R Training

PVT

11/08/2016

Contents

1 Welcome to introduction to R

The idea of these practical session consists out of preparing and learning about statistical applications and data handling within R

1.1 Basic organisation

First things first, check your version package

```
sessionInfo()
```

Lets load a library into R. First I install the package that I want to use and then load it into the workspace with the `library` function

```r
install.packages("installr")

library(installr)
```

1.1.1 Updating R and packages

Sometimes packages will have been updated and will no longer work on the current version of R, thus we should install a complete new version of R while migrating all our previously used packages.

1. First store all your previously installed packages just in case something goes missing or you want to check what version you were using previously

```r
tmp <- installed.packages()

installedpackages <- tmp[, c(1,3)]
saveRDS(installedpackages, file = "~/Desktop/installed_packages.rds")
```

2. Now Install your new R using the `installr` package

```r
library(installr)

# step by step functions:

# tells you if there is a new version of R or not.
check.for.updates.R()

# download and run the latest R installer
install.R()

# copy your packages to the newest R installation from the one version before it (if ask=T, it will ask
copy.packages.between.libraries()

###----------------OR-----------------###
updateR()
```

2 Creating objects in R and working with Rectangular data sets

We call anything created in R an *object*, and we ALWAYS use the `<-` operator. The `=` operator will be used in a later stage as an equivalence `==` or parameter attributes `=`. Rectangular data sets usually refer to situations where samples are in rows of a data set while columns correspond to variables. There are two main structures for rectangular data: *matrices* and *data frames*.

The main *difference* between these two types of objects is the type of data that can be stored within them. A matrix can only contain data of the same type (e.g., character or numeric) while data frames must contain columns of the same data type.

Matrices are more computationally efficient but are obviously limited.

2.1 Data.frames

```r
rm(list = ls())
# Basic ----------------------------------------------------------------------
```

```r
Response <- c("Very Happy", "Happy", "Not Happy")

Response

Women <- c(15, 5, 3)

Men <- c(35, 15, 14)
Children <- c(23, 35, 32)

HappySurvey <- data.frame(Response, Women, Men, Children)

HappySurvey

# Lets see the structure of
str(HappySurvey)

dim(HappySurvey)

length(HappySurvey)
ncol(HappySurvey)
nrow(HappySurvey)

levels(HappySurvey$Response)
as.numeric(HappySurvey$Response)
```

Tip: Do not work with factors if you can avoid them. Only transform them into factors just before your analyis!

2.1.1 Accessing your data.frame object

Lets calculate the Men mean age

```r
HappySurvey$Men

sum(HappySurvey$Men)

#Note the following syntax, these two are the same:
# calling all rows of column 3

x <- HappySurvey$Men
y <- HappySurvey[, 3]

identical(x,y)
```

2.1.2 Combining

It is important to know how to work with multiple data frames and combine them into one big table of aggregated data.

DO NOTE: the different `data.frame` do need have the same number of *columns* and *colnames*!!

Combining vectors into a `data.frame`

```r
df <- cbind(rnorm(1000),
      rt(1000, df = 10),
      runif(1000))

head(df)

df <- cbind(Norm = rnorm(1000),
      Student = rt(1000, df = 10),
      Unif = runif(1000))

head(df)
```

Do note, that with `data.frame`, the defualt is to make any character a factor!

2.2 Arrays

An array can be thought of as a matrix within R. Rows, then columns as is the same with `data.frames`

```r
Ray <- array(c(1:20), dim = c(2, 5))
length(Ray)

colnames(Ray) <- c("men", "boys", "women", "children", "babies")
rownames(Ray) <- c("Satisfaction", "Communication")

rownames(Ray)
colnames(Ray)

Ray[1,]

RayVec <- Ray[1, c(1,2)]

is.vector(RayVec)
is.matrix(RayVec)

RayVec <- Ray[1, c(1,2), drop = F]

is.vector(RayVec)
is.matrix(RayVec)
```

2.3 Lists

Can be considered to be an bit more of an advanced storing method as it has to do with how the list object stores data. The best thing about list though is that you can store different kinds of data in a list object. They do not have to conform to a structure as with `arrays` and `data.frames`. Thus you will encounter them a lot of the times when you are working with R functions from packages

```r
Odendaal <- list(name = "Hanjo",
                 title = "R master Joda",
                 subject = "R training",
                 university = "Stellenbosch",
                 salary = "$10 million Yen")

Odendaal
```

The Important thing about accessing lists, is that the syntax uses `[[]]` types of breackets

```
Odendaal[['name']]
Odendaal[[1]]
```

3 Basic functions

You need to know various function in order to interact with your data frames. A very big side to this navigation, is thinking about the objects in *indices* which corresponds to thinking about matrices

One of the nice thing about R is that it comes with some built in objects

```
month.abb

is.data.frame(HappySurvey)
```

3.1 Lets see some stats functions

```
# Other useful commands include:
mean(x)
## [1] 21.3
min(y)
## [1] 14
median(x)
## [1] 15
summary(x)

chisq.test(HappySurvey$Men,
           HappySurvey$Women,
           simulate.p.value = T,
           B = 400)
```

3.2 Using indices in R

Finding missing values in your `data.frame`

```
probabilities <- c(.05, .67, NA, .32, .90)
na_true <- is.na(probabilities)

which(na_true)
```

Another really useful thing about this idea of *indices* is using them to subset your *data.frames*

```
IndMenWomen <- colnames(HappySurvey) %in% c("Men", "Women")
col <- which(IndMenWomen)

HappySurvey[ ,col]
```

By now you should see that *indices* works on a LOGICAL basis. These logical `boleans` can also be combined with `&` representing AND, while OR is `|`

```r
set.seed(1988)

NormDist <- rnorm(100, mean = 0, sd = 2)

NormDist > 0

which(NormDist > 0)

which(NormDist > 0 & NormDist < 0.5)

NormDist[NormDist > 0]
```

3.3 Playing with plots

Basic Plots

```r
set.seed(1988)

NormDist <- rnorm(1000, mean = 0, sd = 2)

plot(NormDist, type = "p")
plot(NormDist, type = "b")
plot(NormDist, type = "l")
plot(NormDist, type = "h")

hist(NormDist, probability = T)
lines(density(NormDist), col = 2, lty = 2)
lines(density(NormDist, adjust = 3), col = 3, lty = 3)

densEp <- density(NormDist, kernel = "epanechnikov", bw = 0.5)
densEp2 <- density(NormDist, kernel = "epanechnikov", bw = 0.1)
densEp3 <- density(NormDist, kernel = "epanechnikov", bw = 1)

plot(density(NormDist,
             kernel = "gaussian", adjust = 2),
     main = "Density plot norms",
     ylim = c(0, 0.3))

lines(densEp,
        col=rgb(1, 0, 0, 0.5),
      lwd = 2, lty = 2)
lines(densEp2,
        col=rgb(0, 1, 0, 0.75),
      lwd = 2, lty = 3)
lines(densEp3,
        col=rgb(0, 0, 1, 0.5),
      lwd = 2, lty = 4)

legend("topright", legend = c("bw = 0.5", "bw = 0.1", "bw = 1"), col = c("red", "green", "blue"), lty =
```

We might also want to display a grid of information on multiple plots

```r
# PlayPen
par(mfrow = c(1,3))
plot(density(NormDist,
             kernel = "gaussian"),
     main = "Density plot norms")
polygon(densEp,
        col=rgb(1, 0, 0, 0.2),
        border=T)

plot(density(NormDist,
             kernel = "gaussian"),
     main = "Density plot norms")
polygon(densEp2,
        col=rgb(0, 1, 0, 0.2),
        border=T)

plot(density(NormDist,
             kernel = "gaussian"),
     main = "Density plot norms")
polygon(densEp3,
        col=rgb(0, 0, 1, 0.2),
        border=T)
```

4 Loops and condition statements

4.1 If satements

These are logical statements which can be of big help in subsetting a dataset or by controlling process flow. But first lets see how it helps with variables and subsetting

```r
NormDist <- rnorm(100)

distDivision <- ifelse(NormDist > 0, "positive", "negative")

table(distDivision)

# Subset you data
NormDist[which(distDivision == "positive")]
```

4.2 Loops

This is where programming seperates your average analyst from power users. Having a program that can itteratively conduct specific tasks for you will make your life SOOO easy. Look at the following examples.

Lets take my density creation from earlier, if I wanted different bandwiths, I had to go and create a new line of code for estimation. But what if I want to test how the shape of my distribution changes when I run a 20 different of them??

```r
stdDev <- seq(1, 20, 1)

length(stdDev)
```

```
for(i in 1:length(stdDev))
{
  cat("Now running rnorm with standard deviation: ", i, "\n")
  Sys.sleep(0.3)
}
```

Next, lets add a bit of an `if` condtion into our `for` loop for when something happens and when it doesn't

```
NormDist <- rnorm(20)

for(i in 1:length(NormDist))
{
  cat("Now running rnorm with standard deviation: ", i, "\n")
  if(NormDist[i] > 0){
    print("Postive")
  }else{
    print("Negative")
  }
  Sys.sleep(0.3)
}
```

Remember those `bolean` operators?

```
NormDist <- rnorm(20)

for(i in 1:length(NormDist))
{
  if(NormDist[i] > 0.8){
    print("Postive")
  }else if (NormDist[i] > 0 & NormDist[i] < 0.8) {
    print("Hello World")
  } else {
    print("Negative")
  }
  Sys.sleep(0.3)
}

for(i in 1:length(NormDist))
{
  if(NormDist[i] > 0.8 | NormDist[i] < -0.8){
    print("Outlier")
  } else {
    print("Normal")
  }
  Sys.sleep(0.3)
}
```

Now that we know how a loop can be constructed using the `i` index, we will use this knowledge to generate rnorm data with different standard deviation

```
stdDev <- seq(1, 20, 0.5)

nr_samples <- 1000

length(stdDev)
```

```r
Results <- matrix(NA, nrow = 1000, ncol = 20)

Results <- matrix(NA, nrow = nr_samples, ncol = length(stdDev))
for(i in 1:length(stdDev))
{
  cat("Now running rnorm with standard deviation: ", i, "\n")
  NormDist <- rnorm(nr_samples, mean = 0, sd = stdDev[i])
  Results[,i] <- NormDist
  #Sys.sleep(0.3)
}
```

Things to note:

1. See that I am creating a matrix beforehand to populate the data? This is different when we have different data that needs to be combined (.i.e using `data.frames`). Here we are only using numerical data, so a matrix is perfect
2. See that the `i` in the code is being replaced by the *indexed* number in the vector `stdDev`
3. At the same time I am using my `i` to run through my columns in the `Results` data.frame

4.2.1 Combining loops and plots (and nested loops)

```r
library(colorspace)

n <- length(stdDev)

lineColors <- rainbow_hcl(n, c = 100, l = 70)

lineColorsBlue <- sequential_hcl(n, h = 260, c = c(80, 0), l = c(20, 90), power = 1.2)

LC <- list(lineColors, lineColorsBlue)

plot(density(Results[,15]), ylim = c(0, 0.50),
       main = "Density plots with different Std dev")

for(i in 1:ncol(Results))
{
    lines(density(Results[,i]), col = lineColorsBlue[i])
}
```

We can also nest loops, resulting in what is known as inner and outer loops

```r
LC <- list(lineColors, lineColorsBlue)

par(mfrow = c(1,2))
for(j in 1:2)
{

  plot(density(Results[,15]), ylim = c(0, 0.50),
       main = "Density plots with different Std dev")

  for(i in 1:ncol(Results))
  {
    lines(density(Results[,i]), col = LC[[j]][i])
```

```
  }

}
```

5 Advanced data manipulation (Meet dplyr)

To explore the basic data manipulation verbs of dplyr, we'll start with the built in `nycflights13` data frame

```
library(dplyr)
library(nycflights13)
dim(flights)

head(flights)

flights %>% data.frame %>%  head
```

I hope you notice that when we print this flights data frame, the printout does not include All the rows or all the columns. This is because `dplyr` uses what is known as a `tibble` class of data.frames.

- Doesnt bring all rows (~ 10)
- Doesn't have `stringsAsFactors = T` as default - this is awesome
- Stores the items in lists format for better memory usage
  - Very usefull when working with big objects

5.1 Single verb actions

Dplyr aims to provide a function for each basic verb of data manipulation:

- `filter()` (and `slice()`)
- `arrange()`
- `select()` (and `rename()`)
- `distinct()`
- `mutate()` (and `transmute()`)
- `summarise()`
- `sample_n()` (and `sample_frac()`)

The single most important thing to note with dplyr is its notation!! It uses a pipe operator called `%>%`. This operator should be seen in verbal terms as **and then**.

Quick example of this

```
# Using the flights dataset and then filtering on where month equals to 1
Month <-
  flights %>% filter(month == 1)
```

5.1.1 Filter

`filter()` allows you to select a subset of rows in a data frame. There are 2 ways to use these commands. I prefer the previous version because you can effectively chain commands

```
# Method 1
flights %>% filter(month ==1, day == 1)

# Method 2
```

```r
filter(flights, month ==1, day == 1)

# Base R
flights[flights$month == 1 & flights$day == 1, ]
```

To select rows by position, use `slice()`

```r
slice(flights, 1:10)
```

5.1.2 Arrange rows

`arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them. It takes a data frame, and a set of column names (or more complicated expressions) to order by

```r
flights %>% arrange(year, month, day)

# Order in a desc fashion

flights %>% arrange(desc(arr_delay))
```

In base R you would have had to do the following:

```r
flights[order(flights$year, flights$month, flights$day), ]

flights[order(-flights$arr_delay), ]
```

5.1.3 Select coloumns

Often you work with large datasets with many columns but only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions

```r
# Select columns by name
flights %>% select(year, month, day)

# Select all columns between year and day (inclusive)
flights %>%select(year:day)

# Select all columns except those from year to day (inclusive)
flights %>% select(-(year:day))

flights %>% select(contains("time"))
```

Renaming a variable within your dataset using `dplyr`

```r
flights %>% select(tail_num = tailnum)

flights %>% rename(tail_num = tailnum)
```

5.1.3.1 Select helper functions

- `starts_with()`
- `ends_with()`
- `contains()`

```
flights %>% select(starts_with("arr"))

flights %>% select(ends_with("time"))

flights %>% select(contains("time"))
```

Another very helpfull `helper` function for `select()` is `matches()` - this uses something what is called regular expression matching (Read up on own time)

5.1.4 Summarise

It collapses a data frame to a single row. So, think of this as a column operator

```
flights %>% summarise(delay = mean(dep_delay, na.rm = TRUE))
```

5.1.5 Extracting distinct (unqiue) rows

Here we will use the distinct verb

```
flights %>% distinct(tailnum)

flights %>% distinct(origin, dest)
```

5.1.6 Adding/Creating new column mutate/transmute

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. I find this an amazingly easier way to add columns to your data frame

```
Gain_Spd <-
  flights %>%
  transmute(gain = arr_delay - dep_delay,
            speed = distance / air_time * 60)
```

5.1.7 Sampling

```
sample_n(flights, 10)
sample_frac(flights, 0.01)
```

5.2 Grouping verb actions

The single verbs are useful on their own, but they become really powerful when you apply them to groups of observations within a dataset. In dplyr, you do this by with the `group_by()` function. It breaks down a dataset into specified groups of rows. When you then apply the verbs above on the resulting object they'll be automatically applied "by group". Most importantly, all this is achieved by using the same exact syntax you'd use with an ungrouped object.

NB What becomes really important is that you keep your data TIDY!!! One column, One Metric rule for your data! For example, dont have column names such as Month_1, Month_2 etc! Have a column called month with 1,2,3...,12 in the column

We also introduce a method here called **chaining** which is an important facet in using `dplyr`. Chaining means to stack up various commands in a kind op pipeline, `select` and then `filter` and then `group` and then `mean`....

In our example, lets work out the mean `distance` and `arr_delay` based on the tail number of the plane, and then filter where there has been more than at least 20 flights and mean distance must be greater than 2000

```
one <- flights %>%
  group_by(tailnum)

two <- one %>%  summarise(count = n(),
            dist = mean(distance, na.rm = TRUE),
            delay = mean(arr_delay, na.rm = TRUE))

delay <- two %>%
  filter(delay, count > 20, dist < 2000)

# ------------------ OR CHAIN ------------------


delay <-
  flights %>%
  group_by(tailnum) %>%
  summarise(count = n(),
            dist = mean(distance, na.rm = TRUE),
            delay = mean(arr_delay, na.rm = TRUE)) %>%
  filter(delay, count > 20, dist < 2000)
```

Lets quickly plot this data to see the correlation between distance and how late the plane is

```
library(ggplot2)

ggplot(delay, aes(dist, delay)) +
  geom_point(aes(size = count), alpha = 0.3) +
  geom_smooth() +
  scale_size_area()
```

Next, lets look at how a count distinct over the coloumn could be done with ease with `summarise`. For example, we could use these to find the number of planes and the number of flights that go to each possible destination:

```
flights %>%
  group_by(dest) %>%
  summarise(planes = n_distinct(tailnum),
            flights = n())
```

5.2.1 Top n of grouped variable

Lets see what the top depature delay has been destination

```
flights %>%
  select(dest, dep_delay) %>%
  group_by(dest) %>%
  top_n(1, dep_delay) %>%
  ungroup %>%
```

```
  arrange(desc(dep_delay))

flights %>%
  select(dest, dep_delay) %>%
  group_by(dest) %>%
  top_n(1, dep_delay) %>%
  ungroup %>%
  arrange(dep_delay)
```

5.3 Excercises

A species-level database of life history, ecology, and geography of extant and recently extinct mammals

```
animals <- read.table("http://esapubs.org/archive/ecol/E090/184/PanTHERIA_1-0_WR05_Aug2008.txt",
                      sep = "\t",
                      stringsAsFactors = F,
                      header = T)

mammals <- animals %>% tbl_df()
mammals

# Removing crap
names(mammals) <- sub("X[0-9._]+", "", names(mammals))
names(mammals) <- sub("MSW05_", "", names(mammals))

mammals <- select(mammals, Order, Binomial,
  AdultBodyMass_g, AdultHeadBodyLen_mm, AgeatFirstBirth_d,
  GestationLen_d, HomeRange_km2, LitterSize, MaxLongevity_m,
  SexualMaturityAge_d)

# Find all Capital Letters '([A-Z])', replace those letters with lower case versions along with undersc
names(mammals) <- gsub("([A-Z])", "_\\L\\1", names(mammals), perl = TRUE)

# From the start find underscore and replace with nothing (i.e remove first underscore)
names(mammals) <- gsub("^_", "", names(mammals), perl = TRUE)

mammals[mammals == -999] <- NA

names(mammals)[names(mammals) == "binomial"] <- "species"


glimpse(mammals)
```

1. Select all variables that has to do with the mammal's body (heading should have "body" in it)

```
mammals %>% select(contains("body"))
```

2. Create a column that has the mass of the mammals in KG and then select those with a mass greater than 70kg, only return the order, species and weight

```
mammals %>%
  mutate(adult_body_mass_kg = round(adult_body_mass_g/1000,2)) %>%
  filter(adult_body_mass_kg > 70) %>%
  select(order, species, adult_body_mass_kg) %>%
```

```
  arrange(desc(adult_body_mass_kg))
```

3. Return top 5 biggest carnivors for me (weight > 200 kg) as well as the small ones

```
mammals %>%
  mutate(adult_body_mass_kg = round(adult_body_mass_g/1000,2)) %>%
  filter(order == "Carnivora" & adult_body_mass_kg > 200) %>%
  select(order, species, adult_body_mass_kg) %>%
  top_n(5) %>%
  arrange(desc(adult_body_mass_kg))
```

4. Return smallest and largest carnivor

```
mammals %>%
  mutate(adult_body_mass_kg = round(adult_body_mass_g/1000,2)) %>%
  filter(order == "Carnivora" & is.na(adult_body_mass_kg) != T) %>%
  arrange(desc(adult_body_mass_kg)) %>%
  filter(row_number()==1 | row_number()==n()) %>%
  select(order, species, adult_body_mass_kg)

# Bonus top 5 and bottom 5

mammals %>%
  mutate(adult_body_mass_kg = round(adult_body_mass_g/1000,2)) %>%
  filter(order == "Carnivora" & is.na(adult_body_mass_kg) != T) %>%
  arrange(desc(adult_body_mass_kg)) %>%
  filter(row_number() %in% c(1:5) | row_number() %in% c( (n()-5) : n() )) %>%
  select(order, species, adult_body_mass_kg)
```

5. Find the mean weight of the mammals per order

```
mammals %>%
  mutate(adult_body_mass_kg = round(adult_body_mass_g/1000,2)) %>%
  group_by(order) %>%
  summarise(mean = round(mean(adult_body_mass_kg, na.rm = T), 2)) %>%
  arrange(mean)
```

6. Create a database which constists out of a body-mass-to-length ratio

```
mammals %>%
  mutate(mass_to_length = adult_body_mass_g / adult_head_body_len_mm) %>%
  arrange(desc(mass_to_length)) %>%
  select(species, mass_to_length) %>%
  arrange(desc(mass_to_length))


  # ggplot(. , aes(mass_to_length, x = 1:nrow(.))) +
  # geom_point(alpha = 0.3) +
  # scale_size_area()
```

7. What taxonomic orders have a median litter size greater than 3

```
mammals %>%
  group_by(order) %>%
  summarise(median_litter = median(litter_size, na.rm = TRUE)) %>%
```

```
  filter(median_litter > 3) %>%
  arrange(desc(median_litter)) %>%
  select(order, median_litter)
```

5.4 Playing with GGplots

```
hyp <- mammals %>%
  select(order, species, sexual_maturity_age_d, adult_body_mass_g) %>%
  filter(is.na(sexual_maturity_age_d) != T & is.na(adult_body_mass_g) != T) %>%
  mutate(adult_body_mass_g = adult_body_mass_g/1000)

p1 <- ggplot(hyp , aes(sexual_maturity_age_d, adult_body_mass_g)) +
  geom_point(alpha = 0.3) +
  geom_smooth() +
  scale_size_area()

ann_text <- hyp %>% top_n(5, adult_body_mass_g)
ann_text_age <- hyp %>% top_n(5, sexual_maturity_age_d)

ann_text <- bind_rows(ann_text, ann_text_age)

p1 + geom_text(data = ann_text, aes(label = species, colour=factor(order)), vjust=-0.5, hjust = 1, size


library(ggrepel)
p1 + geom_label_repel(data = ann_text,
                      aes(label = species, colour=factor(order)),
                      size = 2,
                      show.legend = F,
                      point.padding = unit(0.5, "lines"))

p1 + geom_label_repel(data = ann_text,
                      aes(label = species, fill= order),
                      color = "white",
                      size = 2.5,
                      show.legend = T,
                      point.padding = unit(0.5, "lines")) +
  theme_classic(base_size = 16) +
  labs(title = "Relation between Adult Body Weight and Sexual Maturity", x = "Sexual Maturity (days)",
  scale_y_continuous(labels=function(x) format(x, big.mark = " ", scientific = FALSE)) +
  scale_x_continuous(labels=function(x) format(x, big.mark = " ", scientific = FALSE))
```

6 Using financial data in R

```
library(rmsfuns)
packagestoload <- c("xts", "readr", "dplyr", "Dplyr2Xts", "PerformanceAnalytics", "ggplot2")
load_pkg(packagelist = packagestoload)
#write_rds(df, path = "data/Brics.rds")
# This can now be loaded as:
```

```
rm(df)
data <- read_rds("data/Brics.rds")

head(df)
```

Using our dplyr notation we now have learned, we can manipulate this dataset the same we manipulate non-financial data To add a column, use mutate. To summarise columns with functions, use summarise:

```
data %>% select(zar)

# Note the differences below:
data %>% mutate(zarbrz = zar+brz)
data %>% mutate_each( funs( mean(., na.rm = TRUE)), -brz ) # mutates each column, except brz.
data %>% summarise( zarmean = mean(zar,na.rm = TRUE))
data %>% summarise_each( funs(mean(., na.rm = TRUE)), -Date)
data %>% summarise_each( funs(mean(., na.rm = TRUE)), rus, zar )

# Using a vector to call columns, use another _:
cols <- c("rus","zar")
data %>% summarise_each_( funs(mean(., na.rm = TRUE)), cols )

# To filter certain dates, e.g., use filter:
load_pkg("lubridate") # awesome for wrangling dates:

data %>% mutate(Date = ymd(Date) ) %>% # Make Date column a lubridate column
  filter(Date >= ymd(20070806)) %>%
  mutate(Date = as.Date(Date)) # Put back as as.Date if choose to..
```

Please note, Aggregation over time is easiest to do in log returns, while aggregation over assets is more easily done with simple returns. Let's explore this a bit.

Continuously compounded returns should be used in statistical analysis because, unlike simple returns, they are not positively biased... Bacon (2011)(p.29) - Bacon, Carl R. 2011. Practical Portfolio Performance Measurement and Attribution. Vol. 568. John Wiley & Sons

Calculating simple returns of our five TRIs can then simply be done as follows:

```
data %>% arrange(Date) %>%
  mutate_each(funs(./lag(.) - 1), -Date)

# Check that there are no missing values:
colSums(is.na(data))
```

Calculate the log difference of each column:

```
d1 <- data %>%
  mutate_each(funs((log(.)-log(lag(.)))), -Date)
# Notice that I use lag(.) to refer to the previous observation.
```

Another way of calculating returns (and doing more advanced calculations), would be to use the PerformanceAnalytics package. This is a very powerful package that can be used to do all kinds of risk, return and portfolio calculations with.

The problem is that it is often times painful moving from dplyr to xts, so use the Dplyr2Xts package to facilitate this transition

```
load_pkg(c("Dplyr2Xts", "PerformanceAnalytics"))
data %>% Dplyr2Xts(.)
```

```
# The tbldiff has now changed to xts. This format can now be piped into PA:
d2 <- data %>%
  Dplyr2Xts(.) %>%
  Return.calculate(., method = c("compound","simple")[1]) %>%
  .[-1,]
# Note that this calculation is identical to our earlier (by hand) calc:
 all.equal(d1,d2)
```

To get the monthly returns from the weekly above

```
dom <- function(x) format(as.Date(x), "%B")
year <- function(x) format(as.Date(x), "%Y")

# The following creates a Year and Month column:
cumdata <-
  data %>%
  mutate(Month = dom(data$Date), Year = year(data$Date))

# Let's create a vector of all the numeric columns:
num.columns <- colnames( cumdata[,sapply(cumdata, is.numeric)] )

# Now we can use group_by() to calculate the monthly dlog returns for each column:
cumdata <-
  cumdata %>% arrange(Date) %>%
  mutate_each_(funs((log(.)-log(lag(.)))), num.columns) %>%
  group_by(Year, Month) %>% summarise_each_( funs(sum(., na.rm = TRUE)), num.columns) %>%
  .[match(month.name, .$Month),] %>%  # do this, else months are alphabetical
  ungroup()
```

6.1 XTS package and Performance Analytics

```
load_pkg(c("xts", "Dplyr2Xts"))
# First do the xts transformation:
xts.data <- Dplyr2Xts(data)
# Subsetting the xts.data
xts.data['2013'] # Dates in 2013
xts.data['2013/'] # Dates since start of 2013
xts.data['2015-1/2015-2'] # Dates from start of Jan through to end of Feb 2015
xts.data['/2008-9'] # Dates prior to 2008 October (before Crisis period)
first(xts.data,'1 month') # The first one month of xts.data
```

```
table.Stats(xts.data)

table.TrailingPeriods(R = xts.data, periods = c(3,6,12,36,60))
```

7 Plotting in R with ggplot - the tidy way

While there are many and extremely diverse packages that can be used for plotting purposes in R - the one I use most is undoubtedly ggplot2.

```
load_pkg(c("ggplot2", "tidyr", "dplyr"))
```

NB - ggplot wants data to be Tidy

- Each variable has a column
- Each observation is a row

```
retdata <- data %>%
  gather(key = Countries, value = TRI, -Date)
```

Now we can plot this data

```
ggplot(data = retdata) +
  geom_line(aes(x = Date, y = TRI, colour = Countries))
```

Now plot each on its own graph

```
g1 <- ggplot(data = retdata) +
  geom_line(aes(x = Date, y = TRI, colour = Countries)) +
  facet_wrap(~Countries, scales = "free")

g1
```

```
# Remove scales = free to make plots have similar scales...
# To keep the plot specifications, but only plot a subset o/t data:
g1 %+% subset(retdata, Date > as.Date('2009-01-31')) + ggtitle("Post-Crisis TRI")
g1 %+% subset(retdata, Date <= as.Date('2008-08-30')) + ggtitle("Pre-Crisis TRI") +theme_minimal()
```

Now for some boxplots

```
ggplot(data = retdata) + geom_boxplot(aes(x = Countries, y = TRI, fill = Countries) )
```

Saving plots in ggplot

```
g1 %>%
ggsave(filename = "figures/Plot.png",
  plot = .,
  width = 6,
  height = 6,
  device = "png")
```