HOMEWORK OF NONPARAMETRIC STATISTICS

HOMEWORK Ⅰ

周杰
116071910053

Shanghai Jiao Tong University

1

学习两种排序算法，在R中实现这些算法，比较新构造的排序算法和R中自带排序算法的性能差异并思考原因。

解答：选择冒泡排序(bubble sort)和插入排序(insertion sort)。

1.1 实现排序算法

以下简述两种排序算法的具体思路并给出具体实现。

1.1.1 冒泡排序(bubble sort)

使用一个二元框在数组上滑动，右值小于左值时进行交换，则第k次循环后数组的后k个数为该数组最大的k个数。程序实现：

```
bubble.sort <- function(array){
   size <- length(array)
      for(i in size:2){
         for(j in 1:(i-1)){
            if(array[j] > array[j + 1]){
               temp <- array[j]
               array[j] <- array[j + 1]
               array[j + 1] <- temp
         }
      }
   }
   return(array)
}
```

测试结果为：

```
> bubble.sort(rnorm(10))
[1] -1.1544865 -0.7236358 -0.7178156 -0.6530491 -0.5129474 -0.3193575
    0.4291747 0.6020488 0.6921175
[10]  1.4432690
```

1.1.2 插入排序(insertion sort)

第k次迭代时保证前k个元素是已经排好序的，此时将第k+1个元素插入到前k个排好序的子数组中形成k+1个排好序的子数组。程序实现：

```
insertion.sort <- function(array){
   size <- length(array)
   for(i in 2:size){
      for(j in (i-1):1){
         if(array[j] > array[j + 1]){
            temp <- array[j]
            array[j] <- array[j + 1]
            array[j + 1] <- temp
         }
      }
   }
   return(array)
}
```

测试结果为：

```
> insertion.sort(rnorm(10))
[1] -1.9191690 -1.0425396 -1.0261981 -0.3403006 0.5403804 0.7376935
    0.8118606 0.8192611 0.8920983
[10] 1.8469651
```

1.2 比较算法性能

显然算法的性能依赖于两个因素：算法的设计(复杂度)以及实现该算法的语言。以下分别就这两个方面进行讨论

1.2.1 算法复杂度的影响

冒泡排序和插入排序的时间复杂度都是$\Theta(n)$其中n为待排序数组的元素数量，因此两者的速度应该是一个数量级的，为此，编写如下程序来记录运行时间。

```
# record run time
RecordRunTime <- function(sort.algorithm, run.times) {
```

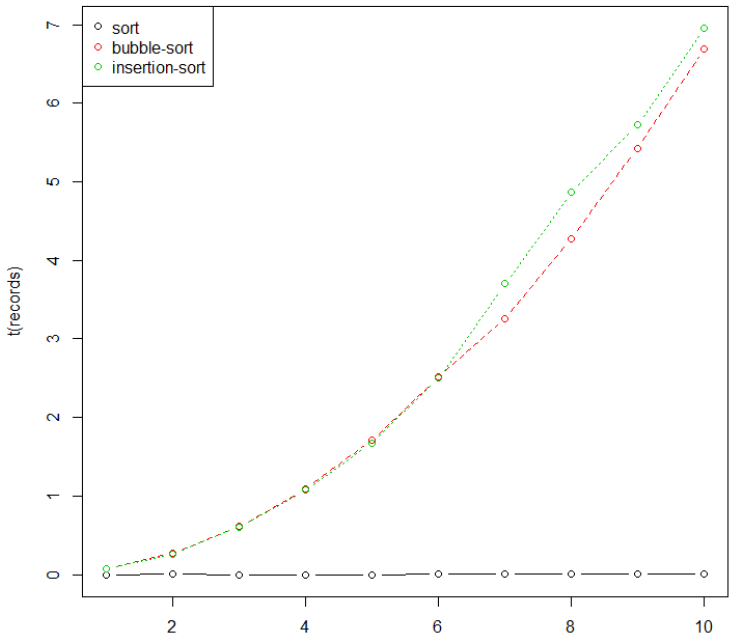
```
  length.run.times <- length(run.times)
  record <- vector(length = length.run.times)
  for(i in 1:length.run.times) {
    start.time <- Sys.time()
    sorted <- sort.algorithm(rnorm(run.times[i]))
    end.time <- Sys.time()
    duration <- as.vector(end.time - start.time)[1]
    record[i] <- duration
  }
  return(record)
}

run.times.lenth <- 10
run.times <- seq(1, run.times.lenth) * 1000
sort.algorithms <- c(sort, bubble.sort, insertion.sort)
num.sort.algorthms <- length(sort.algorithms)
records <- matrix(nrow = num.sort.algorthms, ncol = run.times.lenth)

for(i in 1:length(sort.algorithms)) {
  sort.algorithm <- sort.algorithms[[i]]
  records[i,] <- RecordRunTime(sort.algorithm, run.times)
}

matplot(t(records), type = c("b"),pch=1,col = 1:3)
legend("topleft", legend = c("sort", "bubble-sort",
    "insertion-sort"), col=1:3, pch=1)
```

绘出的图像如下图所示：



从上可以看出，R实现的插入排序和冒泡排序的复杂度一样因此对于同样量级的数据使用的排序时间也是几乎一样的，而内置的sort函数的所用时间几乎都是0。

1.2.2 编程语言的影响

为了比较语言给同一个算法带来的影响，在原来用R语言实现的快速排序的基础上，使用C++语言再次实现该算法，并通过Rcpp封装给R语言调用，代码如下：

```
#include <Rcpp.h>
#include <iterator>
using namespace Rcpp;
// [[Rcpp::export]]
void insertion_sort(NumericVector &x) {
   NumericVector::iterator iter1 = x.begin(), iter3 = x.begin(),
       iter2, iter4;
   ++iter1;
```

```
    --iter3;
       for(;iter1!=x.end();++iter1) {
       iter2 = iter1;
       --iter2;
          for(;iter2!=iter3;--iter2) {
             iter4 = iter2;
             ++iter4;
             if(*iter2 > *iter4) {
                std::swap(*iter2, *iter4);
          }
       }
    }
}

/*** R
array <- rnorm(10000)
start.time <- Sys.time()
insertion_sort(array)
end.time <- Sys.time()
print(end.time - start.time)
*/
```

输出结果为：

```
> array <- rnorm(10000)

> start.time <- Sys.time()

> insertion_sort(array)

> end.time <- Sys.time()

> print(end.time - start.time)
Time difference of 0.07200003 secs
```

对10000个来自于正态分布的样本的排序使用了0.072秒。再来测试R编写的插入排序和内置的sort函数所使用的时间：

```
> array <- rnorm(10000)
```

```
> start.time <- Sys.time()
> sorted <- insertion.sort(array)
> end.time <- Sys.time()
> print(end.time - start.time)
Time difference of 6.831 secs
>
> start.time <- Sys.time()
> sorted <- sort(array)
> end.time <- Sys.time()
> print(end.time - start.time)
Time difference of 0.002000093 secs
```

可以看出R编写的排序一万个数使用了接近7秒，而Rcpp编写的使用了0.07秒，内置版本的sort函数甚至只用了0.002秒！可以看出不同的语言的实现对算法运行速度的影响是有多么的大!

2

对于某个总体分布，写出其次序统计量中心项的渐近分布和极值统计量的极限分布，并在R中模拟渐近拟合情况。

解答：选择标准正态分布，记概率密度函数为$\phi(x)$，分布函数为$\Phi(x)$。

2.1 中心项的渐近分布

样本p分位数的定义为

$$\epsilon_{np} = X_{([np])} + (n-1)(p - \frac{[np]}{n+1})(X_{([np]+1)} - X_{([np])})$$

其近似分布为

$$\sqrt{n}(\epsilon_{np} - \Phi^{-1}(p)) \sim N(0, p(1-p))$$

因此中位数的定义为

$$\epsilon_{n\frac{1}{2}} = \begin{cases} X_{(\frac{n+1}{2})} & x \quad is \quad old \\ \frac{1}{2}(X_{(\frac{n}{2})} + X_{(\frac{n+1}{2})}) & otherwise \end{cases}$$

其渐近分布为

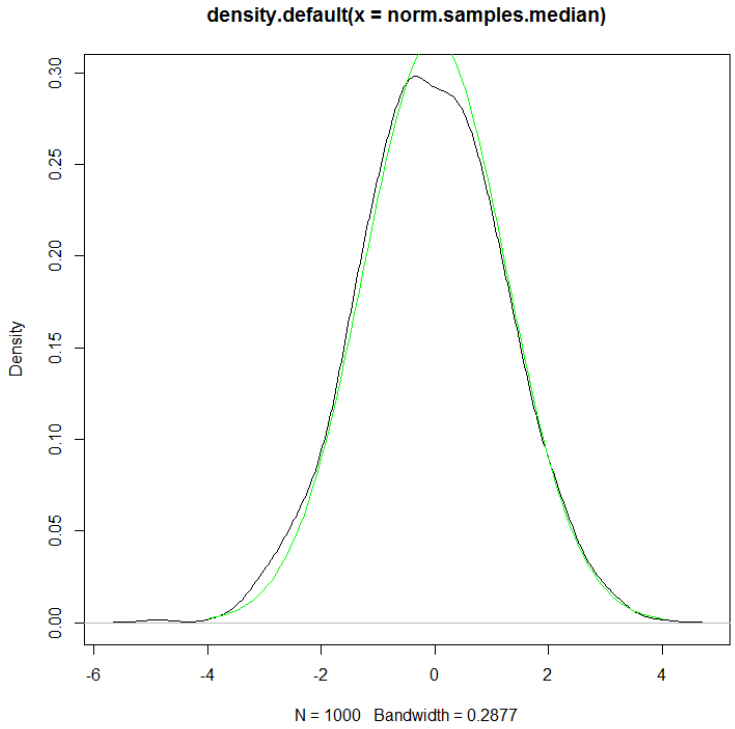
$$\sqrt{n}(\epsilon_{n\frac{1}{2}} - \Phi^{-1}(\frac{1}{2})) \sim N(0, \frac{1}{4\phi^2(\Phi^{-1}(\frac{1}{2}))})$$

下面展示R中进行模拟的结果: n取为10000，模拟1000次，每次取10000个样本中，计算其中位数，组成一个1000的样本，并用**核密度估计(kernel density esitimation)**做出概率密度函数的估计，并与对应的正态分布的概率密度函数进行比较

```
require(graphics)
n = 10000
run.times = 1000
total.times <- n * run.times
norm.samples.vector <- rnorm(total.times)
norm.samples.matrix <- matrix(norm.samples.vector, nrow = n)
norm.samples.median <- apply(norm.samples.matrix, 2, median)


norm.samples.median <- sqrt(n) * (norm.samples.median - qnorm(1/2))
plot(density(norm.samples.median))
sd <- 1 / (2 * dnorm(qnorm(1/2)))
x.vector <- seq(from = -4, to = 4, length.out = 100)
y.vector = dnorm(x.vector, sd=sd)
lines(x.vector, y.vector, col="green")
```

绘出的图像如下图所示：



从上图可以看出，核密度估计出的中位数的渐近分布与均值为零方差为$\frac{1}{4f^2(F^{-1}(\frac{1}{2}))}$分布非常吻合。

2.2 极值统计量的极限分布

因为正态分布的最大次序统计量和最小次序统计量具有对称性，即$X_{(n)} = -X_{(1)}$，因此此处只讨论最大次序统计量，目标为寻找a_n和b_n使得

$$\Phi(\frac{1}{a_n}x + b_n)^n \to G(x)$$

利用渐进展开得到

$$\Phi(x) = 1 - \frac{e^{-x^2/2}}{\sqrt{2\pi}}\left(\frac{1}{x} + \dots\right)$$

第二项中$\frac{e^{-x^2/2}}{x\sqrt{2\pi}}$占收敛速度的主导地位，因此主要处理该项并丢弃后面的内容，令$a_n = b_n$，$b_n \to \infty$，则有

$$\frac{e^{-(\frac{1}{b_n}x+b_n)^2/2}}{(\frac{1}{b_n}x + b_n)\sqrt{2\pi}} = \frac{e^{-x^2/2b_n^2}e^{-x}}{x/b_n^2 + 1}\frac{e^{-b_n^2/2}}{b_n\sqrt{2\pi}}$$

关于右边第一项有

$$\lim_{n\to\infty}\frac{e^{-x^2/2b_n^2}e^{-x}}{x/b_n^2+1}=e^{-x}$$

关于右边第二项有

$$\frac{e^{-b_n^2/2}}{b_n\sqrt{2\pi}}\sim 1-\Phi[b_n]$$

为此选择$b_n=\Phi^{-1}(1-1/n)$，则有右边第二项可以变成

$$\frac{e^{-b_n^2/2}}{b_n\sqrt{2\pi}}\sim 1-\Phi[\Phi^{-1}(1-1/n)]=\frac{1}{n}$$

此时有

$$\frac{e^{-(\frac{1}{a_n}x+b_n)^2/2}}{(\frac{1}{a_n}x+b_n)\sqrt{2\pi}}\sim\frac{e^{-x}}{n}$$

$$\lim_{n\to\infty}\Phi(\frac{1}{a_n}x+b_n)^n=\lim_{n\to\infty}\left(1-\frac{e^{-x}}{n}\right)^n=e^{-e^{-x}}$$

综上所述，选择 $b_n=\Phi^{-1}(1-1/n), a_n=b_n$可以使得

$$\Phi(\frac{1}{a_n}x+b_n)^n\to G$$

其中G为$Gumbel$分布

下面展示R中进行模拟的结果: n取为10000，模拟1000次，每次取10000个样本中的最大值，组成一个1000的样本，并用**核密度估计(kernel density esitimation)**做出概率密度函数的估计，并与Gumbel分布的概率密度函数进行比较

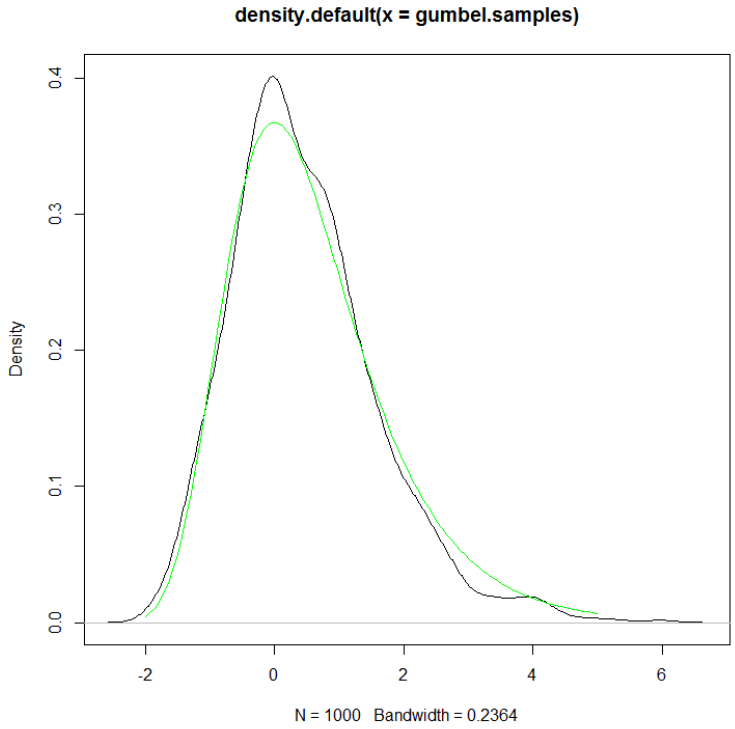
```
require(graphics)

n = 10000
run.times = 1000
total.times <- n * run.times
norm.samples.vector <- rnorm(total.times)
norm.samples.matrix <- matrix(norm.samples.vector, nrow = n)
norm.samples.max <- apply(norm.samples.matrix, 2, max)
b.n <- qnorm(1-1/n)
a.n <- b.n
gumbel.samples = a.n * (norm.samples.max - b.n)

plot(density(gumbel.samples))
```

```
CalculateGulbelDesity <- function(x, mu = 0, beta = 1) {
return(1/beta * exp(-(x + exp(-(x - mu)/beta))))
}
x.vector <- seq(from = -2, to = 5, length.out = 100)
y.vector = CalculateGulbelDesity(x.vector)
lines(x.vector, y.vector, col="green")
```

绘出的图像如下图所示：



从上图可以看出，核密度估计出的极大值的极限分布与Gumbel分布非常吻合。

3

利用本章节所学知识，构造一个统计应用案列。例如某个总体分布的参数估计、置信区间、假设检验等等。

解答：选择分布函数为

$$f(x)=\begin{cases}\exp(-x-\alpha) & x\geq\alpha\\ 0 & otherwise\end{cases}$$

为研究的分布，研究参数α置信区间，有$x_{(1)}$的是α的充分统计量，其概率分布函数为

$$F_{x_{(1)}}(x)=1-[1-F[(x)]]^n=1-\exp(-n(x-\alpha))$$

则$x_{(1)}-\alpha$的概率分度函数为

$$F_{x_{(1)}-\alpha}=1-\exp(-nx)$$

故$x_{(1)}-\alpha\sim Exp(n)$，由此得到$\alpha$置信度为95%的置信区间为

$$[x_{(1)}-Exp_{0.975}(n),x_{(1)}-Exp_{0.025}(n)]$$

其中$Exp_p(n)$表示参数为n的p分位数点。

4

找一个实际数据，利用本章节所学知识进行统计分析。阐述清楚感兴趣的问题、统计模型、所用方法原理及相关统计分析结果。

解答：本题将探讨**中值滤波法(median filter)**在**图像降噪**中的背景、理论、实践和与均值滤波法的比较以及数学解释。

4.1 背景

图像在采集以及传播的过程中会因为各种各样的原因(如采集图像传感器的老化，传播过程中的无意的污染)被添加上噪声，去除噪声还原图像是图像处理是非常重要的一步，而滤波是图像去噪一个非常重要的方法。

4.2 理论

4.2.1 图像表示

一张彩色图像实际上是$h\times w\times 3$的三维张量(记为C)，其中3代表图像的三原色红黄蓝对应的通道，一张黑白图像是一个$h\times w$的矩阵(记为B)。一

般的，表示图像的张量或者矩阵的每个元素的取值在$[0, 255]$之间，0代表纯黑色，255代表纯白色。为了表述简便以及不失一般性，仅仅讨论黑白图像的降噪，彩色图像的降噪原理类似。

4.2.2 图像噪声

$(B_{ij})_{mm}$是一个$m \times m$的矩阵，表示一张(方型)黑白图像，但是在采集或者传播过程中，因为种种原因某些像素被添加了噪声，变成了$(O_{ij})_{mm}$，有

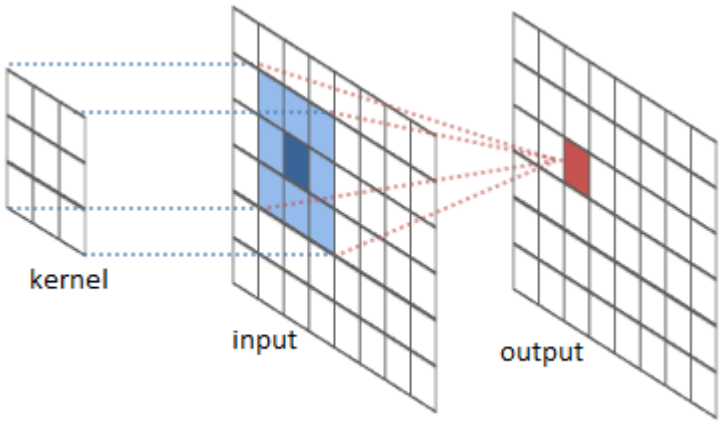
$$O_{ij} = B_{ij} + \epsilon_{ij}$$

其中$\epsilon_{ij} \sim F$，其中F为某个分布。图像降噪的任务是使得图像尽量恢复程原始的矩阵B。

根据F的不同，又有很多种不一样的噪声，其中一种很有名的(也是中值滤波法能很好处理的)噪声是盐椒噪声。其具体的表现形式为图像中随机点的位置出现一些纯白(盐)或纯黑(椒)的点，一张原始图像和添加了盐椒噪声的图像的示例见下图中(a)(b)两图

4.2.3 图像滤波

目前最常用的降噪方法为滤波，选取一个滤波核(一个矩阵)，对图像的每个位置进行滤波，具体过程见下图：



其中kernel矩阵与图中的input的对应大小的区域矩阵进行每个元素对应相乘最后相加的操作得到滤波后对应位置的元素。

根据滤波核F_{nn}的选取，又分为各种滤波类型，如：

均值滤波：

$$F_{mean} = \frac{1}{n^2}\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

中值滤波：

F_{median}没有具体的形式，其功能为选取对应图像的部分矩阵中的中位数作为滤波结果

4.3 应用

选取一张示例图片，并添加噪声，后再对添加噪声的图像分别进行中值滤波和均值滤波，得到的结果如下图所示：

(a) original image

(b) salt and peper added

(c) mean filtered

(d) median filtered

从图像中可以看出，均值滤波并没能有效地去除噪声并且还降低了图

像的清晰度，而中值滤波则在损失清晰度不多的条件上有效地去除了盐椒噪声。

4.4 解释

这里给出中值滤波能够有效去除盐辣噪声的原因:

盐椒噪声随机的出现在图像的某一些像素点当中，以255(盐，对应白色点)或者0(椒，对应黑色点)覆盖原像素点的像素值从而产生噪声。而原始图像的值在$[0, 255]$中，因此对某一给定的像素点进行滤波时，中值滤波选取了这一部分区域中的中位数作为滤波结果，而噪声是0或者255这样的极端值，因此因为中位数的稳健性，滤波得到的结果与原始像素值差不多。而反观均值滤波，因为均值受到极端值的影像非常大，在包含噪声区域进行滤波时得到的结果会非常不理想，这一点也直观的体现在了示例图片(c)上。

综上所述，使用中值滤波来处理盐椒噪声实际上是利用了中位数的稳健性，对于其他类型的噪声，中值滤波的效果并不一定比均值滤波好。