Programming in R

2007 R 統計軟體研習會

蔡政安

Associate Professor
Department of Public Health & Biostatistics Center
China Medical University

Control Structures – conditional executions

- Comparison operators: == (equal), != (not equal), >= (greater than or equal), etc. Logical operators: & (and), | (or) and ! (not).
- **If Statement** operates on length-one logical vectors

  Syntax

  if (cond1=true) { cmd1 } else { cmd2 }

  Example

  if (1==0) { print(1) } else { print(2) }
- Avoid using newlines between '} else'. To apply if statement on many values of vector or data frame, use 'for' or 'apply' loops (see below)

Control Structures – conditional executions

- **Ifelse Statement**: operates on vectors

  Syntax

  ifelse(test, true_value, false_value)

  Example

  x <- 1:10

  ifelse(x<5 | x>8, x, 0)

Control Structures – Loops

- The most commonly used loop structures in R are 'for', 'while' and 'apply' loops. Less common are 'repeat' loops. The 'break' function is used to break out of loops, and 'next' halts the processing of the current iteration and advances the looping index.
- **For Loop**: flexible, but slow when looping over large number of fields (e.g. thousands of rows or columns)

  Syntax

  for(variable in sequence) {

  statements

  }

  Example: mean

```
mydf <- iris
myve <- NULL
for(i in 1:length(mydf[,1])) {
    myve <- c(myve, mean(as.vector(as.matrix(mydf[i,1:3]))))
}
```

Control Structures – Loops

- Example: condition

```
x <- 1:10; z <- NULL
for (i in 1:length(x)) {
      if (x[i]<5) { z <- c(z,x[i]-1)  } else { z <- c(z,x[i]/x[i])  }
}
```

- Example: stop on condition and print error message

```
x <- 1:10; z <- NULL
for (i in 1:length(x)) {
     if (x[i]<5) { z <- c(z,x[i]-1)  } else { stop("values need to be <5") }
}
```

Control Structures – Loops

- **While Loop**: similar to for loop, but the iterations are controlled by a conditional statement.

  Syntax

  while(condition) statements

  Loop continues until condition returns FALSE.

  Example

```
z <- 0
while(z<5) {
    z <- z+2
    print(z)  }
```

Control Structures – Loops

- **Repeat Loop**

  Syntax

  repeat statement

  Loop is repeated until a break is specified. This means there needs to be a second statement to test whether or not to break from the loop.

  Example

  repeat { z <- 2; z <- z+2; print(z); z <- z+6; print(z); break }

The “Apply” Functions

- **The 'apply' Function Family**

  Syntax

  apply(X, MARGIN, FUN, ARGs)

  X: array, matrix or data.frame; MARGIN: 1 for rows, 2 for columns, c(1,2) for both; FUN: one or more functions; ARGs: possible arguments for function

  Example

  apply(my_matrix, 1, function(i) { my_commands })

  Example for single operation

  apply(iris[,1:4], 2, mean); apply(iris[,1:4], 2, sd)

The “Apply” Functions

- Two-step approach: 1st define function, 2nd use function in apply loop (does the same as above 'for loop'*)

```
x <- 1:10; z <- NULL
test <- function(x) { if (x<5) { x-1 } else { x/x } }
apply(as.matrix(x), 1, test)
```

- One-step approach: does the same as above, but function defined in apply loop

```
apply(as.matrix(x), 1, function(x) { if (x<5) { x-1 } else { x/x } })
```

The “tapply” Functions

- **tapply**

  applies a function to array categories of variable lengths (ragged array). Grouping is defined by vector.

  Syntax

  tapply(vector, factor, FUN)

  Example

```
for (i in 1:4) {
    print(tapply(as.vector(iris[,i]), factor(iris$Species), mean))
}
```

The “lapply” and “sapply” Functions

- Both apply a function on vector or list objects. The function lapply returns a list, while sapply returns a more readable vector or matrix structure.

Example for vector objects

```
z <- seq(1,10, by=2); my_matrix <- matrix(runif(100), ncol=10)
lapply(z, function(x) mean(my_matrix[x,]))
sapply(z, function(x) mean(my_matrix[x,]))
```

Example for list objects

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
lapply(x, mean)
sapply(x, mean)
```

Writing your own functions

- A very useful feature of the R environment is the possibility to expand existing functions and to easily write custom functions. In fact, most of the R software can be viewed as a series of R functions.

  Syntax to define functions

  myfun <- function(arg1, arg2, ...) { function_body }

  The value returned by a function is the value of the function body, which is usually an unassigned final expression.

  Syntax to call functions

  myfun(arg1=..., arg2=...)

Named arguments and defaults

If arguments to called functions are given in the “name=object” form, they may be given in any order. Furthermore the argument sequence may begin in the unnamed, positional form, and specify named arguments after the positional arguments.

Thus if there is a function fun1 defined by

```
fun1 <- function(data, data.frame, graph, limit) {
  [function body omitted]
}
```

then the function may be invoked in several ways, for example

```
ans <- fun1(d, df, TRUE, 20)
ans <- fun1(d, df, graph=TRUE, limit=20)
ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

are all equivalent.

Named arguments and defaults

In many cases arguments can be given commonly appropriate default values, in which case they may be omitted altogether from the call when the defaults are appropriate. For example, if fun1 were defined as

fun1 <- function(data, data.frame, graph=TRUE, limit=20) { ... }

it could be called as

ans <- fun1(d, df)

which is now equivalent to the three cases above, or as

ans <- fun1(d, df, limit=10)

which changes one of the defaults.

Named arguments and defaults

Another frequent requirement is to allow one function to pass on argument settings to another. For example many graphics functions use the function **par()** and functions like **plot()** allow the user to pass on graphical parameters to **par()** to control the graphical output. (See The **par()** function, for more details on the **par()** function.) This can be done by including an extra argument, literally "**...**", of the function, which may then be passed on. An outline example is given below.

```
fun1 <- function(data, data.frame, graph=TRUE, limit=20, ...) {
  [omitted statements]
  if (graph)
    par(pch="*", ...)
  [more omissions]
}
```

Control Utilities for Functions

- **Return**

  The evaluation flow of a function may be terminated at any stage with the 'return()' function. This is often used in combination with conditional evaluations.

- **Stop**

  To stop the action of a function and print an error message, one can use the stop() function.

- **Warning**

  To print a warning message in unexpected situations without aborting the evaluation flow of a function, one can use the function 'warning("...")'.

```
myfun <- function(x1) {
      if (x1>=0) print(x1) else stop("This function did not finish, because x1 < 0")
      warning("Value needs to be > 0")
      }
myfun(x1=2); myfun(x1=-2)
```

Useful Utilities

- Function with optional arguments

```
myfun2 <- function(x1=5, opt_arg) {
        if(missing(opt_arg)) {
              z1 <- 1:10
               } else {
               z1 <- opt_arg }
        cat("my function returns:", "\n")
        print(z1/x1)
        }
myfun2(x1=5) ; myfun2(x1=5, opt_arg=30:20)
```

- **system.time(my_expr)**
  # returns CPU (and other) times that 'my_expr' used
- **date()** #returns the current system time and date

Example

```
two.t <- function(y1, y2) {
        n1  <- length(y1); n2  <- length(y2)
        yb1 <- mean(y1);   yb2 <- mean(y2)
        s1  <- var(y1);    s2  <- var(y2)
        s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
        tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
        tst
      }
x<-rnorm(100); y<-rnorm(100)+1
tstat<-two.t(x,y)
```

Examples

Calculate the function sin(x)/x

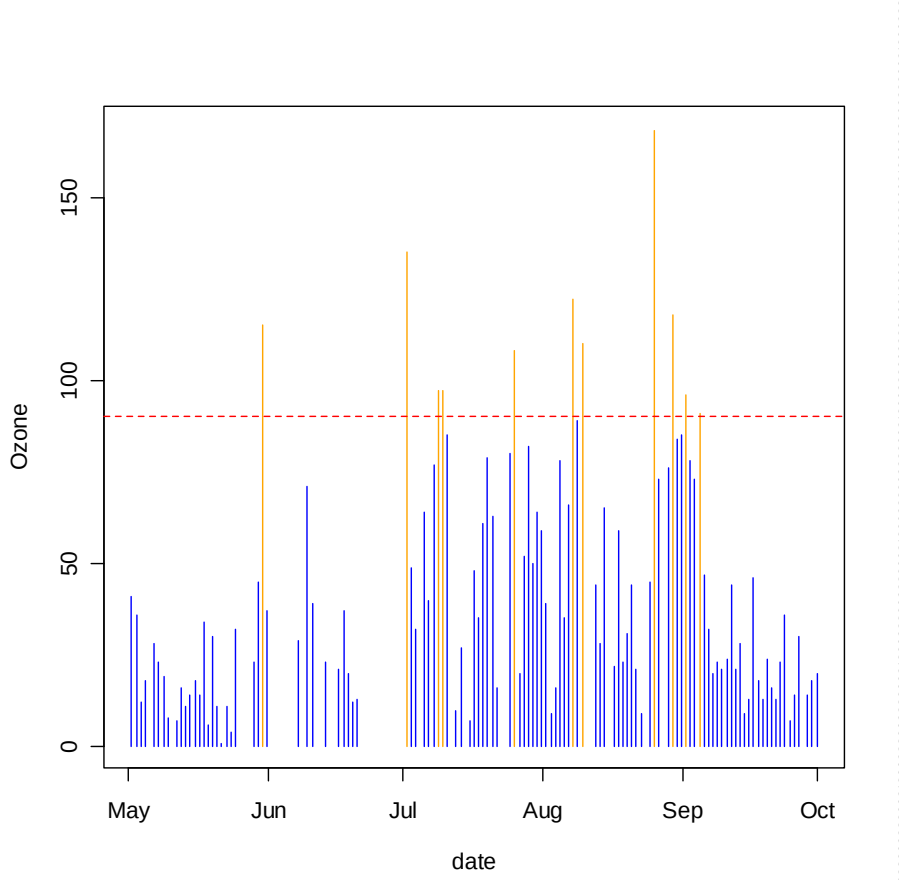
```
sinc  <- function(x){
  if (abs(x) > 1) {    ## x not near 0: calculate in the obvious way.
    s <- sin(x)/x
  }else{   ## x too close to 0 for sin(x) / x to work. Use power series instead.
    s <- 1
    term <- 1
    for( j in seq(3,100,by=2) ){
      term <- term*(-x*x)/(j*(j-1))
      s <- s+term
      if(abs(term) < 1.e-10) break
    }
  }  ## Value returned is value of last expression: s in this case
  s
}
sinc(0.01)
```

Examples

Generate an example data frame with ID numbers and DNA sequences

```
fx <- function(test) {
      x <- as.integer(runif(20, min=1, max=5))
     x[x==1] <- "A"; x[x==2] <- "T"; x[x==3] <- "G"; x[x==4] <- "C"
      paste(x, sep = "", collapse ="")
}
z1 <- c()
for(i in 1:50) {
      z1 <- c(fx(i), z1)
}
z1 <- data.frame(ID=1:length(z1), Seq=z1)
print(z1)
```

Demo




```
data(airquality)
airquality$date<-with(airquality,
ISOdate(1973,Month,Day))

col.level<-ifelse(airquality$Ozone>=90,
"orange","blue")

plot(Ozone~date, data=airquality, type="h",
col=col.level)
abline(h=90,lty=2,col="red")
```

Clustering and Classification in R

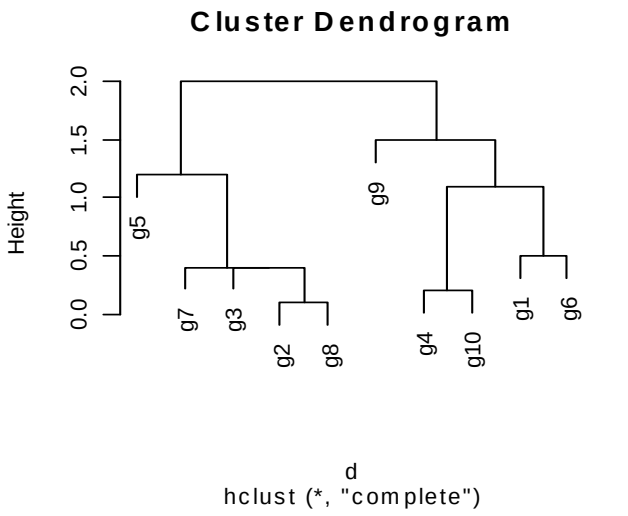
- The cluster package contains clustering code by Rousseuw et al. for a variety of methods, k-means, hierarchical clustering, PAM etc.
- rpart: The CART algorithm for building and pruning classification trees.
- randomForest: a package implements random forest classifiers for classification and regression.
- k-NN: k-nearest neighbour classification is implemented in the class package
- SVM: the e1071 package implements support vector machines.

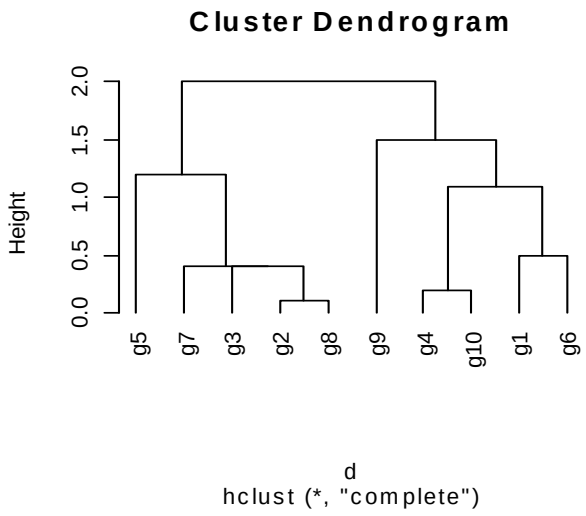
Hierarchical Clustering

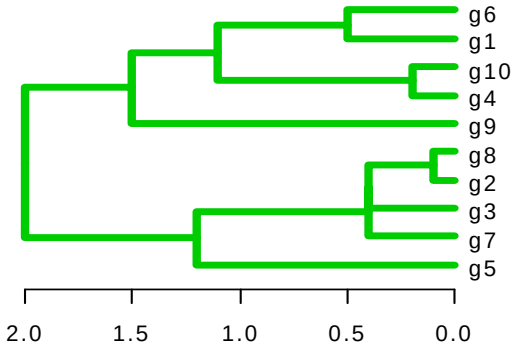
```
y <- matrix(rnorm(50), 10, 5, dimnames=list(paste("g", 1:10, sep=""),
      paste("t", 1:5, sep="")))

# Computes a distance matrix for the rows in the data matrix 'y' using
      the specified distance measure
d=dist(y, method = "euclidean")
c.d <- cor(t(y), method="spearman"); d <- as.dist(1-c.d)

hr <- hclust(d, method = "complete", members=NULL)
par(mfrow = c(2, 2)); plot(hr, hang = 0.1); plot(hr, hang = -1)
plot(as.dendrogram(hr), edgePar=list(col=3, lwd=4), horiz=T)
```


Cluster Dendrogram
Height
0.0
0.5
1.0
1.5
2.0
g5
g7
g3
g2
g8
g9
g4
g10
g1
g6
d
hclust (*, "complete")


Cluster Dendrogram
Height
0.0
0.5
1.0
1.5
2.0
g5
g7
g3
g2
g8
g9
g4
g10
g1
g6
d
hclust (*, "complete")


g6
g1
g10
g4
g9
g8
g2
g3
g7
g5
2.0
1.5
1.0
0.5
0.0

Hierarchical Clustering

- Prints dendrogram structure as text:

  str(as.dendrogram(hr))

- Prints the row labels in the order they appear in the tree

  hr$labels[hr$order]

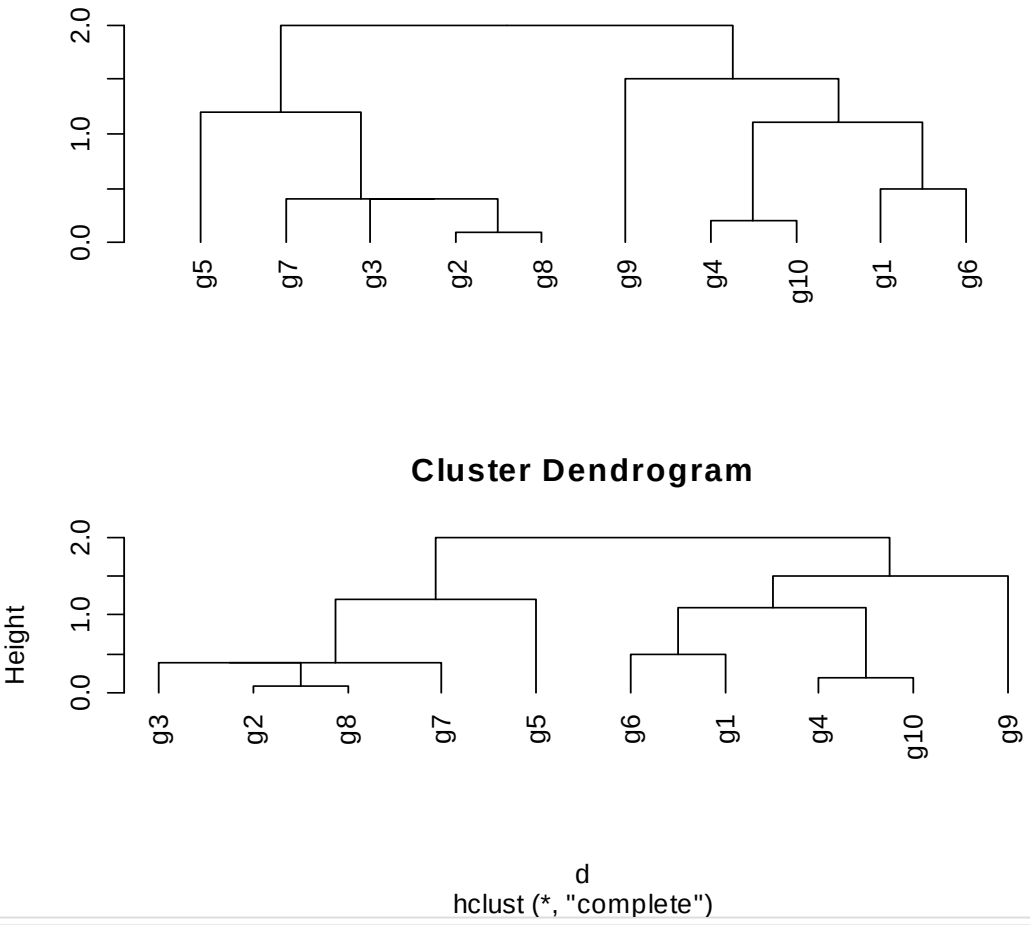
- Reordering a dendrogram:

  library(gclus)

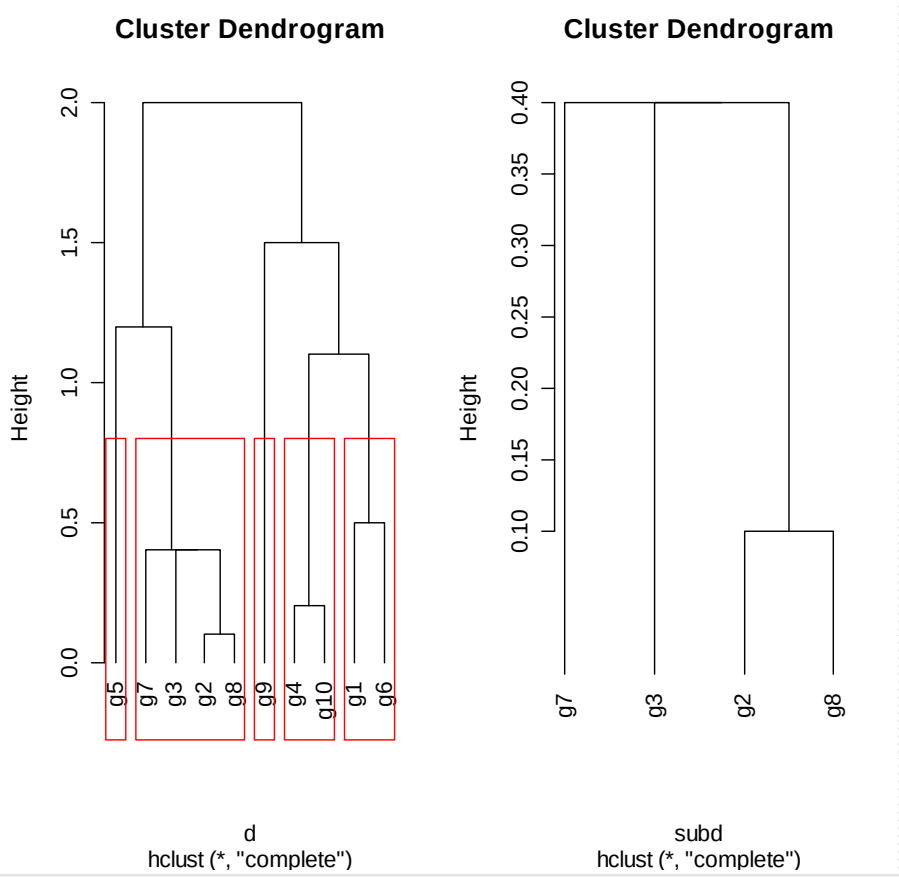
  par(mfrow=c(2,1));

  hrd1 <- as.dendrogram(hr); plot(hrd1);

  hrd2 <- reorder.hclust(hr, d); plot(hrd2,hang=-1);


2.0
1.0
0.0
g5
g7
g3
g2
g8
g9
g4
g10
g1
g6
Cluster Dendrogram
Height
2.0
1.0
0.0
g3
g2
g8
g7
g5
g6
g1
g4
g10
g9
d
hclust (*, "complete")

Subsets Clustering




```
mycl <- cutree(hr, h=max(hr$height)/2)
subcl <- names(mycl[mycl==2]);

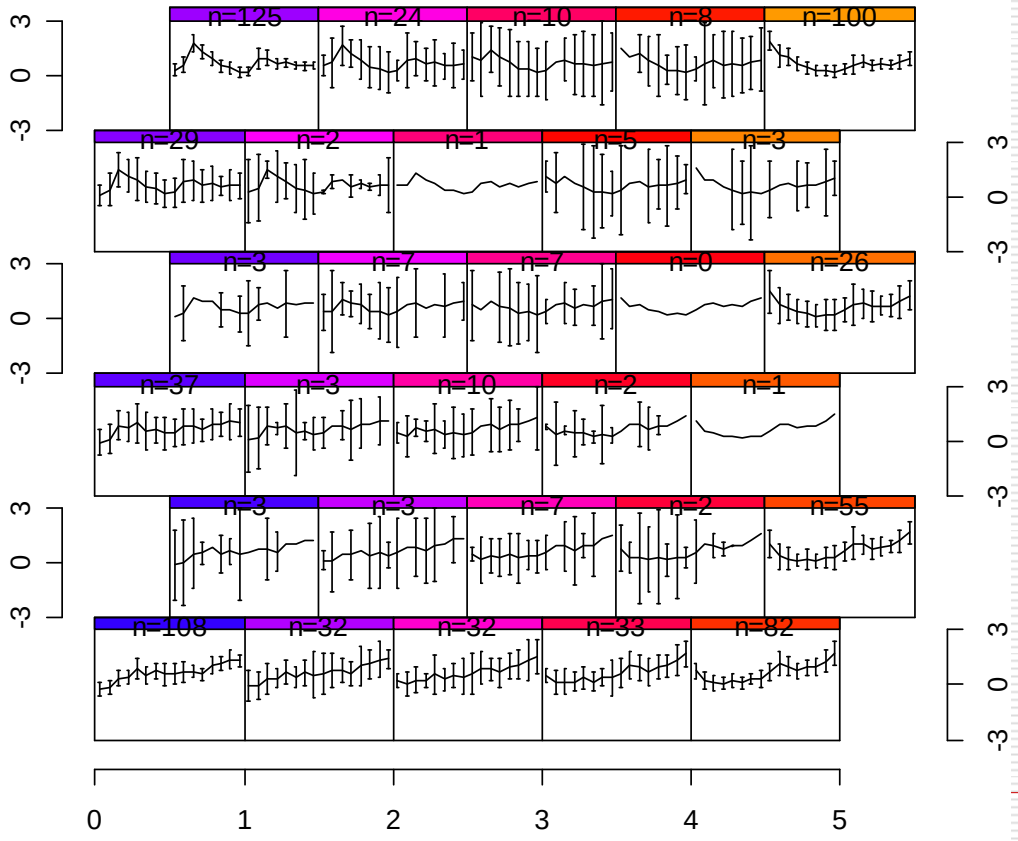
subd <- as.dist(as.matrix(d)[subcl,subcl]);
subhr <- hclust(subd, method = "complete");

par(mfrow = c(1, 2));
plot(hr,hang=-1);
rect.hclust(hr, k=5, border="red");
plot(subhr,hang=-1)
```

Self-Organizing Map (SOM)

- The som package provides the self-organizing map (SOM), known as Kohonen network, which is a popular artificial neural network algorithm in the unsupervised learning area.
- library(som)

```
data(yeast)
yeast <- yeast[, -c(1, 11)]
yeast.f <- filtering(yeast)
yeast.f.n <- normalize(yeast.f) #Normalize the data so that each row has mean 0 and variance 1
foo <- som(yeast.f.n, xdim=5, ydim=6)
foo <- som(yeast.f.n, xdim=5, ydim=6, topol="hexa", neigh="gaussian")
plot(foo)
```

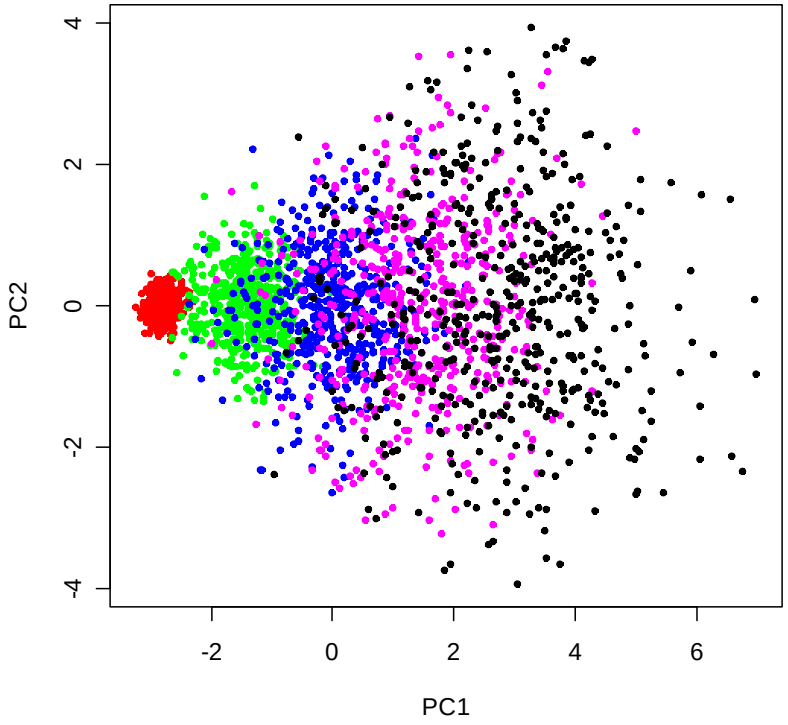

n=125
n=24
n=10
n=8
n=100
n=29
n=2
n=1
n=5
n=3
n=3
n=7
n=7
n=0
n=26
n=37
n=3
n=10
n=2
n=1
n=3
n=3
n=7
n=2
n=55
n=108
n=32
n=32
n=33
n=82
-3
0
3
0
1
2
3
4
5

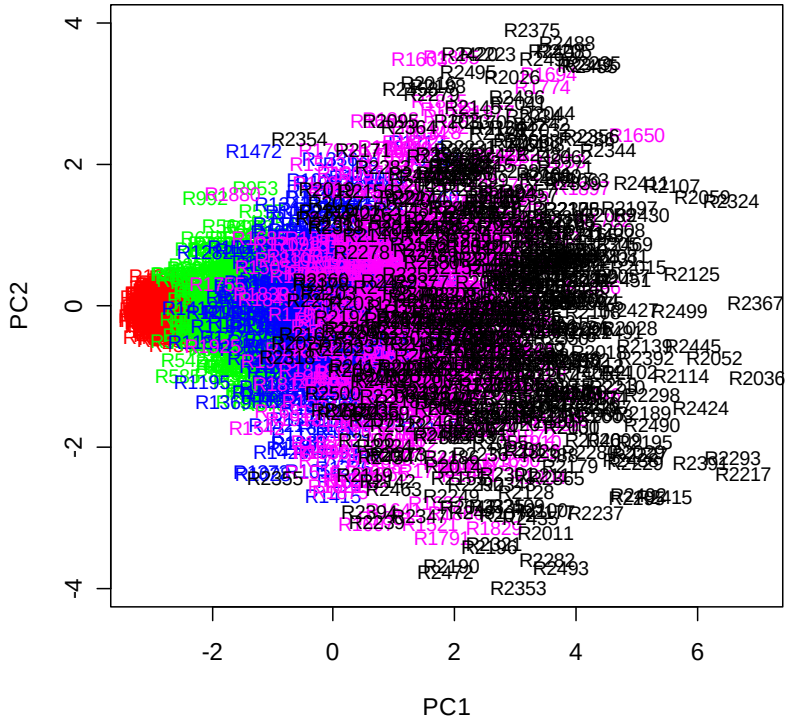
Principal Component Analysis (PCA)

```
z1 <- rnorm(10000, mean=1, sd=1); z2 <- rnorm(10000, mean=3, sd=3);
z3 <- rnorm(10000, mean=5, sd=5); z4 <- rnorm(10000, mean=7, sd=7);
z5 <- rnorm(10000, mean=9, sd=9);
mydata <- matrix(c(z1, z2, z3, z4, z5), 2500, 20, byrow=T,
      dimnames=list(paste("R", 1:2500, sep=""), paste("C", 1:20, sep="")))

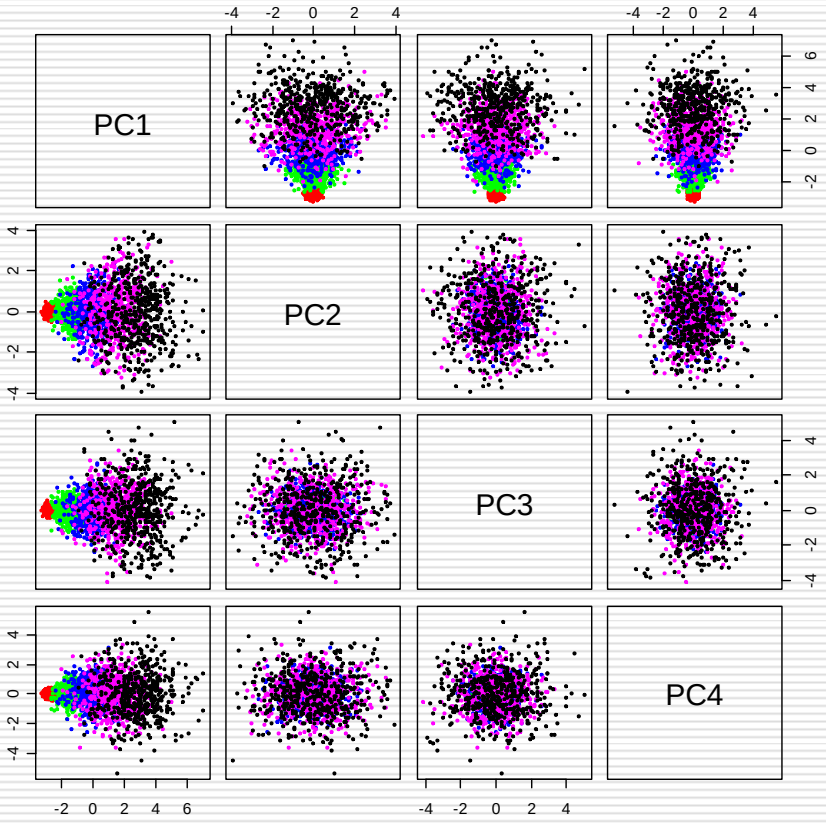
pca <- prcomp(mydata, scale=T)
summary(pca)
x11(height=6, width=12, pointsize=12); par(mfrow=c(1,2))
mycolors <- c("red", "green", "blue", "magenta", "black")
plot(pca$x, pch=20, col=mycolors[sort(rep(1:5, 500))])
plot(pca$x, type="n"); text(pca$x, rownames(pca$x), cex=0.8,
      col=mycolors[sort(rep(1:5, 500))])
```

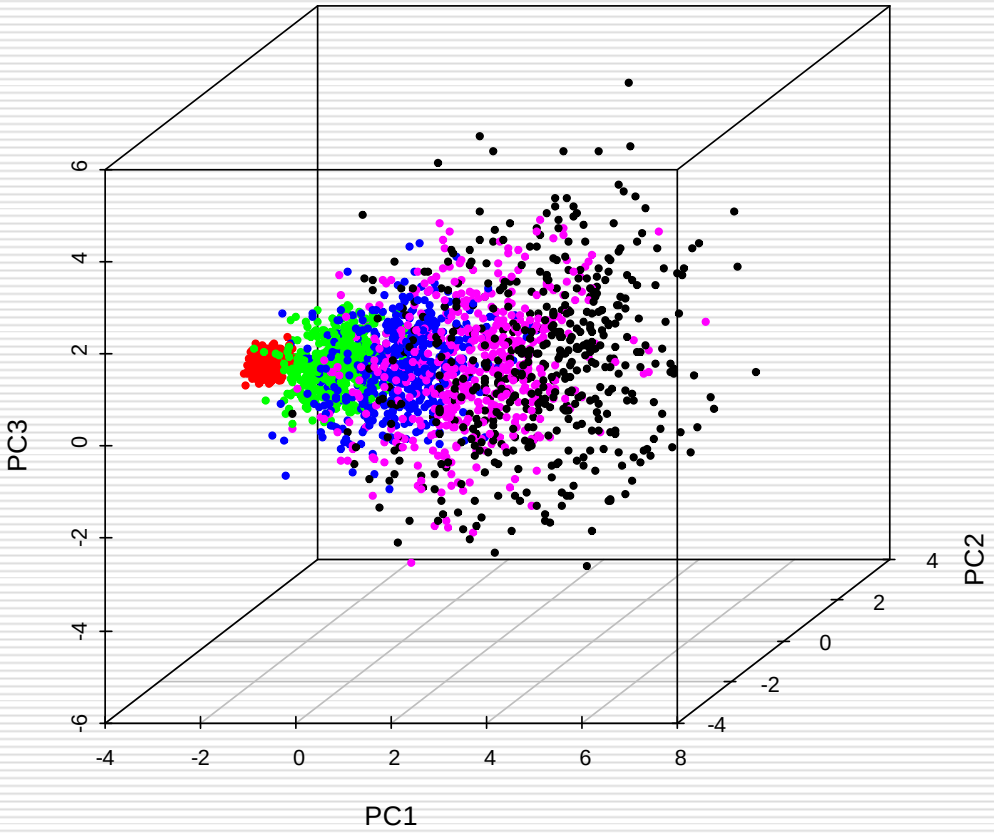
Visualization of First Two Components





Visualization of First Three Components

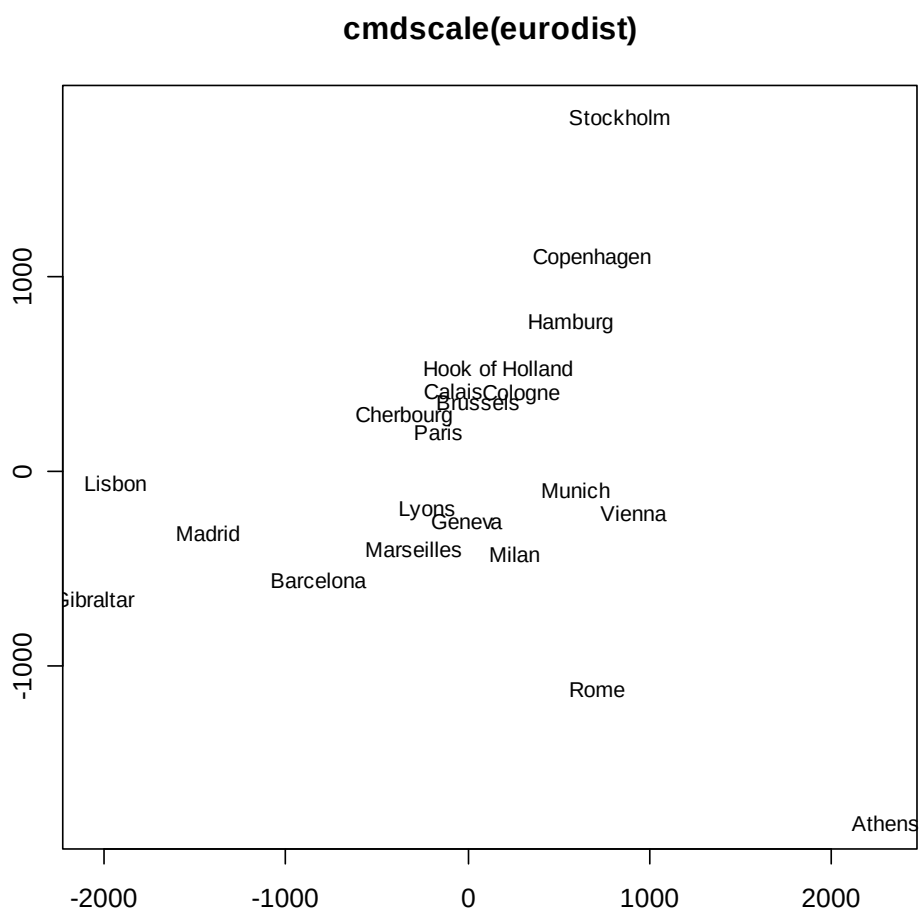




Multidimensional Scaling (MDS)

- Alternative dimensionality reduction approach
- Represents distances in 2D or 3D space
- Multidimensional scaling (MDS) algorithms start with a distance matrix and then assign coordinates for each item in a low-dimensional space to represent the distances graphically. (PCA uses data points)
- In R, cmdscale() is the base function for MDS. Additional MDS functions, sammon() and isoMDS(), can be found in the MASS library.

MDS Demo




```
loc <- cmdscale(eurodist)

plot(loc[,1], -loc[,2], type="n", xlab="",
ylab="", main="cmdscale(eurodist)");

text(loc[,1], -loc[,2], rownames(loc),
cex=0.8)
```

Support Vector Machine (SVM)

- Support vector machines (SVMs) are supervised machine learning classification methods. SVM implementations are available in the R packages kernlab and e1071.

- ```
  library(e1071); data(iris); attach(iris)
  ind=sample(1:150,100)
  model <- svm(Species ~ ., data = iris[ind,])
  x.tr <- subset(iris[ind,], select = -Species); y.tr <- Species[ind]
  model <- svm(x.tr, y.tr)
  print(model); summary(model)
  table(fitted(model), y.tr)

  x.ts <- iris[-ind,-5]
  y.ts <- Species[-ind]
  pred <- predict(model, x.ts)
  table(pred, y.ts)
  ```
  ```