Applying yourself to Environmental data

Clement Kent

clementfkent@gmail.com

the Apply family
applies functions to parts of data

- array data:
  - apply
  - tapply → aggregate
  - outer
- List data:
  - lapply and friends:
    - sapply, mapply, vapply,
    - replicate
    - rapply, dendrapply
    - eapply

Let's get some data

- Temps – a matrix of temperatures no various dates (rows) at different stations (columns)

	Island	Barrie	YYZ
winter	-4	-12	-5
spring	8	6	9
summer	29	28	31
fall	19	15	20

find the mean temperatures in:

- each season, across sites;
- each site, across seasons.

- Now, find the median temperatures

apply applies a function to columns or rows

- apply(Temps,1,mean) # row means
- apply(Temps,2,mean) # column means

- apply(Temps,1,median) # row medians
- apply(Temps,2,sd) # col. standard deviations

apply applies *your* function to columns or rows

- apply(Temps,2,function(x){median(x)-mean(x)})
- apply(Temps,2,function(x){x-mean(x)})
  - # what happens here?

apply applies a function to arrays as well

```
apply(TempsYrly,1,median) #
apply(TempsYrly,2,median) #
apply(TempsYrly,3,median) #
```

robust scaling

```
robustScale<-function(x){(x-median(x)/(IQR(x)/2)}
# like scale, but uses median instead of mean, and
# Inter-quartile Range instead of sd

r<-rnbinom(n=10000,mu=10,size=2) # nice skewed data
hist(r)
histr<-function(x,brks=50,...){hist(x,breaks=brks,col="black",...)}
histr(r) # nice convenience function to have
histr(scale(r),brks=20) # compare with:
histr(robustScale(r) ,brks=20,xlab="number in quadrat")
```

some serious data

- a 20 million year BP series of CO_2 and Northern hemisphere mean temperatures
  - Graversen RG, Drijfhout S, Hazeleger W, van de Wal R, Bintanja R, Helsen M. Greenland's contribution to global sea-level rise by the end of the 21st century. Climate dynamics. 2011;37(7-8):1427-42.

```
r<-CO2_NHtemp[CO2_NHtemp$time> -2e3,] # last 2 Myrs
with(r,plot(dTNH,CO2ppm))
with(r,plot(time,CO2ppm))
t10K<-floor(r$time/10) # 10,000 year lumps
z<-apply(r[,2:3],2,robustScale); z<-aggregate(z, by=list(t10K=t10K),median)
plot(z$t10K,CO2ppm,ty="l",lty=1,col="blue",xlab="time BP (10,000 year units)",ylab="robust z-score")
lines(z$t10K,z$dTNH,col="red")
```

mapply – apply a function with multiple arguments

- *result <- mapply(function, arg1, arg2…argN,moreArgs)*

- mapply(rep, 1:4, 4:1)
- mapply(rep, times = 1:4, x = 4:1)
- netPP<-function(temp,CO2,alpha=0,beta=1){temp*beta*(CO2-alpha)}
- x=mapply(netPP,z$dTNH,z$CO2ppm); plot(x,ty="l",xlab="time",ylab="NPP")
- x=mapply(netPP,z$dTNH,z$CO2ppm,alpha=-2,beta=.5); lines(x,col="red")

with multi-dimensional arrays more fun is possible

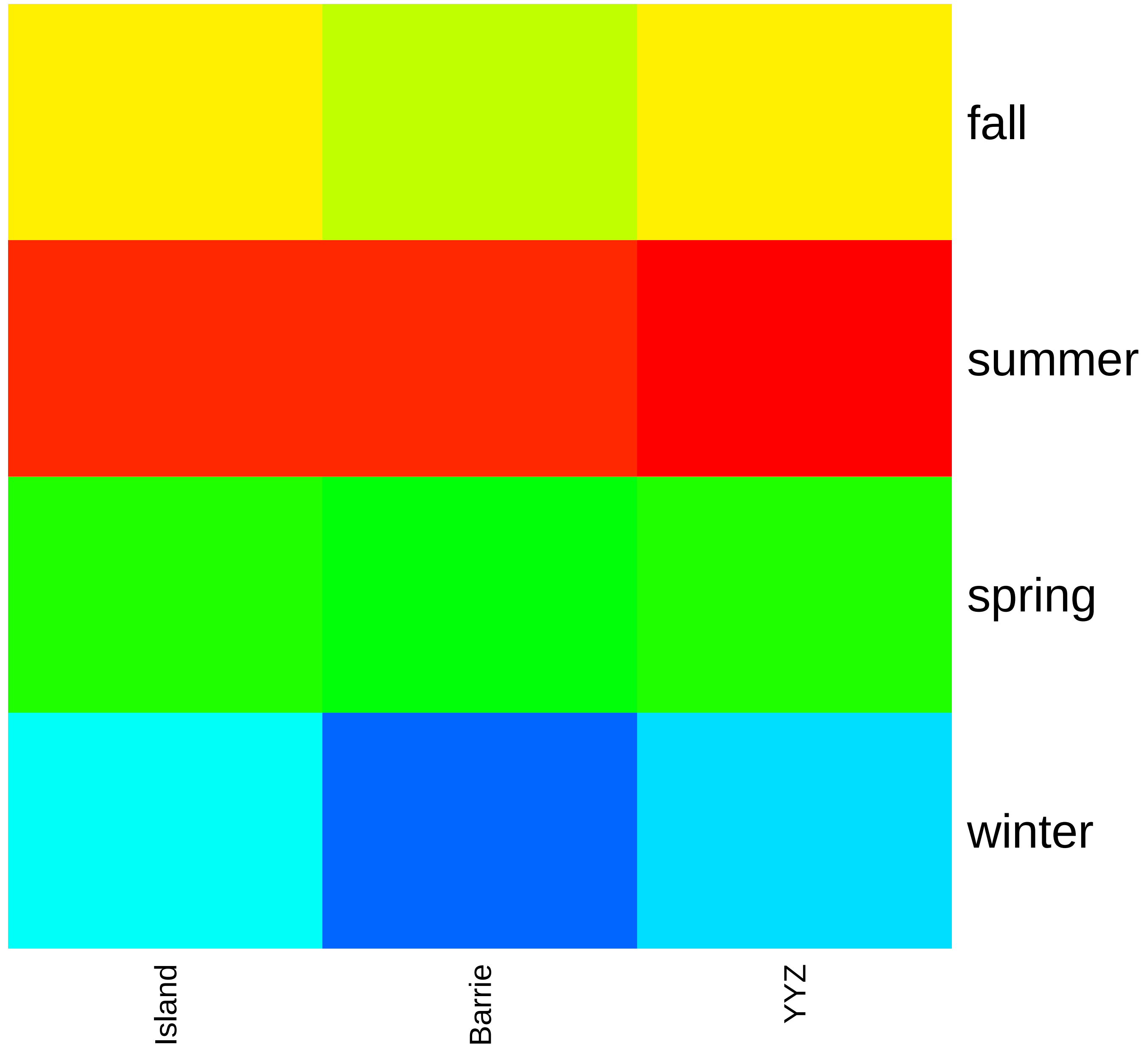
```
apply(TempsYrly,1:2,median) # median how?
apply(TempsYrly,2:3,median) # when?
apply(TempsYrly,c(1,3),median) #  where?
```

a bit of visualization

```
overYears<-apply(TempsYrly,c(1,2),median)

cols<- rev(rainbow(24, start = 0, end = .6))

heatmap(overYears ,col=cols,
        Rowv=NA,Colv=NA,scale="none”,cexCol=1.2)
```


fall
summer
spring
winter
Island
Barrie
YYZ

tapply → aggregate

groups <- as.factor(rbinom(32, n = 16, prob = 0.4))

> tapply(groups, groups, length) #- is almost the same as

7 10 11 12 13 14 15 16

1 2 1 2 2 1 2 5

> tabulate(groups)

[1] 1 2 1 2 2 1 2 5

tapply → aggregate

- tapply(data, factors, FUN = myFunction)
- use when we want to group the data by one or more factors before summarizing each group using
- each group is all the data having a specific combination of values of each of the factors supplied
- use with “ragged” data that doesn’t fit easily in an array

tapply – a simple example

fishSites

```
"Smoke"   "Cache"   "Cache"   "2Rivers" "Rock"    "Smoke"
"Cache"   "Opeongo" "Opeongo" "Canoe"   "Smoke"   "Opeongo"
"Cedar"   "Grand"   "Rock"    "Grand"   "Smoke"   "Sec"
"Grand"   "Grand"   "Rock"    "Mud"     "Opeongo" "2Rivers"
"Opeongo" "Canoe"   "Mud"     "Mud"     "Mud"     "Opeongo"
```

tapply(fishSites,fishSites,length)

```
2Rivers Cache Canoe Cedar Grand  Mud Opeongo Rock Sec Smoke
      2     3     2     1     4    4       6    3   1     4
```

fishSites is both data, and the factor we group on

aggregate instead of tapply

- result<- aggregate(data,by=listOfFactors, function)
- ft1<-tapply(fishCatch$temp,fishCatch[,1:2],mean)
- ft2<-aggregate(fishCatch$temp, by=list(where=fishCatch $sites, when=fishCatch$season), mean)
- ft3<-aggregate(fishCatch$temp, by=fishCatch[,1:2], mean)
- ft4<-aggregate(fishCatch$temp, by=list(when=fishCatch $season, where=fishCatch$sites), mean)
- ft5<-aggregate(fishCatch$temp, by=fishCatch[,2:1], mean)
- ft6<-aggregate(fishCatch[,3:4], by=fishCatch[,2:1], mean)

- Compare these. Where are they the same, where different?

What is N for tapply/aggregated data?

- ft6n<-aggregate(fishCatch[,3:4], by=fishCatch[,2:1], length)
- ft6nt<-tapply(fishCatch[,3], fishCatch[,2:1], length)
- How are these different? where might you use this?

going to “outer” space

- *outer* is very cool. Make yourself into an R astronaut by using *outer* space…
- result<-outer(x,y,myFunction)
  - returns array with as many rows as x, columns as y, and result[i,j]==myFunction(x[i],y[j])
  - classic example: x=1:9; multTable=outer(x,x,”*”)
  - note if using a built-in R function such as +-*/ etc, put it in quotes
  - if using a *named* function (Alex, where are you????) no quotes

what is the closest point in outer space?

- you measured many environmental variables at lots of sampling points.
- Each sampling point has an x and y coordinate (in meters, km, 10^{th} century king’s feet, etc)
- You’ve got a dataframe EnvDat with columns x, y, and N more columns for your N environmental measures (e.g. noon temp, soil carbon, % canopy cover)

myXY is a dataframe of points within the experimental site

Let’s find closest 3 points in EnvDat to each X,Y point.

```
d<-sqrt(outer(myXY$x,EnvDat$x,”-”)^2 + outer(myXY$y,EnvDat$y,”-”)^2)
s<-apply(d,1,order)[,1;3]
# not shown: we could now interpolate weighted noon temp, soil carbon, etc from these 3 nearest points in EnvDat
```

lists are for ragged data

- matrices, arrays all need same number of entries in rows, columns.
- in the real world you may have different numbers of observations at different sites
- lists are for you!
- the i^{th} member of a list L is L[[i]]
- look at variable sites3

sapply applies functions within lists, then simplifies the result

- sapply(sites3,mean)
- sapply(sites3,sd)
- sapply(sites3,function(x){c(m=mean(x),s=sd(x))}
- sapply(sites3,quantile,seq(.2,.8,.1))

lapply doesn’t simplify

- lapply(sites3,mean)
- lapply(sites3,robustScale)

replicate for simulation

- replicate(n_times, expression)
- replicate(3,rnorm(2,0,1))
- replicate(3,rnorm(sample(10,1),0,1))

for another time: recursive applies

- *rapply* recursively applies a function down the levels of a list, most often one representing something like a tree or graph structure
- *dendryapply* does this to dendrogram objects produced by various clustering methods

THANKS!

a less elementary example, Watson!

- you have latitude of your observation sites and the date. How long was the day?
- in Wikipedia, you find the *sunrise equation*:
  - $\cos(w) = -\tan(L) * \tan(D)$
- instantly, you achieve enlightenment, quit graduate school, and start your own cult
- or perhaps you actually want to know the day length, poor unenlightened you!

the Outer space solution

- tanL = tan(LatToR(lat))
- tanD = tan(SunDecl(day,lat))
- cosw = - outer(tanL,tanD, “*”)
- hours = 2 * RadToHour(arccos(cosw))
- check it: dataframe DaySites has lat, day of year

the R *sunrise equation*

- L is Latitude in *radians* (Alex, how do we convert from Google Maps latitudes to radians? help!!!)
- D is the sun *declination* (how much is the sun at noon above where is would be on March 21, in radians…help!!!)
- w is the “hour angle” in radians
  – daylength (hours) = 2 * w / 15

solving the sunrise equation

- you and Alex make a function *LatToR(lat)* to convert latitude in degrees (like 45.1278) to radians
- you and Alex make another function *SunDecl(day, lat)* to convert day of the year (measured from 1=Jan 1) and latitude into sun declination
- you and Alex make a function to convert back from radians into hours….*RadToHour(rads)*
- now, you are ready to rock and roll!