R Fundamentals - Class # 2

J. Primavera

October 21, 2016

Import, overview and analyse simple data

Lists

Grouped expressions

Control statements: conditional execution

Control statement: repetitive execution

Functions

Probability distributions & simulation

Introduction & setup

- Typing commands each time you need them is not very efficient
- Control statements and user-defined functions help us to move forward and wirte more productive and complex programs
- These slides introduce new R programming features. . .
  . . . along with small tasks you will solve independently
- First, setup a working directory as last time and save in it the csv files

Remind: - you need **dir.create()** and **setwd()**

Import, overview and analyse simple data

Reading data

- read dataset "PolicyPtfSum.csv" and save its value in an object using **read.csv()** function

Remind:

- **read.csv()** takes as argument the path to the file
- R searches by default in your working directory
- **read.csv()** has additional argument we don't need now
- Check them out calling *?read.csv*

Overview data

- overview data stored in this object using **str()** and **summary()**

. . . *What's the maximum value of mclaim?*

. . . how many levels has Category_Car?

- Have a look at the first 10 and last 3 rows using **head()** and **tail()**

Remind:

- **head()** and **tail()** display 6 rows by default. . .

. . . But you can choose how many with argument *n*

Spot missing value

- Check for missing value using **any()** and **is.na()**
- Do it in two steps following this pseudo-code:
- obj1 <- is.na(...); is.na(obj1)
- Now nest the two functions to obtain same result

Remind:

- **is.na()** is vectorized, as most of R functions
- Vectorized functions are used same way with scalars, vectors, matrices, etc.
- What changes is the value returned which depend on input provided...

... what kind of object do you expect to be returned by is.na applied to a dataframe?

Localize missing values

- Discover where are *NAs* using **which()** with *arr.ind* argument set to TRUE
- Have a compact view of the result using **str()** and **head()**...

...what do you think represent each row of this result?

- Discover how many different column indexes we have using **unique()** or **table()** functions

Summarize numerical variables

- Calculate total exposure in this dataset using **sum()**

Remind:

- you can access variables in a dataframe with *$* operator
- Or with the [] operator specifying in the colum slot either...

...an integer index indicating which column to pick

...or a string indicating the exact name of variable to pick

- Calculate average material frequency

Summarize numerical variables from pre-filtered data

- Calculate total exposure and average material frequency only for males

Tips: - Anytime you have a doubt on the exact spell (or position) of a variable consider having a quick view at them with **names()** or **str()**

- You could create a filtered dataset first and summarize it after...

...but this is not very efficient

- Rather get advantage of [] operator which allows you to query rows and columns at same time

Storing analysis' results

- Store last two results in a vector *mystats* using function **c()**
- Visualize the results by printing the vector

Tips: - Consider store separately the two results...

...then use them to fill the vector, our final output

- To visualize the value of an object you can use **print()** (explicit printing) or simply the name of the object (auto-printing)

Import and overview full original portfolio data

- Import dataset "PolicyPtf.csv" using **read.csv()**
- This is the original policy portfolio dataset, not summarized (each row is a policy image)
- Overview it using **str()**, **summary()**, **nrow()**, **head()**, etc.
- Original variables have quite cryptic names:
- Numtppd –> number of material claims
- Numtpbi –> number of BI claims
- Indtppd –> cost of material claims
- Indtpbi –> cost of BI claims

Change variable names

Change these names this way:

- Numtppd –> mclaim
- Numtpbi –> bclaim
- Indtppd –> mcost
- Indtpbi –> bcost

. . . using **names()** function

Remind:

- **names()** allow you to access *names* attribute of the data frame
- recall you want to change only some of the names!

Explore numerical data with histograms

- Create a histogram of the variable *mcost* using **hist()**
- Create another one only for rows with a material claim (*mclaim>0*)
- Keeping this condition, create another one for severity (*mcost/mclaim*)

Make it prettier by

- changing number of bins (*nclass=30*)
- title (*main="yourTitle"*)
- x-axis label (*xlab="yourLabel"*)

Store the plot in a variable called *myhist*

Lists

Presenting lists

- Lists are ordered collection of objects known as its components
- Components can be of any type (logical vector, matrix, function, . . . , whatever!)
- Lists' components are numbered and may be referred to as such
- When components are named they can be referred to by name
- Subscripting operator [. . .] can be used to obtain sublists of a list. . .

. . . but to select single components you need *[[. . .]]* or *$*

- Let's now create some objects we will then store in a list

Create a list

- Create an object named “x” taking value “MyReport”
- Store the objects *mystats*, *myhist* and *x* in a list named *mylist* using **list()**
- Give a name to each component in the list using the form list(namecomponent=component,...)
- Check the number of components in mylist with function **lenght()**

Play with sublists

- Sublist *mylist* keeping only the first entry
- Sublist mylist keeping only the first and third entries
- Play a bit with [...] operator filled with positive (and negative) integer indexes as well as variables' names (since our list has named components)

...Do you think is possible to extract the second element of mystat component with []?

Play with extraction of lists' components

- Extract first component using [[...]], [["..."]] and $ operators
- Experiment partial matching with $ operator
- Extract the second element of first component
- Sum mystat vector contained in mylist

Remind: - [...] for lists returns sub-lists - [[...]] returns lists' components

Grouped expressions

Presenting grouped expressions

- In R every command is a function returning some value
- Commands may be grouped together in braces:

{expr_1 expr_2 }

. . . in which case the value of the group is the result of the last expression in the group evaluated

- Grouped expressions are very useful together with control statements available in the R language

Control statements: conditional execution

if-else statement

- *If-else* conditional construction takes the form

if (expr_1) expr_2 else expr_3

where:

- *expr_1* must evaluate to a single logical value
- *expr_2* is returned when expr_1 returns TRUE
- *expr_3* is returned elsewhere

Play with if-else statements

- Use an *if-else* statement to test whether or not data contain at least one missing values
- then trigger a warning in the first case (something like "Watch out! there are missing values. . . ")
- or simply print a message otherwise (something like "everything's fine, go ahead. . . ")

Tips:

- Use the following functions: **any()**, **is.na()**, **print()**, **warning()**

Play further with if-else statements

- Use an if-else statement to test whether data is of class dataframe
- Then create a variable taking value the number of observation in the data when condition is met and *NA* otherwise

Tips:

- use functions: **class()**, **nrow()**

Vectorized if/else: ifelse() function

- **ifelse()** is a vectorized version of the if/else construct
- Create a variable named *mclaimfg* taking value 1 if mclaim>0 and 0 otherwise using **ifelse()**

Tip: - **ifelse()** has this structure

ifelse(condition, if_true, if_false)

Control statement: repetitive execution

looping

- A *for* loop statment has the form:

for (name in expr_1) expr_2

where:

- *name* is the loop variable
- *expr_1* is a vector expression, (often a sequence like 1:20)
- *expr_2* is often a grouped expression with its sub-expressions written in terms of the loop variable

For loops make that *expr_2* is repeatedly evaluated as name ranges through the values in the vector result of *expr_1*

Play with loops

- Use a *for* statment to loop over integer sequence *1:10* and print iteratively the loop variable

. . . not very useful, right?

- Use for statement to loop over columns of dataset and print the class of each of it
- Repeat it but store the classes in a vector

Tips:

- in grouped expressions auto-printing do not apply (use explicit printing)
- initialize your target vector before, out of the loop
- You can use NULL to initialize an empty object

Play further with loops

- A loop variable can be anything, not only an integer sequence
- Use a for statement to loop over all possible values of Poldur (policy age)
- Then use them to calculate total exposure for each age
- Final output is a numerical vector
- Bind result vector with the values of PolDur to make the result readable
- Order the matrix by policy age
- Round total exposures to have no decimal place

Tips:

- loop may start like this for(i in unique(data$Poldur)) ...
- Use functions **cbind()**, **order()**, **round()**

Alternative (better) ways to loop

- *for* looping is not usually the best way to obtain a result in R
- Code that take a whole object view is likely to be both clearer and faster
- More advanced ways to obtain that same result are available in R
- Previous loops can be obtained with:

```
sapply(fulldt, class)
tapply(X = fulldt$Exposure, INDEX = fulldt$Poldur,
       FUN = sum)
```

Apply family of functions

- *apply* functions implement looping
- You can imagine apply functions as doing the following behind the scenes:
- SPLIT up some data into smaller pieces
- APPLY a function to each piece
- then COMBINE the results
- check ?sapply, ?lapply, ?tapply, ?apply to discover more

Functions

Presenting functions in R

- Functions represent one of the most powerful tool of R
- Somehow the transition between interactive and developing programming mode
- A function is defined by an assignment of the form:

name <- function(arg_1, arg_2) expression

where:

- *expression* usually is a grouped expression that uses the arguments to calculate a value

A call to the function then usually takes the form:

name(arg_1 = value_1, arg_2 = value_2)

- Functions can be treated much like any other object
- They can be nested
- They return last expression evaluated

Functions, arguments and defaults

- Arguments in R can be matched by order and by name
- if you mantain the exact order of function definition then there's no need to specify the names of the arguments
- Arguments can have default values:

```
name <- function(arg_1, arg_2 = NULL, arg_3 = 1)
```

- arguments without default values must be always specified when calling the function

Built-in functions

Most of the functions supplied as part of the R system are themselves written in R and thus do not differ materially from user written functions

Check the R code behind built-in functions like mean, sd, etc by simply typing their names

```
sd
```

```
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x)) x else as.double(x), na.rm =
## <bytecode: 0x00000000167a76e0>
## <environment: namespace:stats>
```

- Many functions call some primitive functions whose code (written in C) is masked

Functions in loaded packages

```
library(e1071)
skewness
```

```
## function (x, na.rm = FALSE, type = 3)
## {
##     if (any(ina <- is.na(x))) {
##         if (na.rm)
##             x <- x[!ina]
##         else return(NA)
##     }
##     if (!(type %in% (1:3)))
##         stop("Invalid 'type' argument.")
##     n <- length(x)
##     x <- x - mean(x)
##     y <- sqrt(n) * sum(x^3)/(sum(x^2)^(3/2))
##     if (type == 2) {
##         if (n < 3)
##             stop("Need at least 3 complete observations.
```

Write your functions

- Write a simple function called *TotExpMyPtf* taking no arguments and returning total exposure of PolicyPtf
- Call the function to see if it works (remember the parenthesis even if no argument is needed. . .)
- Store the value returned by the function in an object named “y”
- Calculate the logarithm of it using **log()**
- Generalize the function by adding an argument being the field to sum up, name it *SomeTotMyPtf*
- Call the function with exposure, mclaim, etc.

More general functions

- Write a function *FreqAnyPtf* returning a vector containing material claim frequency and severity of PolicyPtf

Tips:

- Function will use as arguments: *dataset*, *exposure*, *number of material claims* and *cost of material claims*
- Consider building down various calculations and then combining them in the final output
- Don't forget to call the name of the output at the end of the body because functions in R return the last expressions found

Functions with control structures

- Write a new function *FreqAnyPtfCtrl* identical to previous one but with a control over arguments validity
- Control whether data is a data.frame using is.data.frame()
- *exposure*, *number of material claims* and *cost of material claims* are numeric using is.numeric()
- Test the control structures are working

Remind:

- Remind that integers are also numeric
- Remind you can revert logical expressions with *!* operator

Tips:

- Use *if* statement followed by **stop()** function
- stop() takes as argument a string/message to print whether a condition is met

Functions with optional arguments

- Write a function *GroupedFreqAnyPtf* similar to last one but with an additional argument indicating a categorical variable to group by the results by (univariate analysis)
- argument indicating grouping variable has default to *NULL*
- Return a matrix with two KPIs by row and grouping variable levels by column
- When no value is provided for grouping argument then the same result of previous function (only totals) should be returned

Tips:

- Use a for loop or **tapply()** as seen before
- Consider calculate freq and sev vectors separetely and then combine them with **rbind()** or directly **matrix()**
- Consider make the result prettier and easier to read renaming the dimension of the matrix with **dimnames()**

Functions returning lists

- Sometimes functions need to return more than an object
- In this case lists come very handy
- Write a simple function returning a list with the division, multiplication, addition and difference between any two numbers
- Call the function and store the result in a new object
- Check the result with **str()** function
- Extract the division with $ operator

Probability distributions & simulation

Probability distributions

- R language implements the tables of main probability distributions (normal, poisson, binomial, etc.)
- For each distribution R provide 4 useful functions
- **pnorm()**, evaluate the cumulative distribution function
- **qnorm()**, quantile function (given q, the smallest x such that P(X <= x) > q)
- **dnorm()**, the probability density function
- **rnorm()**, simulate from the distribution
- Replace norm with *pois*, *binom*, *gamma*, ecc. and you have same functions for these other distributions

Let's simulate some deviates

- Simulate 1000 random numbers from normal distribution with mean=2 and sd=1 and store them in an object called "x" (use **rnorm()**)
- simulate 1000 random numbers from gamma distribution with scale=1000 and shape=0.8 and store it in an object called "z" (use **rgamma()**)
- Each distribution has its own parameters
- Sometimes they have default values, sometimes not
- Default parameters for normal distribution are those of standard distr. (mean=0, sd=1)

Visualize distributions

- Plot the histogram of simulated data using **hist()** function
- Adjust the number of bins with *nclass* argument
- Replace the absolute frequency count with the relative one using argument probability=TRUE
- Superimpose the theoretical density on the histogram

```
hist(nn, nclass = 30, probability = TRUE)
curve(dnorm(x, mean = 2, sd = 1), col = "red", add = TRUE)
```

Quantiles & probabilities

- Calculate the 95th quantile of normal deviates using **quantile()** and compare it with the theoretical one using **qnorm()**
- Calculate the probability (relative frequency) of having values above 2000 for gamma deviates (tip: use **length()** function) and compare it with the theoretical one using **pgamma()**

Sampling

- Use **sample()** function to sample randomly 4 elements from the sequence 1:10 without replacement:
- use the size argument
- replacement argument is FALSE by default, so no need to specify it
- Use **sample()** to divide PolicyPtf dataset into a training (80%) and a test (20%) sample