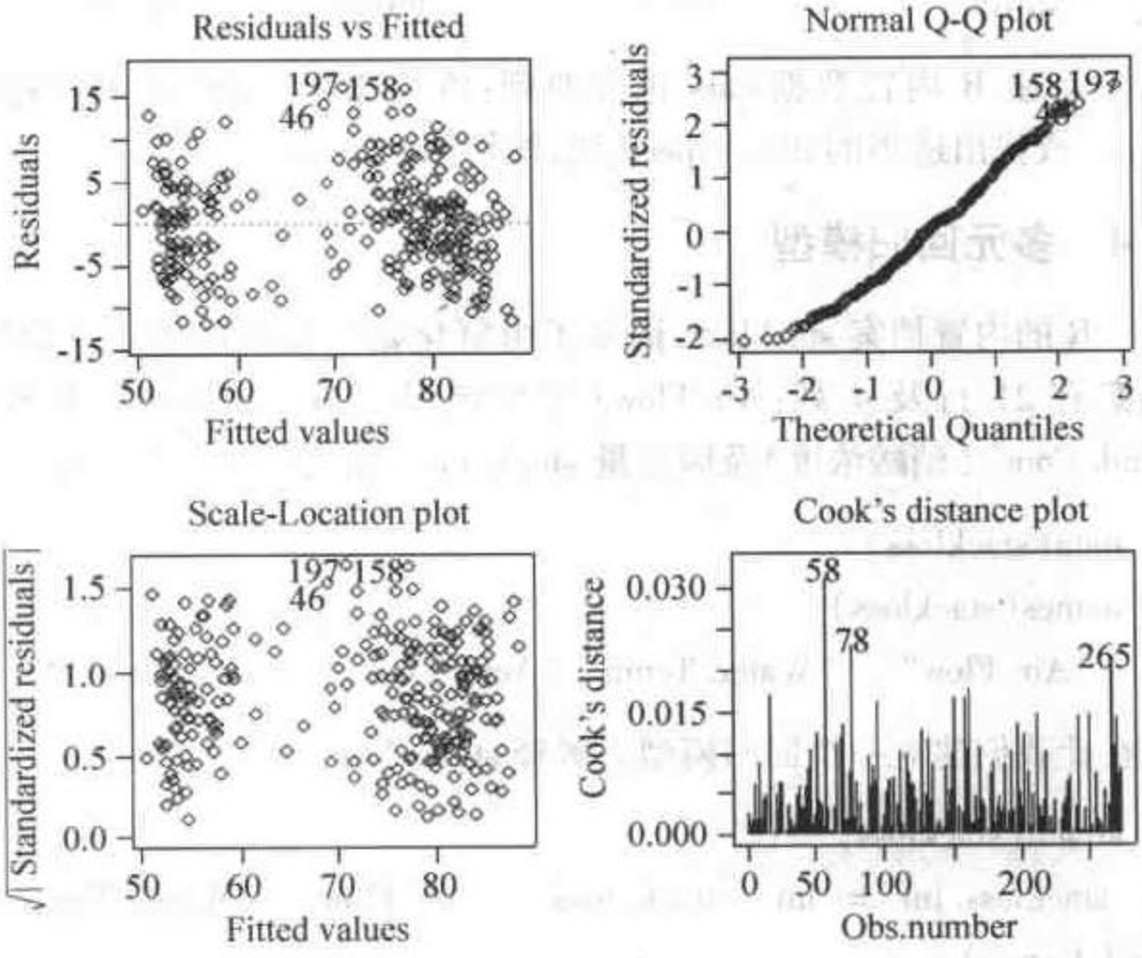
```
> summary(faithful.lm)

Call:
lm(formula = waiting ~ eruptions)

Residuals:
     Min       1Q   Median       3Q      Max
-12.0796  -4.4831   0.2122   3.9246  15.9719

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  33.4744     1.1549   28.98   <2e-16 ***
eruptions    10.7296     0.3148   34.09   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.914 on 270 degrees of freedom
Multiple R-Squared: 0.8115,     Adjusted R-squared: 0.8108
F-statistic:  1162 on 1 and 270 DF,  p-value: < 2.2e-16
```

指令 summary 提供了很多有用的数据，如 p - value 及 multiple R - Squared等。从以上结果发现，eruptions 是一个很好的自变量，适用于线性回归模型来解释 waiting。（指令 abline(lsfit(waiting, eruptions)) 会在 plot 的图像中加入以上之线性回归线，我们将会在下一

章详细介绍)。

图像指令 plot 亦可以对回归模型作诊断检验(diagnostic test)。因为 plot(faithful. lm) 会输出四组诊断图像,所以我们先用指令 par(mfrow = c(2,2)),使 R 预留适当空间,并将图像窗口分成四份(两行及两列)。关于 par( )指令及其它图像指令将会在下一章有详细介绍。

```
> par(mfrow = c(2,2))
> plot(faithful. lm)
```



指令 par(mfrow = c(2,2))可以如上述的将图像输出窗口分成四份(两行及两列)。只要输入 par(mfrow = c(1,1)),下一次图像输出窗口就可回复正常。

这四组图像显示一些有用的诊断信息(diagnostic information),例如残余图(residual plot),正态分位图(quantile normal plot),曲氏距离(Cook's distance plot)等。曲氏距离图将每一数据点对回归线的

影响力(influence)画出,数值愈大表示影响力愈大。

faithful. lm 内还包含很多有用的项目可供应用,如 coefficients(系数)、residuals(残余向量)及 fitted. values(拟合向量)等。我们可用 names(faithful. lm)去显示这些项目,help(lm) 亦有关于 lm( )及这些项目的说明。

```
> names(faithful. lm)
[1] "coefficients"      "residuals"     "effects"      "rank"
[5] "fitted. values"    "assign"        "qr"           "df. residual"
[9] "xlevels"           "call"          "terms"        "model"
```

问题:R 内置数据 cars 包含两列:汽车速度 speed 及刹车距离 dist。或找出适当的 dist ~ speed 线,线性回归模型。

6.4 多元回归模型

R 的内置档案 stackloss,记录了由氧化氨气而制造硝酸的数据。档案有 21 行及 4 列:Air. Flow(空气流量)、Water. Temp(水温)、Acid. Conc.(硝酸浓度)及因变量 stack. loss(氨气损失之百分比)。

```
> data(stackloss)
> names(stackloss)
[1] "Air. Flow"    "Water. Temp"  "Acid. Conc. "  "stack. loss"
```

现在让我们建立一个回归模型去解释 stack. loss:

```
> attach(stackloss)
> stackloss. lm = lm ( stack. loss ~ Air. Flow + Water. Temp + Acid. Conc. )
> summary(stackloss. lm)

Call:
lm(formula = stack. loss ~ Air. Flow + Water. Temp + Acid. Conc. )

Residuals:
     Min       1Q       Median      3Q      Max
```

-7.2377 -1.7117 -0.4551 2.3614 5.6978

Coefficients:

	Estimate	Std. Error	t value	Pr(> |t|)
(Intercept)	-39.9197	11.8960	-3.356	0.00375 **
Air. Flow	0.7156	0.1349	5.307	5.8e-05 ***
Water. Temp	1.2953	0.3680	3.520	0.00263 **
Acid. Conc.	-0.1521	0.1563	-0.973	0.34405

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

Residual standard error: 3.243 on 17 degrees of freedom
Multiple R-Squared: 0.9136, Adjusted R-squared: 0.8983
F-statistic: 59.9 on 3 and 17 DF, p-value: 3.016e-09

从以上的结果得知,多元线性回归模型为:

stack.loss = -39.9197 + 0.7156 Air.Flow + 1.2953 Water.Temp - 0.1521 Acid.Conc.

最后的一个 p-值很小(3.01e-09),是表示并非所有自变量都无用,但也并不是每一个自变量都是有用的。Acid.Conc. 的 p-值很高(0.344),因此,Acid.Conc. 应该被移除。最简单的方法是重新输入新的回归模型(更好的方法是利用光标进行修改):

```
> stackloss.lm = lm(stack.loss ~ Air.Flow + Water.Temp)
```

又或者可以用以下指令:

```
> stackloss.lm = update(stackloss.lm, . ~. - Acid.Conc.)
```

这两指令所得结果相同。在 update 指令内,"~"前面的"."表示因变量不变,而 - Acid.Conc. 则表示回归模型减少了 Acid.Conc.。无论用那种方法,我们应有以下结果:

```
> summary(stackloss.lm)
```

Call:

```
lm(formula = stack.loss ~ Air.Flow + Water.Temp)

Residuals:
     Min       1Q   Median       3Q      Max
-7.5290  -1.7505   0.1894   2.1156   5.6588

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -50.3588     5.1383  -9.801 1.22e-08 ***
Air.Flow      0.6712     0.1267   5.298 4.90e-05 ***
Water.Temp    1.2954     0.3675   3.525  0.00242 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.239 on 18 degrees of freedom
Multiple R-Squared: 0.9088,     Adjusted R-squared: 0.8986
F-statistic: 89.64 on 2 and 18 DF,  p-value: 4.382e-10
```

以上的结果显示,拟合的多元回归模型为:

stack.loss = -50.3588 + 0.6712 Air.Flow + 1.2954 Water.Temp

Air.Flow 及 Water.Temp 的 p-值很小,所以它们是有用的自变量;而 multiple R-squared 亦很高 (0.9088)。直至目前,这回归模型应是很理想。为了示范起见,我们尝试加入 Air.Flow 及 Water.Temp 的交互作用 (interaction):

```
> summary(lm(stack.loss ~ Air.Flow + Water.Temp + Air.Flow*Water.Temp))
```

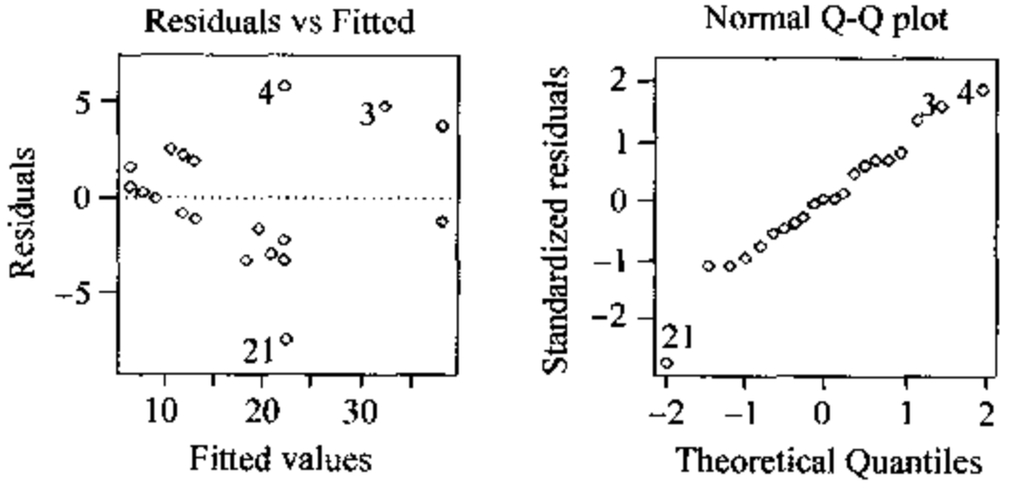
注意:我们不一定要储存 lm() 的输出,可以如上指令直接输出回归模型的概要。这样,原先的 stackloss.lm 就不会被改变。(另外的方法是用 model1, model2 等储存不同回归模型的输出)。

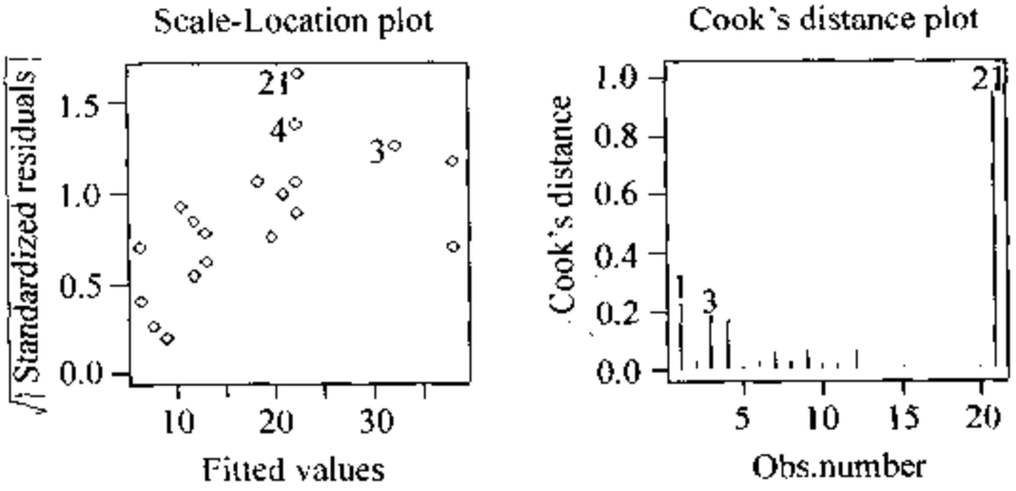
```
Call:
lm(formula = stack.loss ~ Air.Flow + Water.Temp + Air.Flow*
    Water.Temp)

Residuals:
```

```
    Min      1Q  Median      3Q     Max
-5.0695 -1.3591 -0.6886  2.2066  6.9717

Coefficients:
                            Estimate Std. Error t value Pr(>|t|)
(Intercept)                 22.29030   35.09602   0.635   0.5338
Air. Flow                   -0.51551    0.57984  -0.889   0.3864
Water. Temp                 -1.93006    1.58044  -1.221   0.2387
Air. Flow:Water. Temp        0.05176    0.02478   2.089   0.0521 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.973 on 17 degrees of freedom
Multiple R-Squared: 0.9274,     Adjusted R-squared: 0.9146
F-statistic: 72.38 on 3 and 17 DF,  p-value: 6.904e-10
```

以上的结果显示,加入了交互作用之后,自变量的 p 值反而变大。这回归模型并不适合,我们还是用回原先的回归模型好了。(幸好原先的 stackloss.lm 还未改变)。最后我们亦要对回归模型作诊断检验:

```
> par(mfrow=c(2,2))
> plot(stackloss.lm)
```



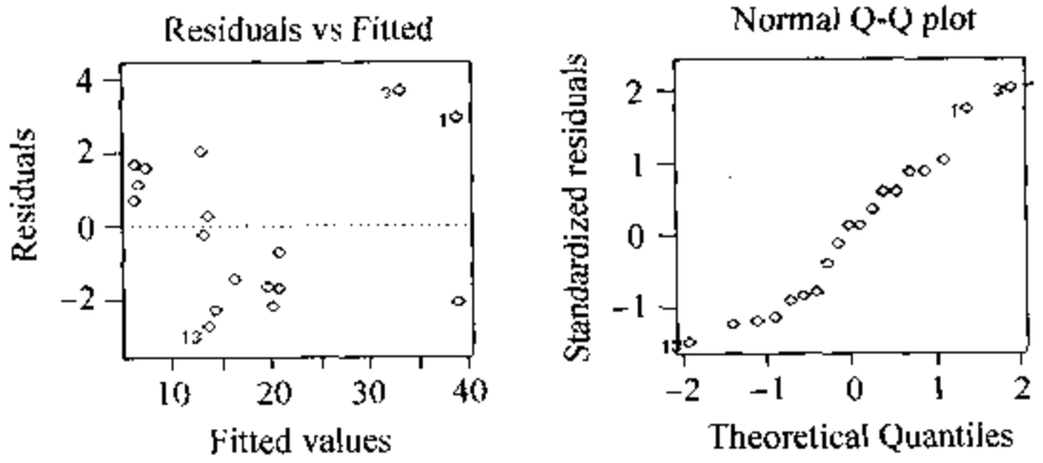


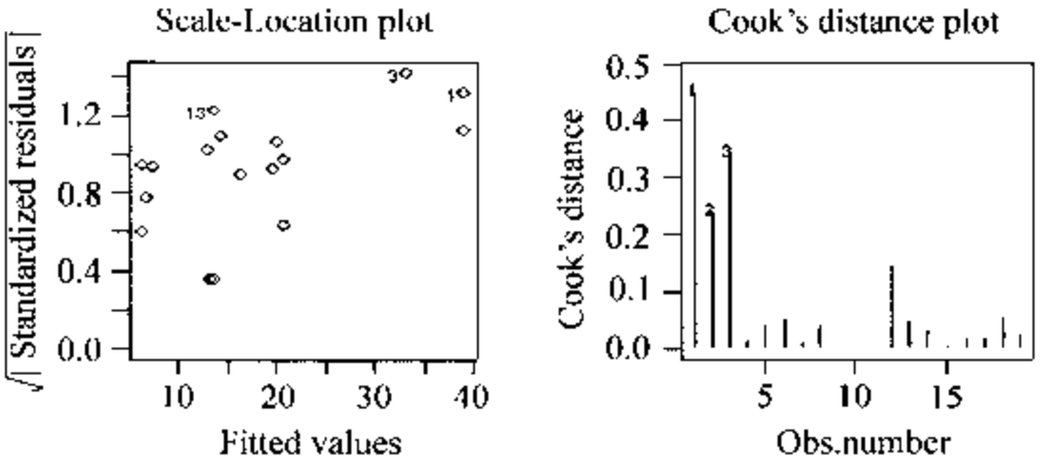
从以上图像发现，第 21 点是异常值（outlier）及对回归模型有很大的影响力（曲氏距离很大）。在这情况下，我们应该移除这一点。R 提供了以下一个方便的办法：

```
> stackloss.lm = lm(stack.loss ~ Air.Flow + Water.Temp + Acid.Conc., subset = -21)
```

如果再用 plot(stackloss.lm) 去看诊断图像的话，就会发现第 4 点跟第 21 点一样需要移除。

```
> stackloss.lm = lm(stack.loss ~ Air.Flow + Water.Temp + Acid.Conc., subset = c(-21, -4))
> plot(stackloss.lm)
```





移除第4点及第21点之后,诊断图像再无显著问题。

```
> summary(stackloss.lm)

Call:
lm(formula = stack.loss ~ Air.Flow + Water.Temp, subset = c(-
21, -4))

Residuals:
     Min       1Q   Median       3Q      Max
-2.7648  -1.6970   0.2352   1.3615   3.6282

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -49.9387     3.2154  -15.53 4.53e-11 ***
Air.Flow      0.9300     0.0922   10.09 2.44e-08 ***
Water.Temp    0.5424     0.2659    2.04   0.0582 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.013 on 16 degrees of freedom
Multiple R-Squared: 0.9668,     Adjusted R-squared: 0.9626
F-statistic: 232.7 on 2 and 16 DF,  p-value: 1.492e-12
```

最后,我们得到了以下的多元回归模型:

stack. loss = -49.9387 + 0.93 Air. Flow + 0.5424 Water. Temp

在绝大部分回归模型的教科书中都会有回归模型的方差分析表(ANOVA Table)。以下的指令可输出回归模型的方差分析表:

```
> anova(stackloss. lm)
Analysis of Variance Table

Response: stack. loss
             Df  Sum Sq Mean Sq  F value     Pr( >F)
Air. Flow     1 1868.12 1868.12 461.1398 3.188e-13 ***
Water. Temp   1   16.85   16.85   4.1605    0.05825 .
Residuals    16   64.82    4.05
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

6.5 方差分析模型(Analysis of Variance Model)

R 内置档案 PlantGrowth 记录了用不同肥料种植植物的重量。

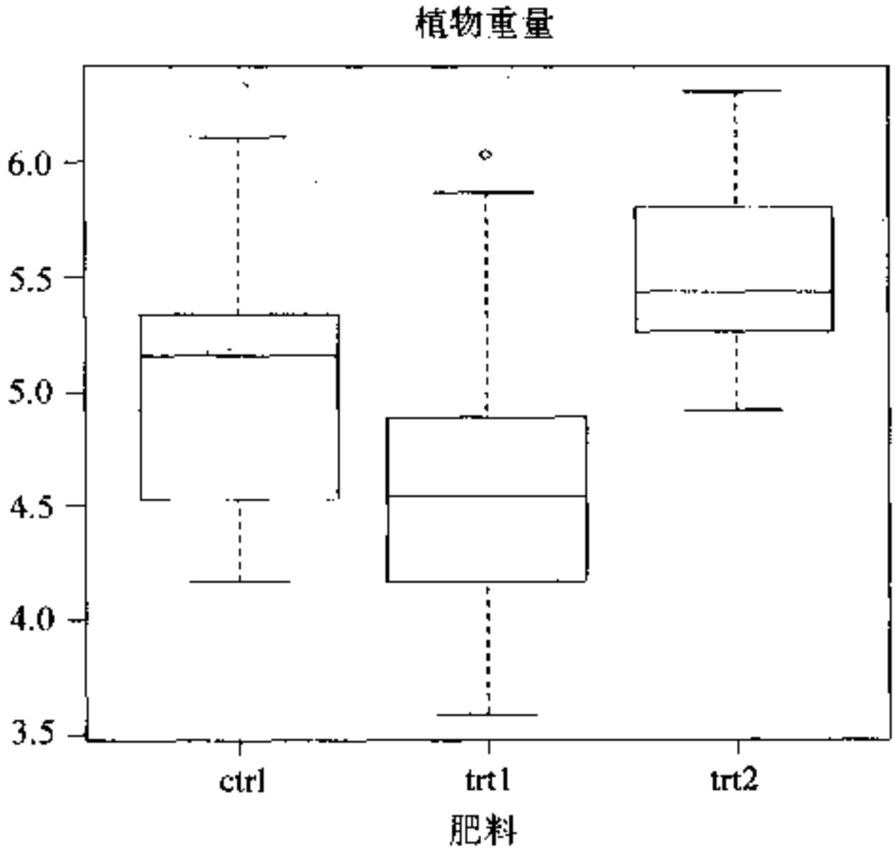
```
> data(PlantGrowth)
> attach(PlantGrowth)
> names(PlantGrowth)
[1] "weight" "group"
```

数据中有三个组别:控制组别(Control group)及以两种肥料种植。注意:我们先要将 group 转成因子(请参看 2.4 节)。

```
> group = as. factor(group)
```

首先用 plot 绘制以下盒形图(参看 2.8.2 节)。我们亦以此为例子,用方差分析来检验这三个组别植物的重量是否一样?

```
> plot(group,weight,main = "植物重量",xlab = "肥料")
```



以下是方差分析的指令,表示统计模型的语法基本上跟 lm( ) 相同。

```
> summary( aov( weight ~ group) )
             Df  Sum Sq Mean Sq F value  Pr( >F)
group         2  3.7663  1.8832  4.8461 0.01591 *
Residuals    27 10.4921  0.3886
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

由于方差分析表中的 p - 值很小 (0.01591),所以这三个组别植物的重量有显著分别。

6.6 广义线性模型(Generalized Linear Models)

广义线性模型和回归模型有很多地方都十分相似。指令 glm( ) 可以建立广义线性模型。定义模型公式跟 lm( ) 十分相似,也可使用自变量如 subset = 及进行诊断检验等,但同时亦加入不少新的自变

量。最重要的自变量是 family = ,它是用作指定广义线性模型内之连接函数(link function);如 family = poisson 或 family = binomial 等。默认值为 family = gaussian,这就相当于 lm( ),可以用来建立线性回归模型。关于 glm( ) 各自变量的详细数据,可看 help(glm) 。至于广义线性模型的理论部分,可参考 Gernealized Linear Models 一书(作者:McCullagh and Nelder) 。由于篇幅所限,我们只以 logistic regression (family = binomial)为例子。熟识广义线性模型的读者可自行探索其它模型如 Poisson regression 等。

6.6.1 Binomial 模型(Logistic Regression)

在线性回归模型里,因变量是连续的。但在很多情况下,因变量是二元的。例如:是否痊愈、是否成功、是否生还等等。以下的例子是病人随机接受两种药物 (A 或 B) 或安慰药(placebo)的治疗。

数据中包含四列变量:性别(Sex),治疗方法(Treat),痊愈人数(Cured)及未能痊愈人数(NotCured)。

	Sex	Treat	Cured	NotCured
1	F	Placebo	40	43
2	F	Drug A	5	7
3	F	Drug B	26	32
4	M	Placebo	11	6
5	M	Drug A	48	20
6	M	Drug B	52	20

由于数据并非很大,我们大可以由键盘输入。(这亦是一个很好的练习,使用者可参考第二章,提示:可利用 data. frame 指令)。假设我们已有以下的数据框:

```
> attach(neuro)
> neuro
  Sex Treatment Cured NotCured
1   F   Placebo    40       43
2   F         A     5        7
3   F         B    26       32
```

4	M	Placebo	11	6
5	M	A	48	20
6	M	B	52	20

首先,我们必须将 Sex 及 Treatment 转为因子;Cured 及 NotCured 转为数值向量:

```
> neuro $ Sex = as. factor(neuro $ Sex)
> neuro $ Treatment = as. factor(neuro $ Treatment)
> neuro $ Cured = as. numeric(neuro $ Cured)
> neuro $ NotCured = as. numeric(neuro $ NotCured)
```

利用 glm( )来拟合 logistic regression,它的因变量有两列:成功次数及失败次数。因此,利用指令 cbind 合并 Cured 和 NotCured 为应变量,并将它们编入模型内:

```
> anova(glm(cbind(Cured, NotCured) ~ Sex + Treatment, family = binomial))
```

```
Analysis of Deviance Table
Binomial model
Response: cbind(Cured, NotCured)
Terms added sequentially (first to last)
```

	Df	Deviance	Resid. Df	Resid. Dev
NULL			5	19.71457
Sex	1	19.07467	4	0.63990
Treatment	2	0.04528	2	0.59462

这显示出两种药物和安慰剂并没有效用上的差别。反之,男性比女性有多于30%的痊愈机会。

在本节中,我们只用 logistic regression (family = binomial)为例。事实上,glm 包含很多连接函数。有兴趣的读者可以用 help(family)去查看所有的广义线性模型。当然,要应用这些模型,读者一定要对广义线性模型的理论有一定的了解。

第七章

统计模型(二)

7.1 引言

在上一章里介绍了线性回归模型、方差分析及广义线性模型。这章主要介绍其它常用的统计模型。在 R 软件中很多程序馆(library)里面有很多函数。每当开启 R 时,R 会自动加载 base 及 stats 的程序馆。例如我们上一章提及的 lm()、aov()、glm() 及以前提及过的内置函数等等,都可在这两个程序馆找到。好奇的读者可键入指令 library(help = stats) 或 library(help = base) 查看程序馆内的函数。

以下介绍的统计模型及方法不一定可以在 stats 或 base 找到。建立这些统计模型时先要加载特定的程序馆。有需要时,我们会说明如何加载适当的程序馆。

7.2 线性判别分析(Linear Discriminant Analysis)

R 内置档案中有著名的鸢尾花(iris)数据档案。此档案内有五列:Sepal. Length(花萼长度),Sepal. Width(花萼阔度),Petal. Length(花瓣长度),Petal. Width(花瓣阔度)及 Species(品种)。档案内共有三种品种:setosa,versicolor 及 virginica。每种品种各有 50 行;即档案内共有 150 行。首先我们读取 iris 档案:

```
> data(iris)
```

```
> attach(iris)
> names(iris)
[1] "Sepal. Length" "Sepal. Width" "Petal. Length" "Petal. Width"
  "Species"
```

现在就用线性判别分析,用花萼及花瓣的长和阔,来辨别鸢尾花的品种。在 MASS (注意大写)的程序馆内的 lda( ) 函数就是这线性判别分析。

```
> library(MASS)
> iris. lda = lda(Species ~ Sepal. Length + Sepal. Width + Petal. Length
+ Petal. Width)
```

(或者可以输入 iris. lda = lda(iris[,1 :4] ,iris[,5])

首先我们要用指令 library(MASS) 加载 MASS 程序馆,否则找不到 lda( ) 函数。然后在 lda( )内设定统计模型。第一种方法的语法跟 lm( ) 一样;第二种方法是将预测量放在前面,因变量放后面。无论用哪一种方法,我们可得到以下结果:

```
> iris. lda
Call:
lda(iris[ , 1 :4] , iris[ , 5])

Prior probabilities of groups:
    setosa versicolor  virginica
0. 3333333  0. 3333333  0. 3333333
Group means:
           Sepal. Length Sepal. Width Petal. Length Petal. Width
setosa         5. 006       3. 428       1. 462       0. 246
versicolor     5. 936       2. 770       4. 260       1. 326
virginica      6. 588       2. 974       5. 552       2. 026

Coefficients of linear discriminants:
                    LD1          LD2
Sepal. Length  0. 8293776  0. 02410215
```

Sepal. Width	1.5344731	2.16452123
Petal. Length	-2.2012117	-0.93192121
Petal. Width	-2.8104603	2.83918785

Proportion of trace:

LD1	LD2
0.9912	0.0088

以上包括了每组的平均向量,线性判别函数的系数等等。读者可用 help(lda)查看 lda 的自变量,包括 CV (Cross Validation,互相证实)及 prior (设定先验概率) 等。至于有关线性判别的理论,读者可参考多元统计分析的书籍。

值得一提,R 有内置函数 predict(),可以将 lda() 的输出应用于原本 iris 的数据进行预测。从而对比真正的品种,看看误差率有多少。

```
> iris.id = predict(iris.lda) $ class
> table(iris.pred,Species)
```

	Species		
iris.id	setosa	versicolor	virginica
setosa	50	0	0
versicolor	0	48	1
virginica	0	2	49

在 predict() 函数输出中 class 这一项,储存预测的品种。我们应用了 table()去列表预测与真正品种的比较。从以上得知,在 150 个数据中,只有 3 个错误,误差率为 2%。而有两朵 versicolor 的鸢尾花被误认为 virginica;一朵 virginica 的鸢尾花被误认为 versicolor。

7.3 聚类分析 (Cluster Analysis)

在上一节 iris 的例子中,我们知道每朵鸢尾花的品种并应用了这些数据。假设我们只知道数据内有三种品种而不知道每朵花的真正品种,就只凭花萼及花瓣的长度和阔度去分成三组,这就是聚类分析。聚类分析比判别分析更困难,误差率亦较高。聚类分析通常有

两种:分层方法(hierarchical method)和非分层方法(non-hierarchical method)。R 有内置函数 hclust( ) 及 kmeans( ),分别是分层及非分层的聚类分析方法。无论用分层或非分层的方法,都需要计算每对数据之间之距离或非相似性(dissimilarity)。R 有内置函数 dist( ) 去计算每对数据之距离矩阵(distance matrix)。

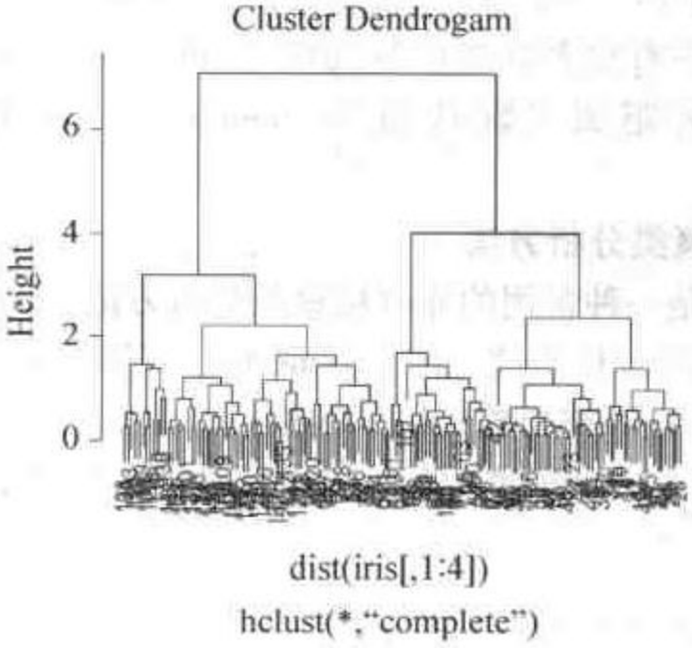
我们仍以 iris 为例子,说明这两种聚类分析方法。

7.3.1 分层聚类分析方法

如上一节一样,首先读取 iris 数据:

```
> data(iris)
> attach(iris)
> iris.hc = hclust(dist(iris[,1:4]))
```

hclust( ) 就是 R 的分层聚类分析函数,它的输入项是距离矩阵。dist( ) 正好计算 iris 的距离矩阵。(量度两个数据向量距离的定义也有很多,dist( ) 预设为欧基里德型(Euclidean)。关于 dist( ) 其它的选项可参考 help(dist))。还有,我们只用 iris 最前的四列数据,亦即是鸢尾花萼及花瓣的长和阔。hclust( ) 的输出储存在 iris.hc 内。对于分层聚类分析,我们通常可绘制树枝型分类图(dendrogram)。指令 plot(iris.hc) 绘制以下的树枝型分类图:



这树枝型分类图主要是首先将两相似（距离最短）的数据向量以线相连并视作为一点，然后继续相连，直至所有数据（共 150 点）连在一起。由于总共有 150 点之多，我们无法从树枝型分类图将数据分类。（这亦是分层聚类分析的缺点，当数据点太多时，树枝型分类图无法清楚列出每一点分类）。不过，内置函数 cutree() 可以利用 hclust() 的输出，将数据点编制若干组，每一数据点都附上有组别的标签。

```
> iris.id = cutree(iris.hc,3)
> table(iris.id,Species)
        Species
iris.id  setosa  versicolor  virginica
      1      50           0          0
      2       0          23         49
      3       0          27          1
```

以上指令就是根据 hclust 的输出：iris.hc，我们将 iris 编成三组，标签（1,2 或 3）储存在 iris.id。然后我们将 iris.id 与真正的品种 Species 作比较。从以上的表格来看，卷标 1 是 setosa，2 是 virginica（因为在这一组中有较多为 virginica），3 是 versicolor。而在这 150 朵鸢尾花之中，有 24（=23+1）朵花被误认，误差率为 16%。

hclust() 亦有很多自变量及选项，如可以有不同的方法去定义两点或两组的距离（默认值为 complete）。详情可参看 help(hclust)。

7.3.2 非分层聚类分析方法

k-mean 是一种常用的非分层聚类分析方法。kmeans() 就是 R 的 k-mean 聚类分析函数。注意：kmeans() 的输入是数据矩阵而非距离矩阵，而使用者亦要输入组别的数目。以下的 kmeans() 指令将 iris 数据分成三组（我们亦只用 iris 前四列数据），结果储存于iris.km 内：

```
> iris.km = kmeans(iris[,1:4],3)
> names(iris.km)
```

```
[1] "cluster"   "centers"   "withinss"  "size"
```

iris. km 内有四个项目,其中 cluster(1,2,或 3) 就是每朵花的标签。我们亦不妨将 cluster 与真正的品种 Species 作比较:

```
> table(iris. km $ cluster,Species)
   Species
     setosa   versicolor   virginica
1      50         0           0
2       0        48          14
3       0         2          36
```

从以上结果来看,误差率为 16/150 = 10.67%,较 hclust( ) 为好。

7.4 分类树 (Classification Tree)

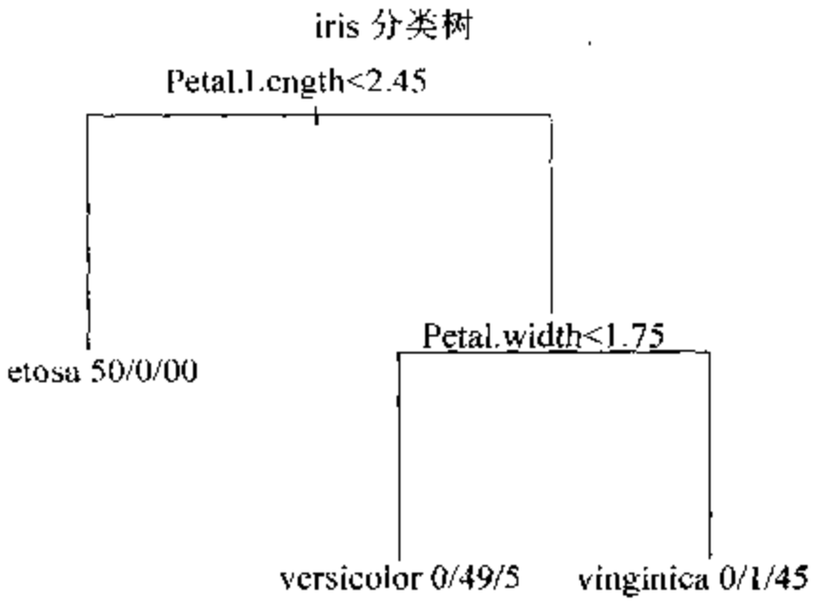
在上节提及的聚类分析,虽然可以将数据分门别类,但未能提供数据分类的法则。分类树就是一个既可将数据分类,亦可提供分类法则的一个方法。不过,分类树就像判别分析一样,是需要用数据类别的数据;而并非像聚类分析,不需数据类别的数据。R 的 rpart (代表 Recursive Partitioning) 程序馆中有 rpart( )函数可建立分类树。

```
> library(rpart)
> iris. ct = rpart( Species ~ Sepal. Length + Sepal. Width + Petal. Length
+ Petal. Width,method = "class" )
```

我们记得先要加载 rpart 程序馆。在 rpart( ) 内设定模型与 lm( ) 或 lda( ) 相似。method = "class"是一个重要的选项。因此 Species 是范畴性 (Categorical),我们就用 class。还有其它的选项如: method = anova, poisson, exp 等等,读者可参照 help(rpart) 。储存结果后可用以下指令绘制树型图:

```
> plot(iris. ct,asp =2,main = "iris 分类树" )
> text(iris. ct,use. n = T,cex =0. 6)
```

在 plot( ) 内的 asp = 2（模样比例（aspect ratio））及 text( ) 内的 cex = 0.6（文字放大倍数（character expansion factor））是用来调校树型图的比例，可以使树型图及文字有较好的配合。这些数字通常用尝试几次才能得到较好的比例。use. n = T 是显示终点（terminal node）的数据。



以上的树型图提供了一个简易的分类法则：

如果 Petal. Length < 2.45，Species = setosa (50/0/0)；

否则，如果 Petal. Width < 1.75，Species = versicolor (0/49/5)；

否则 Species = virginica (0/1/45)。

在每个终点的三个数字是这三种鸢尾花的分布（第一个数字是setosa的数目，第二个是 versicolor，第三个是 virginica）。用第一条法则，50 朵 setosa 花可完全正确地辨认。用第二条法则，有 5 朵 virginica 花误认为 versicolor。用第三条法则，有 1 朵 versicolor 花被误认为 virginica。误差率为 6/150 = 4%。我们也可以用指令 print(iris. ct) 显示较详细结果：

```
> print( iris. ct)
n = 150
node), split, n, loss, yval, (yprob)
      * denotes terminal node
```

1) root 150 100 setosa (0.33333333 0.33333333 0.33333333)
2) Petal. Length < 2.45 50 0 setosa (1.00000000 0.00000000 0.00000000) *
3) Petal. Length > = 2.45 100 50 versicolor (0.00000000 0.50000000 0.50000000)
6) Petal. Width < 1.75 54 5 versicolor (0.00000000 0.90740741 0.09259259) *
7) Petal. Width > = 1.75 46 1 virginica (0.00000000 0.02173913 0.97826087) *

假如我们想表列分类树的结果与真正品种,我们先要制造每朵花的分类标签。

```
> prob = predict(iris. ct)
> iris. id = 1 * (prob[ ,1] > 1/3) + 2 * (prob[ ,2] > 1/3) + 3 * (prob
[ ,3] > 1/3)
> table(iris. id,Species)
     Species
iris. id   setosa   versicolor   virginica
     1     50          0           0
     2      0         49           5
     3      0          1          45
```

首先,predict(iris. ct) 回传三列概率向量,分别代表这朵花属于 setosa,versicolor 及 virginica 的机会。最自然不过的就是哪一个概率大于 1/3,我们就将这朵花编入这概率代表的组别。iris. id 就是我们所需的标签。最后,用 table() 表列 iris. id 与真正品种。这结果与树型图所显示的结果一致。

问题:试用 kmean 聚类分析将 iris 数据分成三组,然后用 kmean 得到的标签作为训练数据,用分类树找出判别这三组的法则。

7.5 人工神经网络 (Artificial Neural Network)

除了线性判别分析 lda() 及分类树 rpart()外,R 亦有人工神经

网络 nnet( ) 去将数据分类。(人工神经网络是模拟大脑内的神经网络学习模式) 。首先使用者设定人工神经网络的自变量,然后以数据去训练网络,直至误差率下降至可接受水平为止。人工神经网络的自变量有输入项数目、输出项数目、隐蔽层 (Hidden layer) 数目及隐蔽层内神经元 (Neuron)数目。再以 iris 为例,输入数目为 4,输出数目为 1。为简单起见,设定只有一层隐蔽层及隐蔽层内只有两个神经元,我们就有如下图 4 -2 -1 的人工神经网络:

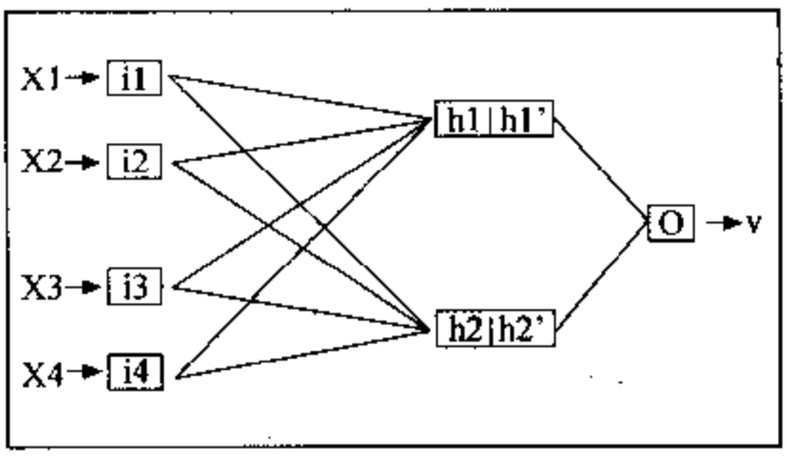


图 4 -2 -1　人工神经网络

上图中 i1,. . . ,i4 代表输入,o 代表输出,h1,h2 代表隐蔽层的神经元。每一条连接线代表一个自变量。我们先要加载 nnet 程序馆才可使用 nnet( ) 函数。Species 亦要转为数值而非范畴性。这样,输出是一个实数。

```
> library(nnet)
> sp = as. numeric(Species)
> iris. nn = nnet(sp ~ Sepal. Length + Sepal. Width + Petal. Length + Petal. Width, size = 2, linout = T)
# weights:  13
initial  value 1037. 739516
iter  10 value 49. 670777
iter  20 value 9. 504911
iter  30 value 7. 409853
```

```
iter       40 value 6.617702
iter       50 value 5.761859
iter       60 value 5.732296
iter       70 value 5.647696
iter       80 value 5.295863
iter       90 value 4.862270
final      value 4.847189
converged
```

注意：nnet( ) 开始的自变量值会不同，读者执行这些指令时可能会得到不同结果。我们亦可以用 summary(iris. nn) 去显示更详细的结果：

```
> summary( iris. nn)
a  4 - 2 - 1  network  with  13  weights
options       were       - linear      output       units
b - > h1     i1 - > h1    i2 - > h1    i3 - > h1    i4 - > h1
  - 8.26       - 41.74      - 28.43      - 12.44      - 1.95
b - > h2     i1 - > h2    i2 - > h2    i3 - > h2    i4 - > h2
  - 1.27       - 0.32       - 0.71        0.71         1.44
 b - > o      h1 - > o     h2 - > o
   0.96       - 37.96        2.47
```

以上是这个人工神经网络经训练之后的自变量值。这些自变量代表了以下一组的方程式：

$$h_1 = -8.26 - 41.74x_1 - 28.43x_2 - 12.44x_3 - 1.95x_4$$

$$h_2 = -1.27 - 0.32x_1 - 0.71x_2 + 0.71x_3 + 1.44x_4$$

$$h_1 = \exp(h_1)/[1 + \exp(h_1)] \quad h_2 = \exp(h_2)[1 + \exp(h_2)]$$

$$v = 0.96 - 37.96h_1 + 2.47h_2$$

以上第一及第二条式是连接输入与 h1，h2 的模型，第三条式是神经元的激活函数（activation function）。最后一条是连接神经元与输出的模型。人工神经网络的输出储存在 iris. nn 的 fitted. values。它是

一个实数的向量,数值接近 1 代表这朵花是 setosa;2 代表 versicolor;3 代表 virginica。我们只需用 round( ) 将 fitted. values 转化成最近的整数,这就是每朵花的标签。

```
> iris. id = round( iris. nn $ fitted. values)
> table( iris. id, Species)
      Species
iris. id    setosa    versicolor    virginica
      1      50          0            0
      2       0         46            0
      3       0          1           50
```

从以上结果得知,误差率为 4/150 = 2.67%。

7.6 程序馆(Library)

到目前为止,我们已介绍过不少统计模型、函数及程序馆。事实上,R 还有很多常用的函数及程序馆可供使用。例如:factanal( )(因子分析 factor analysis)、princomp( )(主成分分析 principal component analysis)、cancor( )(经典相关分析 canonical correlation analysis)、arima( )(时间序列分析 time series)、density( )(核心密度估计 kernal density estimation)、nls( )(非线性最小平方 nonlinear least squares)、optim( )(最优化方法 optimization) 等等。这些函数放置于 stats 程序馆内。读者可键入 help(函数名称) 便可查看有关该函数之用法。读者亦可键入指令 library(help = stats) 查看 stats 程序馆内包括哪些函数。

其实 stats 程序馆只是 R 内众多程序馆之一,只不过当每次开启 R 时被自动加载。使用者可随时加载其他程序馆,如以上提及之 MASS, rpart, nnet 等等。如果想查看所有程序馆,可键入 library( ):

```
base        The R Base Package
boot        Bootstrap R (S - Plus) Functions (Canty)
class       Functions for Classification
cluster     Functions for clustering (by Rousseeuw et al. )
```

ctest	Defunct Package for Classical Tests
eda	Defunct Package for Exploratory Data Analysis
foreign	Read data stored by Minitab, S, SAS, SPSS, Stata, ...
graphics	The R Graphics Package
grid	The Grid Graphics Package
KernSmooth	Functions for kernel smoothing for Wand & Jones (1995)
lattice	Lattice Graphics
lqs	Resistant Regression and Covariance Estimation
MASS	Main Package of Venables and Ripley's MASS
methods	Formal Methods and Classes
mgcv	Multiple smoothing parameter estimation and GAMs by GCV
mle	Defunct package for maximum likelihood estimation
modreg	Defunct Package for Modern Regression: Smoothing and Local Methods
mva	Defunct Package for Classical Multivariate Analysis
nlme	Linear and nonlinear mixed effects models
nls	Defunct Package for Nonlinear Regression
nnet	Feed – forward Neural Networks and Multinomial Log – Linear Models
rpart	Recursive Partitioning
spatial	Functions for Kriging and Point Pattern Analysis
splines	Regression Spline Functions and Classes
stats	The R Stats Package
stats4	Statistical functions using S4 classes
stepfun	Defunct Package for Step Functions, incl. Empirical Distributions
survival	Survival analysis, including penalised likelihood.
tcltk	Tcl/Tk Interface
tools	Tools for Package Development
ts	Defunct Package of Time Series Functions
utils	The R Utils Package

以上包括程序馆的名称及其简单说明。例如 survival 是有关生存分析，spatial 是有关空间统计方法等等。假如读者想进一步知道 survival 程序馆的内容，可用指令 library(help = survival)。这样，R 会列出 survival 程序馆内的函数。例如读者会发现函数 coxph()（Cox's Proportional Hazards Regression 比例危险率回归）。通过应用 help() 和对 R 的基本概念及操作有一定了解，读者可自行学习及探索（甚至自行开发）很多不同类型的程序馆及函数。能够使读者掌握 R 的基本概念及操作及可以自行学习及探索 R 的程序馆及函数，亦是本书一大目的。

最后，R 的程序馆不只限于以上所列。很多统计学者自行开发程序馆及函数。本着自由软件（freeware）的精神，上载 R 的官方网站（www.r-project.org）免费供人使用。例如笔者想应用 GARCH 模型（一个在金融时间序列常用的模型），发觉以上的程序馆并未提供。但在官方网站的 packages 中找到了 tseries.zip 的压缩档案。下载及解压至 R 的 library 数据夹内，R 就多了一个 tseries 的程序馆。用指令 library(tseries)加载后，就可以应用 garch() 函数。

我们以内置数据 EuStockMarkets 为例。档案内有四种欧洲股票市场的指数。我们先读取数据，然后将其中的法兰克福 DAX 指数转成时间序列：

```
> data(EuStockMarkets)
> d = data.frame(EuStockMarkets)
> names(d)
[1] "DAX" "SMI" "CAC" "FTSE"
> t = as.ts(d $ DAX)
```

然后计算指数的相对变动（Relative change）：$u_i = (t_i - t_{i-1})/t_{i-1}$。加载 tseries 程序馆，用程序馆内的 garch() 函数去拟合序列 u_i 的波幅（Volatility）。

```
> u = (lag(t) - t)/t
> library(tseries)
> u.gh = garch(u,order = c(1,1))
```

```
> round(u.gh $ coef,6)
     a0          a1          b1
0.000004 0.067580 0.892882
```

我们用 GARCH(1,1) 去拟合 u 序列。根据以上结果,DAX 指数的方差率为(Variance rate)为:

$$\sigma_n^2 = 0.00004 + 0.06758u_{n-1}^2 + 0.892882\sigma_{n-1}^2$$

有关 GARCH 的理论,详情可参考《时间序列与金融数据分析》(陈毅恒著,黄长全译. 中国统计出版社)一书。

第八章

制造图像

8.1 引言

以往各章节内，在配合所介绍的函数时，亦有介绍 R 的制图功能。这章会较有系统及深入介绍怎样在 R 环境下制造和修饰图像。使用者应在阅读这章前先熟习 R 的基本运作和数据框的使用方法。

8.2 Old Faithful 喷泉

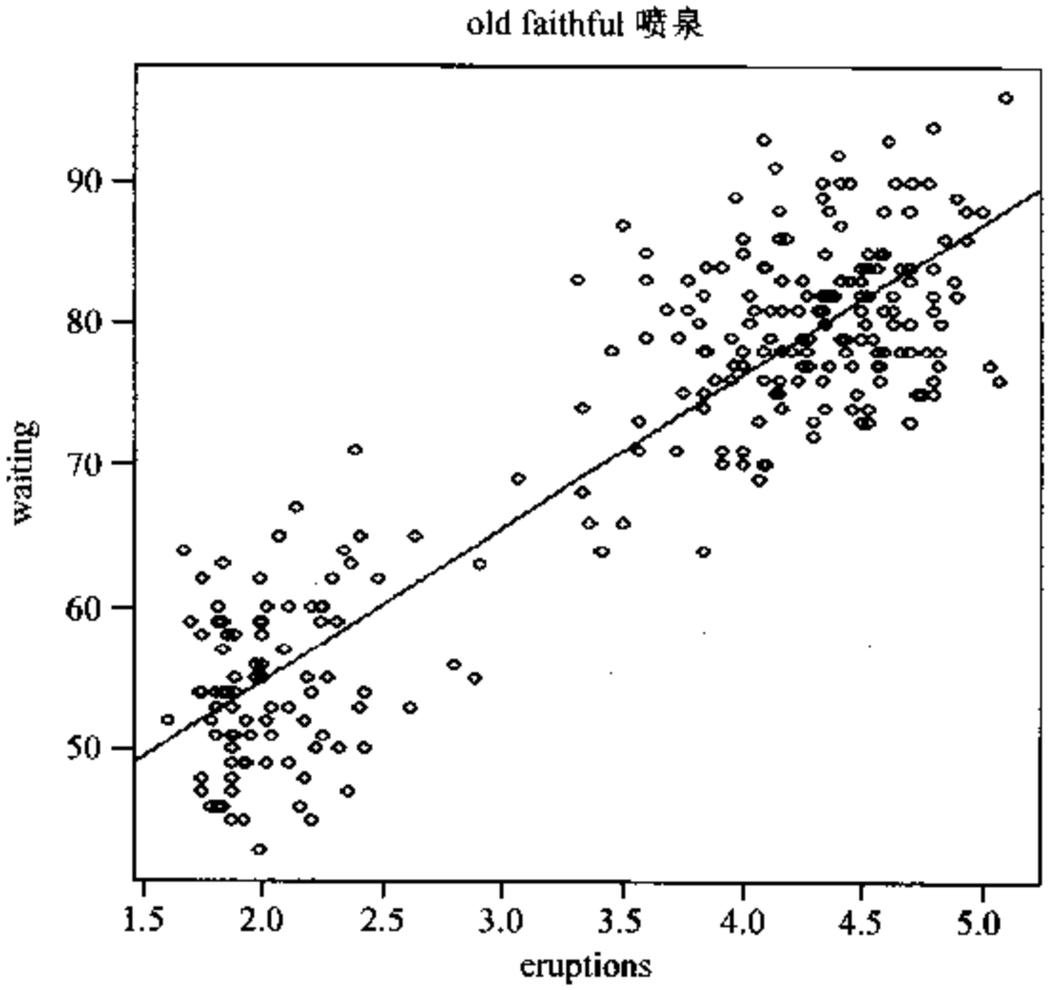
我们再以第六章介绍过的 Old Faithful 喷泉数据为例，进一步说明 R 的制图功能。首先我读取 Old Faithful 数据。

```
> data(faithful)
> attach(faithful)
> names(faithful)
[1] "eruptions" "waiting"
```

Old Faithful 喷泉有两列数据：eruptions 是每次喷水的时间而 waiting 是每次喷水相隔的时间。（数据是以每分钟计）。现在以下面指令绘制 waiting 与 eruptions 的散点图（scatter plot），加上标题及最小平方线（least square line）。

```
> plot(waiting ~ eruptions, main = "Old faithful 喷泉")
> abline(lsfit(eruptions, waiting))
```

以上 lsfit( ) 函是回传 waiting ~ eruptions 最小平方线之截距及斜率。再配合 abline( ) 函数在散点图内绘制最小平方线。



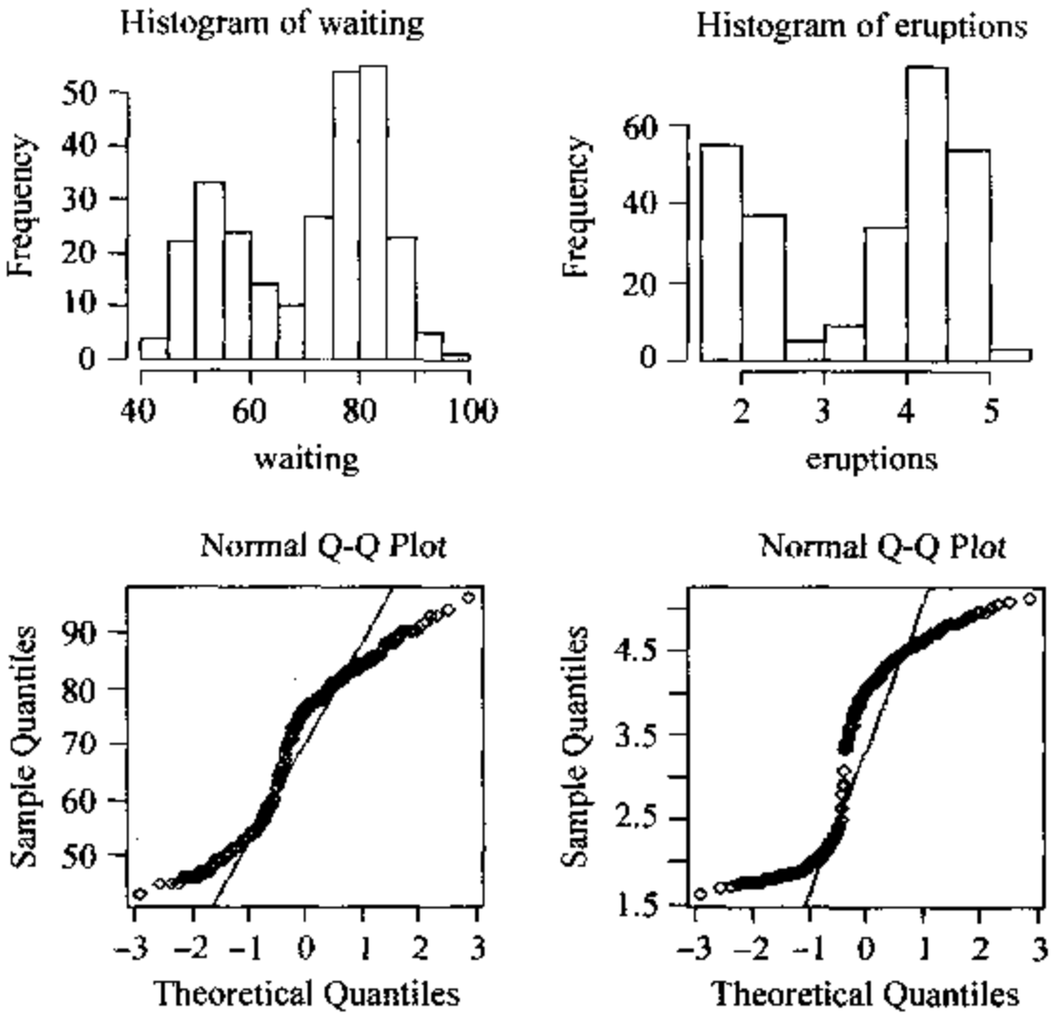
显然易见,图中的数据点分成两组:eruptions < 3.2 及 eruptions > =3.2。(或者可以 waiting < 70 及 waiting > =70 去划分)。

8.3 多重图框(Multiframe Graphic)

我们用直方图亦可以看出这两组数据点。不过先用 par(mfrow =c(2,2)) 去预设一个 2×2 的多重图框。

```
> par( mfrow = c(2,2))
> hist( waiting)
> hist( eruptions)
> qqnorm( waiting)
> qqline( waiting)
```

```
> qqnorm(eruptions)
> qqline(eruptions)
```



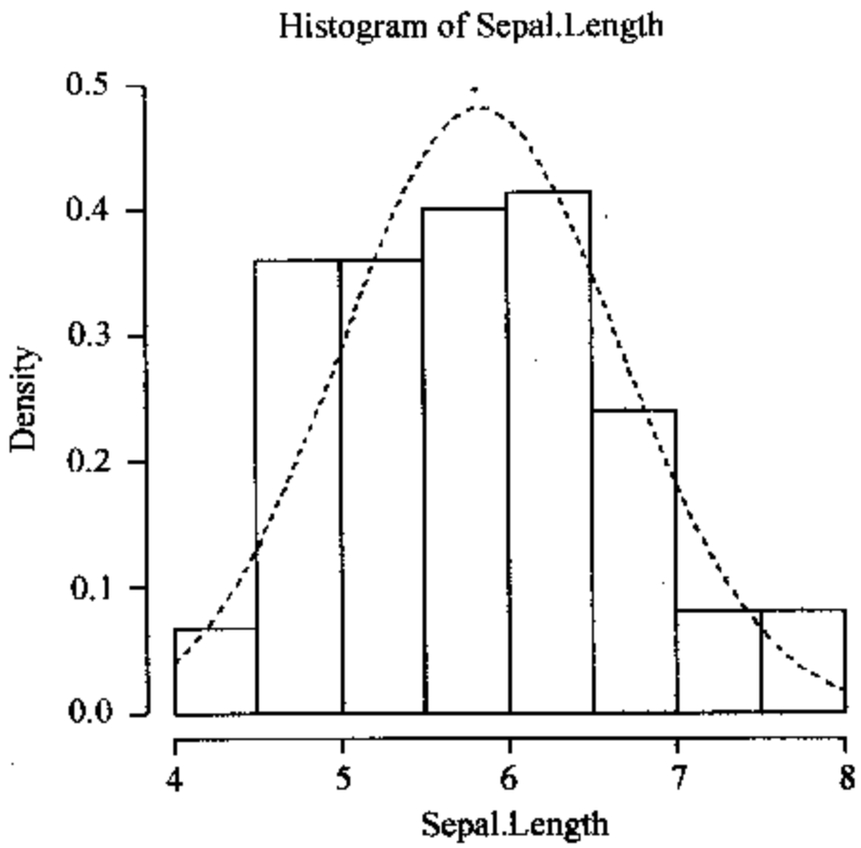
在 par( )内，mfrow = c(2,2)是设定一个 2 × 2 多重图框。mfrow 的意思是图像是逐行输入。视乎使用者的喜好，亦可以 mfcol 逐列输入图像。第二行的正态分位图 qqnorm( ) 是用来诊断数据分布是否近似正态分布。(这是一个常用的诊断方法，在第 6.3 节回归模型中已出现过)。qqline( )是在正态分位图中加上参考线，图中的点愈接近这条直线就表示数据愈接近正态分布。很明显，这些数据并非近似一个单一的正态分布，而是似由两个正态分布组成的混合分布(mixture distribution)。

问题 1：上图中的标题是自动加入，如何修改及加入自定的标题？

问题 2：紧接以上的例子，如果再有绘图指令，图框仍然是 2 × 2 多重图框。如何可以取消 2 × 2 多重图框，变回普通的图框？

8.4 修饰图像

R 提供了很多修饰图像的方法。我们就用在第七章出现过的iris鸢尾花数据为例。下面是 Sepal. Length 的直方图，在图中加上了正态密度的虚线。



首先我们用以下指令绘制 Sepal. Length 的直方图。

```
> data(iris)
> attach(iris)
> hist(Sepal. Length,freq = F,ylim = c(0,0. 5))
```

这里要说明一下，一般的直方图的 y - 轴的尺度是频率而非密度。但我们要加上正态密度线，尺度是非要密度不可。hist( ) 内的自变量 freq = F 就是指示 hist( ) 的 y - 轴不用频率而用密度。还有，ylim = c(0,0. 5) 是要设定 y - 轴的上下限为 0 及 0. 5。否则，R 就会选择最合适的上下限：0 至 0. 4。这样的话，我们就不能完全地看到整条正态密度曲线。接着就是用以下指令绘制正态密度曲线：

```
> m = mean(Sepal. Length)
> s = sd(Sepal. Length)
> x = seq(4,8,0. 1)
> lines(x,dnorm(x,m,s),lty = 2)
```

在绘制正态密度曲线之前,我们要想想究竟分布的平均及标准差是什么。最简单而合理的是用样本的平均及标准差。m 及 s 就储存了样本平均及标准差。数值向量 x 是 x - 轴的坐标,dnorm(x,m,s) 是相应的密度。最后我们以 lines() 绘制曲线,自变量 lty = 2 代表以虚线绘制。读者如果想知更多关于绘制的自变量,可用 help(par) 查看。

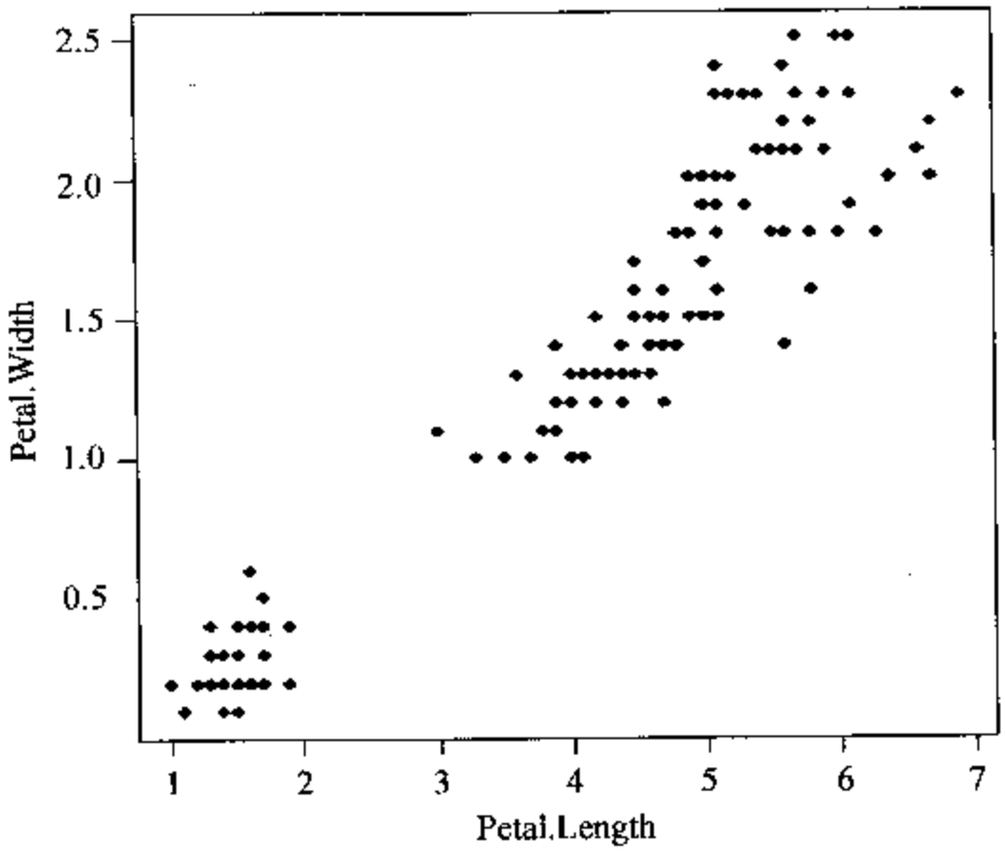
问题:请产生 1000 点自由度为 5 卡方分布(Chi - square distribution)的随机随机数。然后绘制直方图并加上自由度为 5 卡方分布的密度曲线。

8.4.1 应用颜色及字符

在 R 的绘图可用不同颜色,例如 hist(x,col = "red") 就有红色的直方图。除了这些简单的应用之外,我们也有一些更好的及较有意义的应用。在 7.4 节的分类树中,我们知道只要用 Sepal. Length 及 Sepal. Width 就可大概分辨三种鸢尾花。我们先用以下指令绘制 Sepal. Length 与 Sepal. Width 的散点图,而在散点图中,以不同的颜色代表不同品种。

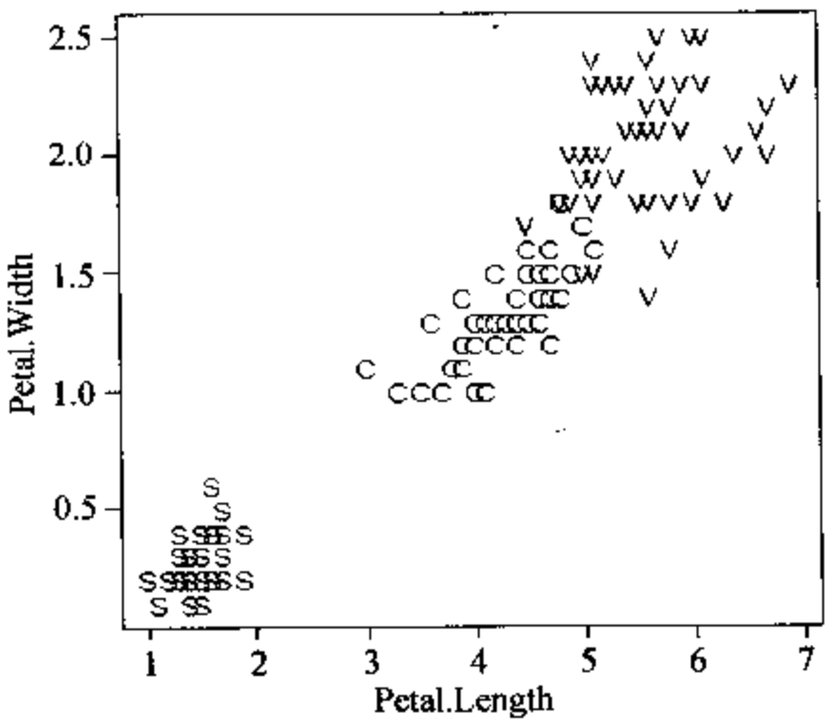
```
> plot(Petal. Length, Petal. Width, bg = c("red","green","blue")
[Species],pch = 21)
```

在 iris 中, Species 是储存鸢尾花品种的数据:分别是 setosa、versicolor 和 virginica。在以上 plot() 之中的自变量:bg = c("red","green","blue")[Species],就是分别以红色、绿色和蓝色绘制 setosa、versicolor 和 virginica。最后,pch = 21 是指用圆形绘制数据点。R 有很多其它形状可供选择,详情可用 help(points) 查看。



有时在黑白印刷的书籍里,彩色是派不上用场。我们也可以用不同的字符代表不同的品种（S,C,V 分别代表 setosa、versicolor 和 virginica）:

```
> plot(Petal. Length,Petal. Width,pch = c("S","C","V")[Species])
```



8.4.2 增加直线

在 7.4 节分类树中,我们找到简单的法则帮助分类:

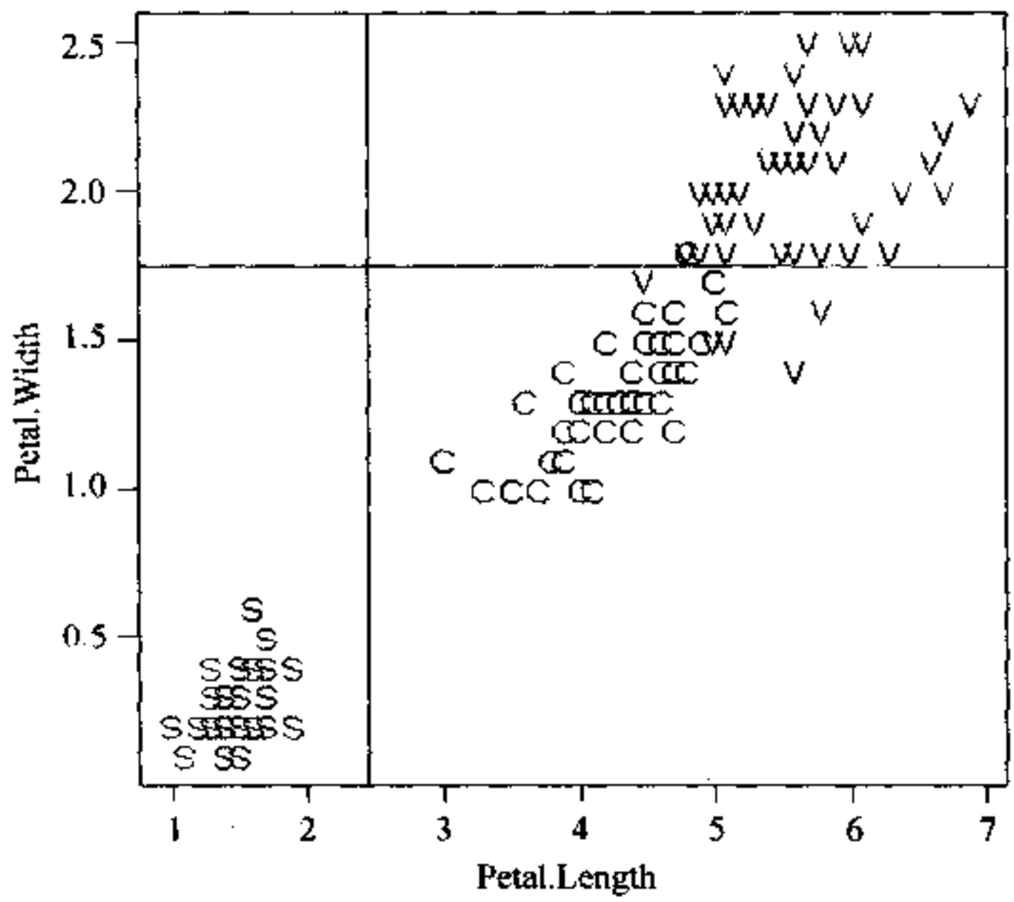
如果 Petal. Length < 2.45,Species = setosa (50/0/0);

否则,如果 Petal. Width < 1.75,Species = versicolor (0/49/5);

否则 Species = virginica (0/1/45)。

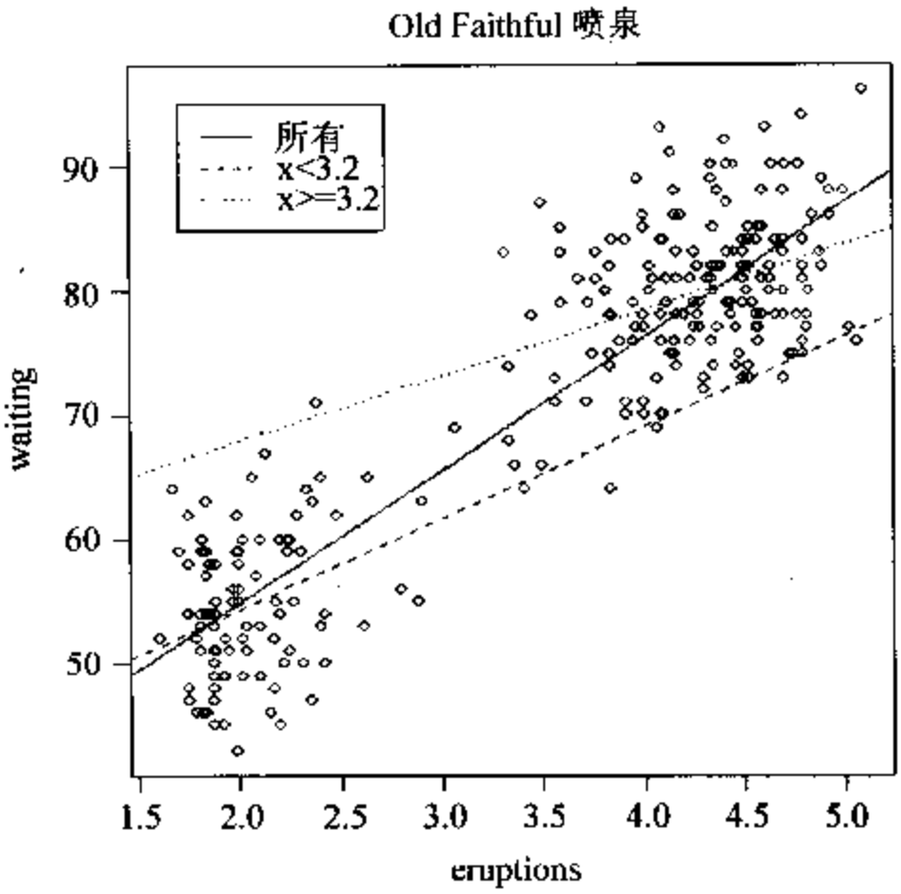
现在我们就用以上的法则,在散点图内加上直线。

```
> abline(v = 2.45)
> abline(h = 1.75)
```



在 abline 的自变量,h 代表水平线而 v 代表垂直线。当然,abline 还可在图像上增加直线 y = a + b * x。我们再以 8.2 节的 Old Faithful 喷泉为例。我们知道数据点自然分成两组:eruptions < 3.2 及 eruptions > =3.2。现在尝试绘制三条最小平方线:一条是包括所有数据点,另一条只包含 eruptions < 3.2,最后一条是只包含 eruptions > = 3.2。

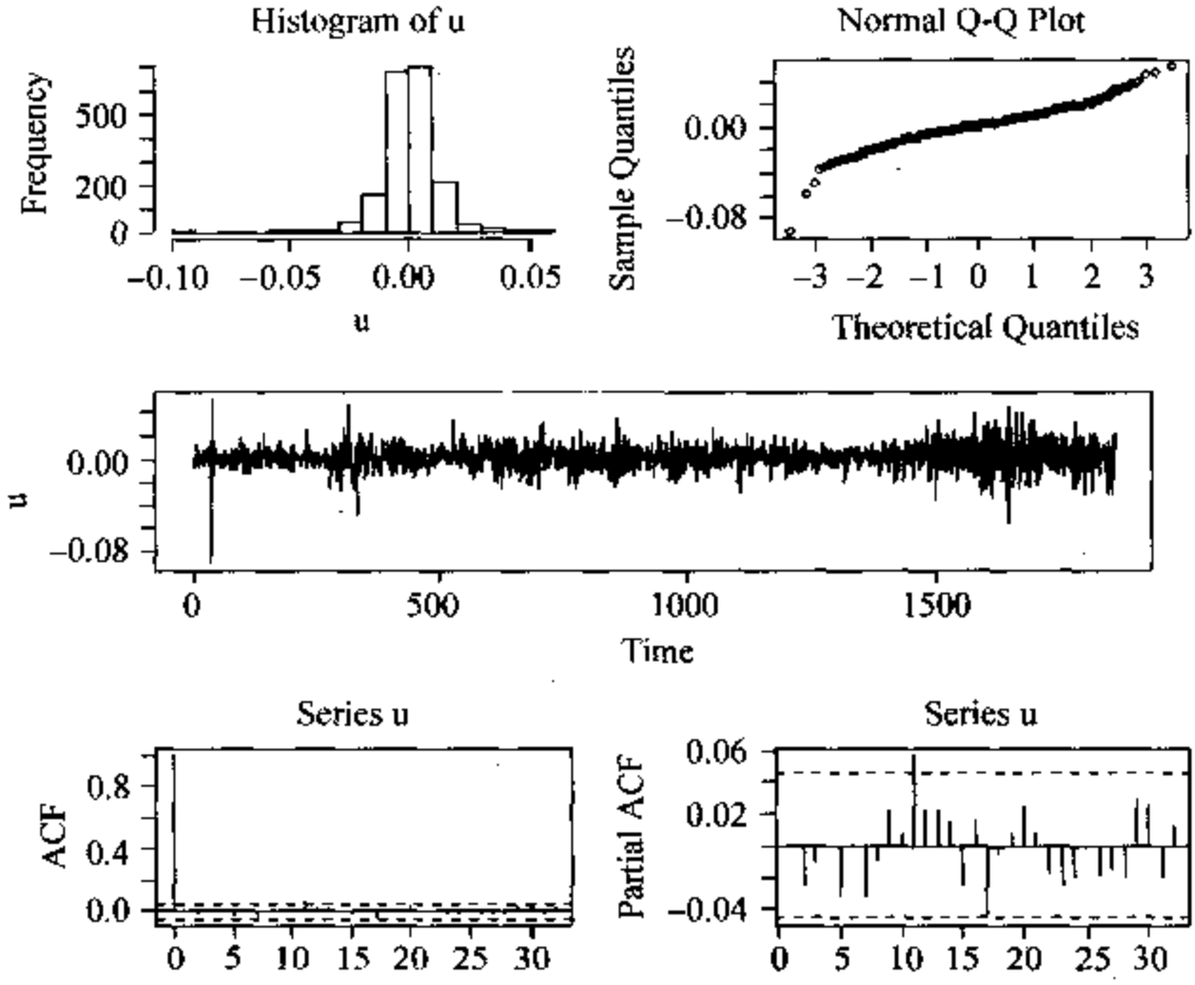
```
> plot(eruptions,waiting,main = "Old Faithful 喷泉")
> abline(lsfit(eruptions,waiting))
> d1 = faithful[eruptions < 3.2,]
> d2 = faithful[eruptions > = 3.2,]
> abline(lsfit(d1 $ eruptions,d1 $ waiting),lty = 2)
> abline(lsfit(d2 $ eruptions,d2 $ waiting),lty = 3)
> legend(1.7,95,c("所有","x < 3.2","x > = 3.2"),lty = c(1:
3))
```



首先绘制 waiting 对 eruptions 的散点图及第一条最小平方线。然后我们选取了 eruptions < 3.2 的数据点并储存在 d1。最后将 eruptions > = 3.2 的数据点储存在 d2。在 8.2 节已经提及过,lsfit( ) 回传最小平方线的截距 a 及斜率 b。lty = 2 是表示用虚线,lty = 3 是用点线。(可用 help(par) 查看更多选项)。当然,我们可以直接输入 a,b 值去画直线:y = ax + b。(例如第一条线是 abline(33.47, 10.73))。

8.5 多重图格（Multiframe Grid）

在8.3节中，我们可以用 par(mfrow =) 或 par(mfcol =) 去定义多重图框。通常在图框中，每一行都有相同数目的图像。但有时我们可以制造一些特别的多重图框。用 mfg 配合 mfrow，我们可以更灵活地绘制多重图框。再以7.7节的 EuStockMarkets 数据的 DAX 指数为例，我们要绘制以下的图像。注意：第二行只有一幅时间序列的图像。



我们先选取 EuStockMarkets 数据，定义 u 及设定一个 3 × 2 的多重图框。在第一行绘制 u 的直方图及正态分位图。

```
> data(EuStockMarkets)
> d = data.frame(EuStockMarkets)
> t = as.ts(d $ DAX)
```

```
> u = (lag(t) - t)/t
> par(mfrow = c(3,2))
> hist(u)
> qqnorm(u)
```

由于第二行只有一幅图像，我们用 par(mfrow = c(3,1)) 重新设定多重图框为 3x1。然后再用指令 par(mfg = c(2,1,3,1)) 去设定要绘制图像的位置。注意：在 c(2,1,3,1) 的前两个数字是代表想要绘制图像的位置，最后的两个数字是指目前多重图框的定义。然后绘制 u 的时间序列。

```
> par(mfrow = c(3,1))
> par(mfg = c(2,1,3,1))
> plot(u)
```

完成之后，我们要再次设定多重图框为 3 ×2，再用 par(mfg = c(3,1,3,2)) 去设定要绘制图像的位置。最后绘制 u 的自相关函数（autocorrelation function）及偏自相关函数（partial autocorrelation function）。

```
> par(mfrow = c(3,2))
> par(mfg = c(3,1,3,2))
> acf(u)
> pacf(u)
```

问题：试用 iris 数据内的 Sepal. Length，绘制一个多重图框。左边是三种花的盒形图，右边是三个由上至下的直方图，分别是三种花的 Sepal. Length。

第九章

最优化方法

9.1 引言

R 除了有很多内置统计函数外，亦有不少关于最优化方法的函数。在统计学上，最优化方法也应用很多，例如：最大似然估计、非线性最小平方及稳健回归（Robust regression）等等。我们在这章中介绍一般的最优化方法。

9.2 非线性方程求解

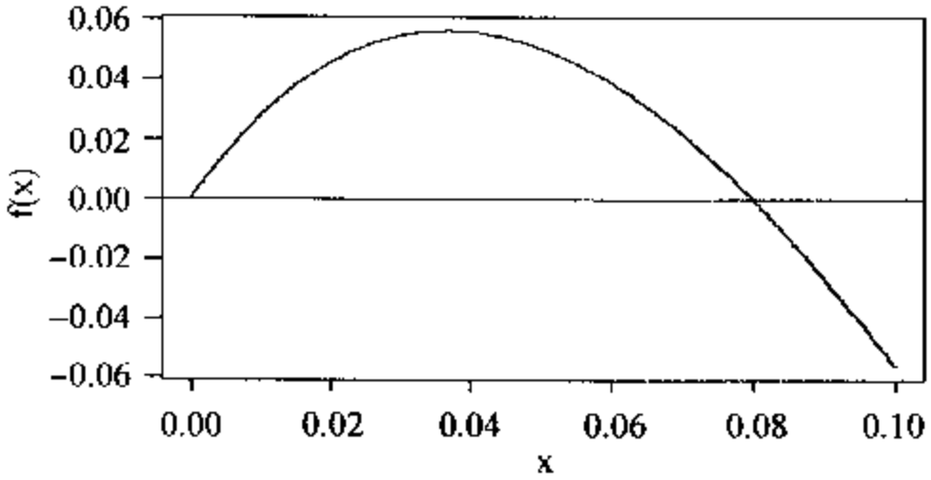
在工程、金融或其它学科中，我们有时要求非线性方程的根。R 提供了 uniroot( ) 函数去寻找一维非线性方程的根。以下是一个计算利率的简单例子，求方程式：$1-(1+x)^{-10}=6.71x$ 的根。

```
> f = function(x) { 1 - (1 + x)^( -10)  -6.71 * x }
> x = uniroot(f,c(0.0001,1))
> x $ root
[1] 0.08000033
```

首先定义函数 $f(x)=1-(1+x)^{-10}-6.71x$，然后用 uniroot( ) 去寻找方程式 $f(x)=0$ 的根。答案是 $x=0.08000033$。注意：c(0.0001,1) 是设定寻找区域 $[a,b]$ 的上下限，而 $f(a)$ 及 $f(b)$ 需要有不同正负号，亦即是 $f(a)*f(b)<0$。由于 $f(x)$ 是一元函数，我们可以绘制

$f(x)$从而更清楚知道$f(x)$的根。

```
> plot(f,xlim = c(0,0.1))
> abline(h = 0)
```

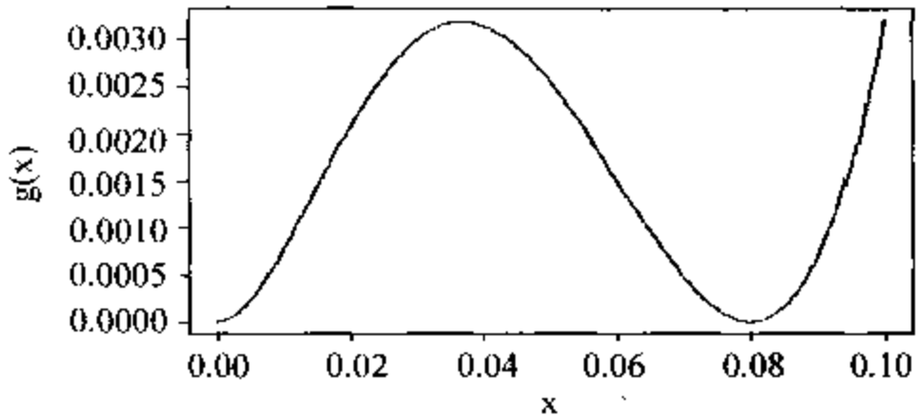


9.3 一元函数最优化方法

R 内置函数 optimize( ) 是对一元函数最优化。对方程式$f(x) = 0$求解是相当于对$g(x) = f(x) * f(x)$极小化。我们就利用 9.2 节$f(x)$的例子，定义$g(x) = f(x) * f(x)$，然后对$g(x)$极小化：

```
> g = function(x) { f(x)^2 }
> optimize(g,c(0.0001,1))
$ minimum
[1] 0.0800112
$ objective
[1] 0
```

答案$x = 0.0800112$与 9.2 节的答案大致相同。（用 optimize( ) 的好处是区域的上下限不一定要有不同的正负号。）我们亦可以用 plot (g,xlim = c(0,0.1)) 绘制 g(x) 函数：



再以一个与 L1 拟合（一种稳健统计方法）的例子来说明。我们极小化以下函数：

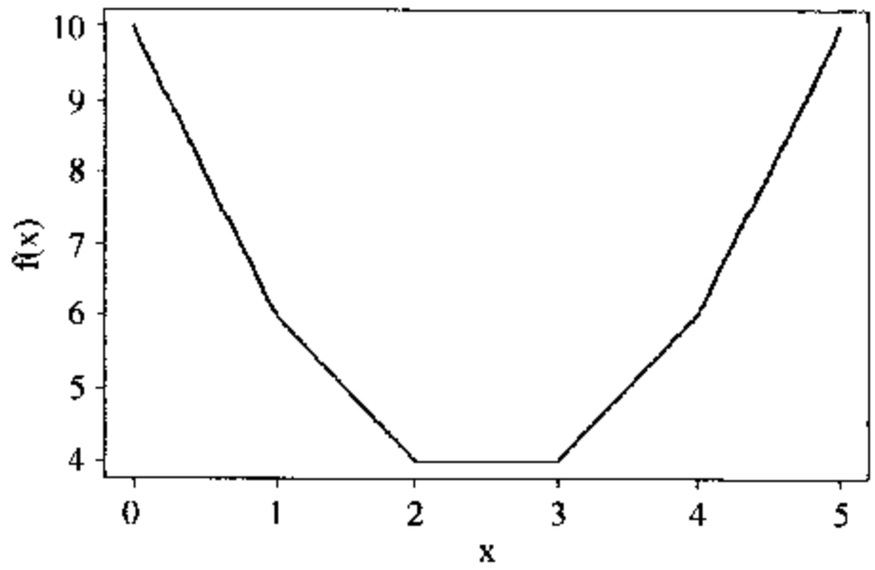
$$f(x) = |x-1| + |x-2| + |x-3| + |x-4|$$

```
> f = function(x) { abs(x - 1) + abs(x - 2) + abs(x - 3) + abs(x - 4) }
> optimize(f,c(0,5))
$ minimum
[1] 2.55042
$ objective
[1] 4
```

答案 x = 2.55042 而函数极小值为 4。如果我们再试

```
> optimize(f,c(0,10))
$ minimum
[1] 2.32339
$ objective
[1] 4
```

答案是 $x = 2.32339$ 而函数极小值仍然是 4。我们可以用 plot(f, xlim = c(0,5)) 去绘制 $f(x)$，就更清楚了解此函数的极小值。



从上图知道,所有由 2 至 3 之间的实数都是答案。optimize( ) 只提供其中一个答案。optimize( ) 预设了对函数极小化;如果想对函数极大化,可以加入自变量 maximum = T。

9.4 多元函数最优化方法

多元函数极优化问题远比一元函数的复杂和困难。我们通过以下的例子去说明多元函数极优化的困难。假设我们寻求以下函数的最大值:

$$z = f(x,y) = \frac{\sin\sqrt{(x-5)^2 + (y-5)^2}}{\sqrt{(x-5)^2 + (y-5)^2}}$$

R 有内置函数 optim( ) 去找寻多元函数的极值,但我们先绘制这函数来看看。由于这是二维函数,我们还可勉强用三维透视图 (3D Perspective Plot) 去绘制这函数,亦正好补充在上一章没有提及的三维透视绘图。首先定义函数 fxy:

```
> fxy = function(x,y) {
  p = sqrt((x-5)^2 + (y-5)^2)
  sin(p)/p }
```

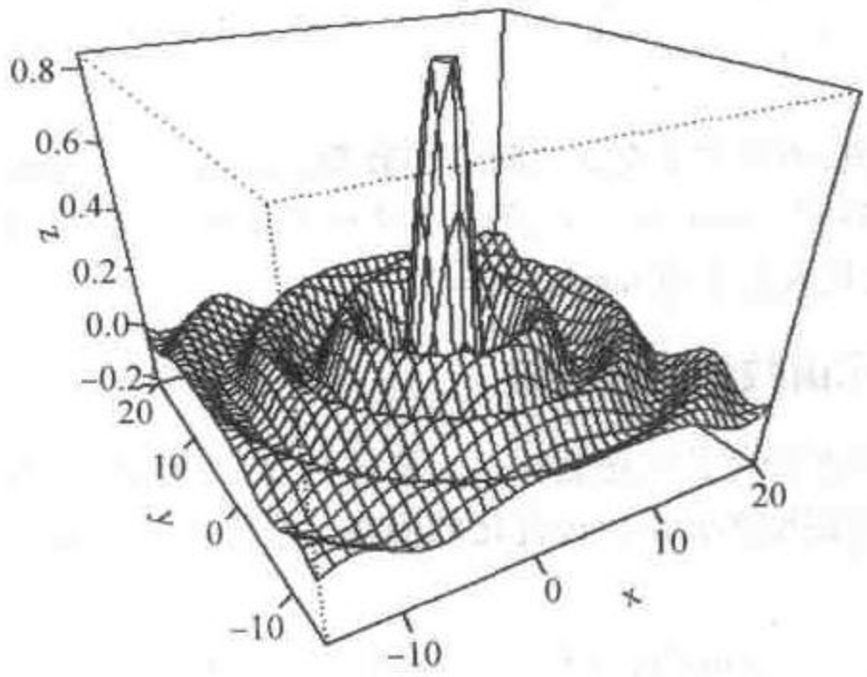
然后用以下指令制造格点 (x,y,z):

```
> x = seq(-15,20,step = 0.5)
```

```
> y = x
> z = outer(x,y,fxy)
```

x 和 y 都是长度为 36 的数值向量，outer(x,y,fxy) 是回传一个 36x36 的数值矩阵。假设在向量 x 中第 i 个元素是 xi，向量 y 中第 j 个元素是 yj，则 z 内的第 (i,j) 个元素，也就是 fxy(xi,yj) 的数值。我们有了 (x,y,z) 之后，就可用以下指令绘制：

```
> persp(x,y,z,theta = -30,phi = 30,ticktype = "detailed")
```



theta = -30 和 phi = 30 是设定视点角度，ticktype = "detailed" 是显示 x,y,z - 轴的刻度。由上图来看，可以知道当 x = 5 及 y = 5 时，fxy 的值是最大。(最大值是 1，但不能直接计算，需要用洛必达法则 L'Hospital's rule 去计算)。这函数亦有很多局部极大及极小值。所以一般的最优化方法都未必能找到总体最优值 (Global optimum)。

现在就以 optim() 函数找出 fxy 的极大值。但我们先要重新定义函数 fxy，原因是 optim() 要求输入函数的自变量是向量而非标量。以下重新定义 fxy 为 fx：

```
> fx = function(x) {
    p = sqrt((x[1] -5)^2 + (x[2] -5)^2)
    sin(p)/p }
```

```
> optim(c(0,0),fx,control = list(fnscale = -1))
$ par
[1] -1.136790  0.307462
$ value
[1] 0.1283746
```

第一个自变量 c(0,0)是使用者输入的原始点。optim( ) 函数的预设是极小化,control = list(fnscale = -1) 是要进行极大化。从以上来看,optim( ) 的搜索被困在局部极大点 (local maximum),并非找出真正的答案。我们尝试改变原始点为 c(2,2):

```
> optim(c(2,2),fx,control = list(fnscale = -1))
$ par
[1] 4.999951 4.999967
$ value
[1] 1
$ counts
```

这次,optim( ) 可以找到真正的极大点。由此可见,对最优化方法而言,好的原始点是很重要。optim( ) 还可以选择以不同的最优化方法,详情可参看 help(optim)。

R 还有些最优化的函数如线性限制最优化(Linear constraint optimization) 的 constrOptim( ) 及线性规划 (Linear Programming) 的 simplex( )函数(内置在 boot 程序馆)。有兴趣的读者可以用 help( ) 查看。

问题:试找出 $f(x,y) = 100(y - x^2)^2 + (1 - x)^2$ 之极小值。(可用 (x,y) = (-1.2,1)作为原始点。)

9.5 应用

在第 6.4 节中的 stackloss 数据有异常值,我们就以此为例,用 optim( ) 来执行 L1 拟合的稳健回归。首先我们读取 stackloss 数据及定义 L1 损失函数:

$$f(x) = \sum_{i=1}^{n} |y_i - \alpha - \beta' x_i|$$

```
> data(stackloss)
> attach(stackloss)
> fx = function(x) {
data(stackloss)
attach(stackloss)
sum(abs(stack.loss - x[1] - x[2] * Air.Flow - x[3] * Water.Temp - x
[4] * Acid.Conc.))}
```

我们先执行 lm()并以最小平方估计 ls $ coef 作为自变量的原始值,然后用 optim() 去极小化 f(x)。

```
> ls = lm(stack.loss ~ Air.Flow + Water.Temp + Acid.Conc.)
> optim(ls $ coef,fx)
> optim(ls $ coef,fx)
$ par
(Intercept)      Air.Flow     Water.Temp     Acid.Conc.
-39.05068480   0.82454258   0.60240870   -0.07020176
$ value
[1] 42.21094
```

L1 估计值的损失为 42.21094。用以下指令显示及比较最小平方估计及损失值(=49.69902)。很明显,L1 估计的损失值较小。

```
> ls $ coef
(Intercept)      Air.Flow     Water.Temp     Acid.Conc.
-39.9196744   0.7156402   1.2952861   -0.1521225
> fx(ls $ coef)
[1] 49.69902
```

主要参考书目：

1. W. N. Venables, B. D. Ripley. , Modern Applied Statistics with S, 4th ed. , Springer.
2. Phil. Spector, An introduction to S and S－plus, Duxbury Press.
3. Peter Dalgaard, Introductory Statistics with R, Springer.
4. 陈毅恒著，黄长全译，时间序列与金融数据分析，中国统计出版社.

英汉词汇对照及索引