FE8828 Programming Web Applications in Finance

Week 2

Data, visualization, and web: part 1

Dr. Yang Ye <Email:yy@runchee.com>

Nov 09, 2017

Lecture 04: R catch-up

Let’s review some R basics.

- Vector/Matrix/String/Date/Time
- Anonymous function
- List
- Environment
- Pipe
- Load/Save
- Data Frame

R: Vector

Vector/Matrix/List/Dataframe

```
# Create a vector from number
v <- c(1, 3)
v[1] <- 3
v
## [1] 3 3
```

```
# repeat 100 for 10 times.
rep(100, 10)
##  [1] 100 100 100 100 100 100 100 100 100 100
```

R: Matrix

```
# create matrix of 10x10
mat <- matrix(2, 3, 4)
mat
##      [,1] [,2] [,3] [,4]
## [1,]    2    2    2    2
## [2,]    2    2    2    2
## [3,]    2    2    2    2
# set first row to 4
mat[1,] <- 4
# set element (2, 2) to 6
mat[2, 2] <- 6
```

Find element in Vector

- `which()`
- `match()`
- `%in%`

```
data <- 10:1
match(c(1, 3), data)
## [1] 10  8
data[match(c(1, 3), data)]
## [1] 1 3
which(1 == data | 3 == data)
## [1]  8 10
data[which(1== data | 3 == data)]
## [1] 3 1
```

Check whether element exists

- False case when element doesn’t exist

```
match(c(11, 31), 10:1)
## [1] NA NA
which(11== 10:1 | 31 == 10:1)
## integer(0)
```

```
if (all(c(1, 33) %in% 1:3)) {
  cat("Found all\n")
}

if (any(c(1, 33) %in% 1:3)) {
  cat("Found one/some.\n")
}
## Found one/some.
```

Random

```
# Norm random number
rnorm(3, mean = 10, sd = 3)
## [1]  6.575252 11.226616 13.571712
```

```
# Uniform random number
runif(3)
## [1] 0.2201473 0.1328510 0.3524160
```

```
# Sample
sample(1:10, 10, replace = F)
##  [1]  8  4  5  9  1  6  7  3  2 10
# To Be/Not to Be
sample(c(T, F), 10, replace = T)
##  [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE
# Throw a dice
sample(1:6, 10, replace = T)
##  [1] 5 4 5 6 4 4 4 2 5 4
```

Print

- `cat(paste0(..., "\n"))` is what I used most.
- `"\n"` is appended to the end to create a line break.
- `paste0/paste` can use to create new strings from any data types.
- `paste0` combines any thing without space. `paste` uses space, by default.
- `paste0/paste` with `collapse` helps with vector to print them in one line.
- `paste0/paste` works with all types of data.

```
x <- c(Sys.Date(), Sys.Date(), Sys.Date())
cat(paste0("Current dates is ", x, ".\n"))
## Current dates is 2017-11-08.
##  Current dates is 2017-11-08.
##  Current dates is 2017-11-08.
cat(paste0("Current dates is ", paste0(x, collapse = ", "), ".\n"))
## Current dates is 2017-11-08, 2017-11-08, 2017-11-08.
```

String

```
# sub-string
# substr(x, start, stop)
substr("The fox jumps.", 6, 6 + 5 - 1)
## [1] "ox ju"
```

```
# paste0/paste to concatenate string/convert to string
new_string <- paste0("This is ", "cat")
new_string <- paste0("This is ", "cat", sep = "a")
new_string <- paste0(1:3, sep = "a")
```

```
# toupper/tolower
toupper("big")
## [1] "BIG"
tolower("LOWER")
## [1] "lower"
```

Find/Replace string in string

```
# grepl: Find, returns T or F
grepl("A", "ABC", fixed = T)
## [1] TRUE
grepl("D", "ABC", fixed = T)
## [1] FALSE
```

```
# sub: replace for one time
# sub(pattern, replace, string,...)
# fixed = T means use fixed string. Not regular expression
sub("D", "ABC", "DDD", fixed = T)
## [1] "ABCDD"
# gsub: replace for all
gsub("D", "ABC", "DDD", fixed = T)
## [1] "ABCABCABC"
```

Find/Replace String with Regular Expression (RE)

If you start to use regular expression, sub/grepl becomes super powerful.

```
# If we need to find `Start` appearing the beginning of the string
grepl("^Start", "Start with me")
## [1] TRUE
grepl("^Start", "me Start")
## [1] FALSE
```

```
# To find something in the end
sub("X$", "Z", "XYZ ends with X")
## [1] "XYZ ends with Z"
```

Match/Extraction with Regular Expression (RE)

Match (RE)

```
sub("[^\\_]+\\_.*", "", "USDCNY_M1")
## [1] ""
```

- `[^\\_]`: Character not containing _. Because _ is a special character, we quote it with two backslashes.
- +: One or more
- `.`: Any character
- *: none or more.

Match/Extraction with Regular Expression (RE)

Extraction (RE)

```
# Rough cut
sub("([^\\_]+)\\_.*", "\\1", "USDCNY_M1")
## [1] "USDCNY"
```

```
# Nice cut
sub("([^\\_]+)\\_(.*)", "\\1 \\2", "USDCNY_M1")
## [1] "USDCNY M1"
```

```
# Wonderful cut
sub("([^\\_]+)\\_([[:alpha:]])([[:digit:]])", "\\1 \\2 \\3", "USDCNY_M1")
## [1] "USDCNY M 1"
```

Cheatsheat is available at https://www.rstudio.com/resources/cheatsheets/

Date

```
# Create date
dt1 <- as.Date("2017-11-03")
dt1
## [1] "2017-11-03"
dt2 <- Sys.Date()
dt2
## [1] "2017-11-08"
```

```
library(lubridate)
```

```
# Date is such a central role in finance.
# More function about date can be found in package `lubridate`
# Create date with lubridate, a package which provides lots of date functions.
ymd(20171003)
## [1] "2017-10-03"
ymd("20171003")
## [1] "2017-10-03"
```

Date: format code

We can use codes for convert date to/from string.

- %Y/%y: four-digit year/two-digit year
- %m: month in number
- %b/%B: month in abbreviation/full, i.e. Jan/January.
- %d: day

```
format(Sys.Date(), format = "%Y/%m/%d")
## [1] "2017/11/08"
```

```
as.Date("2017-11-03", format = "%Y-%m-%d") # %m for number month
## [1] "2017-11-03"
as.Date("2017-Nov-03", format = "%Y-%b-%d") # %b for the 3-letter month
## [1] "2017-11-03"
as.Date("03Nov2017", format = "%d%b%Y")
## [1] "2017-11-03"
```

Other functions from `lubridate`

```
library(lubridate)
# Change a date
x <- as.Date("2017-10-10")
month(x) <- 1
x
## [1] "2017-01-10"
```

```
# Set to the end of the month
day(x) <- days_in_month(x)
```

Business days

Use package `bizdays`

```
# install.packages("bizdays")
library(bizdays)
```

```
# 'weekends' is a calendar of weekdays
bizdays("2017-10-16", "2017-10-30", "weekends")
## [1] 10

# add bizdays
add.bizdays("2017-11-03", 5, "weekends")
## [1] "2017-11-10"

# generate all business days between.
# You will find this useful for later financial application.
bizseq("2017-11-10", "2017-12-01", "weekends")
##  [1] "2017-11-10" "2017-11-13" "2017-11-14" "2017-11-15" "2017-11-16"
##  [6] "2017-11-17" "2017-11-20" "2017-11-21" "2017-11-22" "2017-11-23"
## [11] "2017-11-24" "2017-11-27" "2017-11-28" "2017-11-29" "2017-11-30"
## [16] "2017-12-01"
```

Calendar

If not provided, start.date is by default the first holiday and end.date is the last holiday. So we provide them here.

```
create.calendar(name="Singapore", holidays = c(as.Date("2017-10-18")),
                start.date = as.Date("2017-01-01"), end.date = as.Date("2071-01-01"),
                weekdays = c("saturday", "sunday"))

bizdays("2017-10-13", "2017-10-30", "weekends")
## [1] 11
# One day less
bizdays("2017-10-13", "2017-10-30", "Singapore")
## [1] 10
```

Write function

Input parameters

```
func1 <- function() { }

func2 <- function(input1, input2) { }

# Param input1 is default to 1
func3 <- function(input1 = 1, input2) { }

func4 <- function(input1, input_type = c("int", "char"))
{
  # This would check wheher input_type is set to one of the pre-set values.
  input_type = match.arg(input_type)
}

func5 <- function(in1, in2) {
  if (in1 < 0) {
    return(0)
  } else {
    return(in1 + in2)
  }
}
```

Write function

```
# The last value before function finishes will be returned automatically. No need to use re
func5 <- function(in1, in2) {
  if (in1 < 0) {
    0
  } else {
    in1 + in2
  }
}

# Unless there is extra steps before
func6 <- function(in1, in2) {
  if (in1 < 0) {
    return(0) # if we have 0 here, it's not the last step before function exits.
  } else {
    res <- in1 + in2
  }

  res <- res * 3
  res
}
```

Exercise

Write functions to do

- Determine leap year?
- Print the list of month names in abbreviation or full
- How many working days in Singapore in 2017?

Time

Convert time to character/string

- %H: hour
- %M: minute
- %S: second

```
format(Sys.time(), format = "%H%M")
## [1] "0043"
format(Sys.time(), format = "%H:%M:%S")
## [1] "00:43:24"
format(Sys.time(), format = "%H:%M:%S")
## [1] "00:43:24"
library(lubridate)
ymd_hms("2011-12-31 12:59:59")
## [1] "2011-12-31 12:59:59 UTC"
```

Change time, lubridate provides `hour`, `minute`

```
x <- Sys.time()
x
## [1] "2017-11-08 00:43:24 +08"
```

```
hour(x) <- 12
x
## [1] "2017-11-08 12:43:24 +08"
minute(x) <- 3
x
## [1] "2017-11-08 12:03:24 +08"
minute(x) <- 123 # what will happen?
x
## [1] "2017-11-08 14:03:24 +08"
```

List

```r
# Create list with list() function
# Nameless list
# list[_n_] => item by order
a <- list(3, 4)
a[[1]]
## [1] 3
a[[2]]
## [1] 4

# Named list, you can use $ and [ operators
# list[[]]: gives back a value
# list$name => list[["name"]]
a <- list(a = 3, b = 4)
a[[1]]
## [1] 3
a[[2]]
## [1] 4
a[["a"]]
## [1] 3
a$a
## [1] 3
```

List

```
# When you want to use a number as key, use backtick
list_of_strikes <- list()
list_of_strikes$`65` <- 3
list_of_strikes$`60` <- 4

# if a name doesn't exist in the list
a$c
## NULL
# Use `is.null()` to check
if (is.null(a$c)) {
  cat("c doesn't exist in list a\n")
}
## c doesn't exist in list a
```

List’s Usage - I

```
# List can be used as map/dictionary.
# Map
basket <- sample(c("Apple", "Orange", "Pear"), 100, replace = T)
fruit_count <- list()
for (b in basket) {
  if (is.null(fruit_count[[b]])) {
    fruit_count[[b]] <- 1
  } else {
    fruit_count[[b]] <- fruit_count[[b]] + 1
  }
}
fruit_count
## $Pear
## [1] 33
##
## $Orange
## [1] 30
##
## $Apple
## [1] 37
```

List’s Usage - 2

```
# Let's write a generic function to do this
add_to_map <- function(map, key, value) {
  if (is.null(map[[key]])) {
    map[[key]] <- value
  } else {
    map[[key]] <- map[[key]] + value
  }
  map
}

# You may copy function add_to_map to every file that you want to use this kind of dictiona
fruit_count <- add_to_map(fruit_count, "Pomelo", 12)
fruit_count
## $Pear
## [1] 33
##
## $Orange
## [1] 30
##
## $Apple
## [1] 37
##
## $Pomelo
## [1] 12
```

List’s Usage

```
# Use case 1: Use list to pass data in or out.
do_lots_of_work <- function(a, b, c) {
}
# pass in
do_lots_of_work <- function(lst) {
  lst$a + lst$b
}
# pass out
ret_lots_of_work <- function() {
  return(list(a = a, b = b))
}

res <- ret_lots_of_work()
res$a
## $a
## [1] 3
##
## $b
## [1] 4
res$b
## [1] "Apple"
```

```
# Case 2: configuration
app_config <- list(MAX = 10, MIN = 10, DISPLAY_RESULT = T)

do_lots_of_work <- function(app_config) {
  app_config$MAX
}
```

Object

```
# Object
# Define class with attributes.
vanilla_option <- setClass("vanilla_option",
                           slots = c(type = "character",
                                     strike = "numeric",
                                     underlying = "numeric"))
# Create object, either way
opt1 <- new("vanilla_option", type = "c", strike = 100, underlying = 100)
opt2 <- vanilla_option(type = "c", strike = 100, underlying = 100)

# Use @ to visit member. or,
opt1@type
## [1] "c"
slot(opt1, "strike")
## [1] 100
```

Work with objects

```
# Generate a vector of options
opts <- sapply(1:10000, function(x) {
                       vanilla_option(type = sample(c("c", "p"), 1),
                                      strike = round(runif(1) * 100, 0),
                                      underlying = round(runif(1) * 100, 0)) } )

# install.packages("fOptions")
library(fOptions)

start <- Sys.time()
# GBSOption also returns an object. We just need its price attribute.
res1 <- sapply(opts, function(o) {
  (GBSOption(o@type, o@underlying, o@strike, Time = 1,
             r = 0.01, b = 0, sigma = 0.3))@price
})
cat(as.numeric(Sys.time() - start))
## 2.713003
head(res1, n = 4)
## [1] 3.597272e-04 3.960199e+01 1.130114e-01 3.231863e+00

# Alternatively to sapply, we can use map* functions from purrr package
# map is a generic function that returns a list
# map_dbl is for result of double, it would return a vector
res2 <- purrr::map_dbl(opts, function(o) {
  (GBSOption(o@type, o@underlying, o@strike, Time = 1,
             r = 0.01, b = 0, sigma = 0.3))@price
})
head(res2, n = 4)
## [1] 3.597272e-04 3.960199e+01 1.130114e-01 3.231863e+00
```

Anonymous Function

```
# Function that's defined in-place, which doesnt' need to have a name.
(function(x) { print(x) }) (3)
## [1] 3
# if there is only one line, you can skip { }
(function(x) print(x)) (3)
## [1] 3

# For longer functions, you can make it multi-lines.
(function(x) {
  if (x > 3) {
    print(x)
  } else {
    print(x - 3)
  }
})(3)
## [1] 0
```

purrr::map and sapply Function

```
# These two are equivalent.
res1 <- purrr::map(1:10, function(x) { rnorm(x, n = 10) })
# function(x) func(x) can be simplied as func.
res2 <- purrr::map(1:10, rnorm, n = 10)
head(res1, n = 1)
## [[1]]
##  [1]  2.04172131  1.15437251  0.48023393 -0.20466246  1.56425258
##  [6]  2.76269333  2.05444062  2.54427530 -0.04313395  1.68816826
# purrr:map returns a list()
```

```
# This is what we really want to do. Generate ten normal distribution and get their mean.
# rnorm(n, mean = 0, sd = 1). Where doesn't input go to?
res <- purrr::map(1:10, rnorm, n = 1000)
map_dbl(res, mean)
##  [1] 0.9938541 1.9212187 3.0139946 4.0068545 4.9709724 5.9927339 6.9607755
##  [8] 7.9686268 8.9490252 9.9611648

# sapply achieves the same as purrr::map, a bit slower.
# Package purrr succeeds original R base.
sapply(1:10, function(x) x ^ 2 )
##  [1]   1   4   9  16  25  36  49  64  81 100
sapply(1:10, function(x) `^`(x, 2) )
##  [1]   1   4   9  16  25  36  49  64  81 100
sapply(1:10, function(x) `^`(2, x) )
##  [1]    2    4    8   16   32   64  128  256  512 1024
```

Read/Write data

```
# set working directory
setwd("C:/TEMP")
# Save this_is_var1 to a file
saveRDS(this_is_var1, file = "C:/TEMP/DATA/data.Rds")
# Load a variable from a file. `new_loaded` is the name given to it.
new_loaded <- readRDS(file = "C:/TEMP/DATA/data.Rds")
```

- On Windows, use double slashes `\\` or single backslash `/`. e.g. `C:\\TEMP\\DATA`, `C:/TEMP/DATA`
- On Mac, use backslash `/Users/.../`

R: data frame

The basic structure of a data frame:

- There is one observation per row and
- Each column represents a variable, a measure, feature, or characteristic of that observation.
- In summary, **2D table**

```
df <- data_frame(date = seq(as.Date("2017-01-01"), as.Date("2017-01-10"), by = "day"),
                 stock = replicate(10, paste0(sample(LETTERS, 3, replace = T), collapse = "
                 quantity = round(runif(10) * 10000 ,0))
# df["date"]: gives a data frame
# df[["date"]]: gives value
# df$date: same as [["date"]]

# Get three rows
df[c(3, 6, 9), , drop = F]
```

```r
# Get three columns
df[, 1, drop = F]
```

```
# This would return a vector
df[, 1, drop = T]
## Warning: drop ignored
```

```

# Use column names
df[, c("date", "quantity"), drop = F]
```

R: data frame

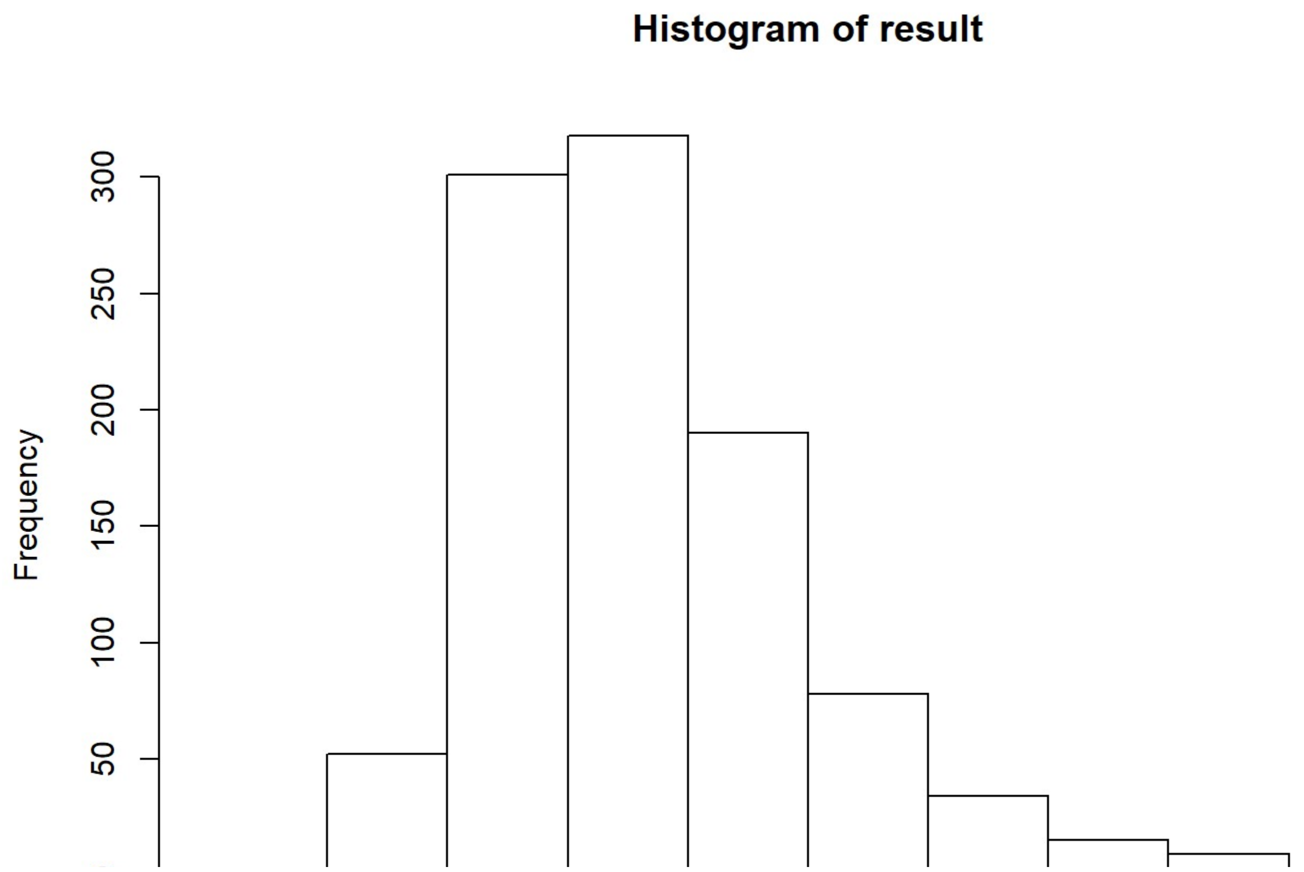
Common functions for data frame

```
View()
head()
tail()
str()
nrow()
ncol()
dim() # returns both nrow and ncol
colnames()/rownames()
```

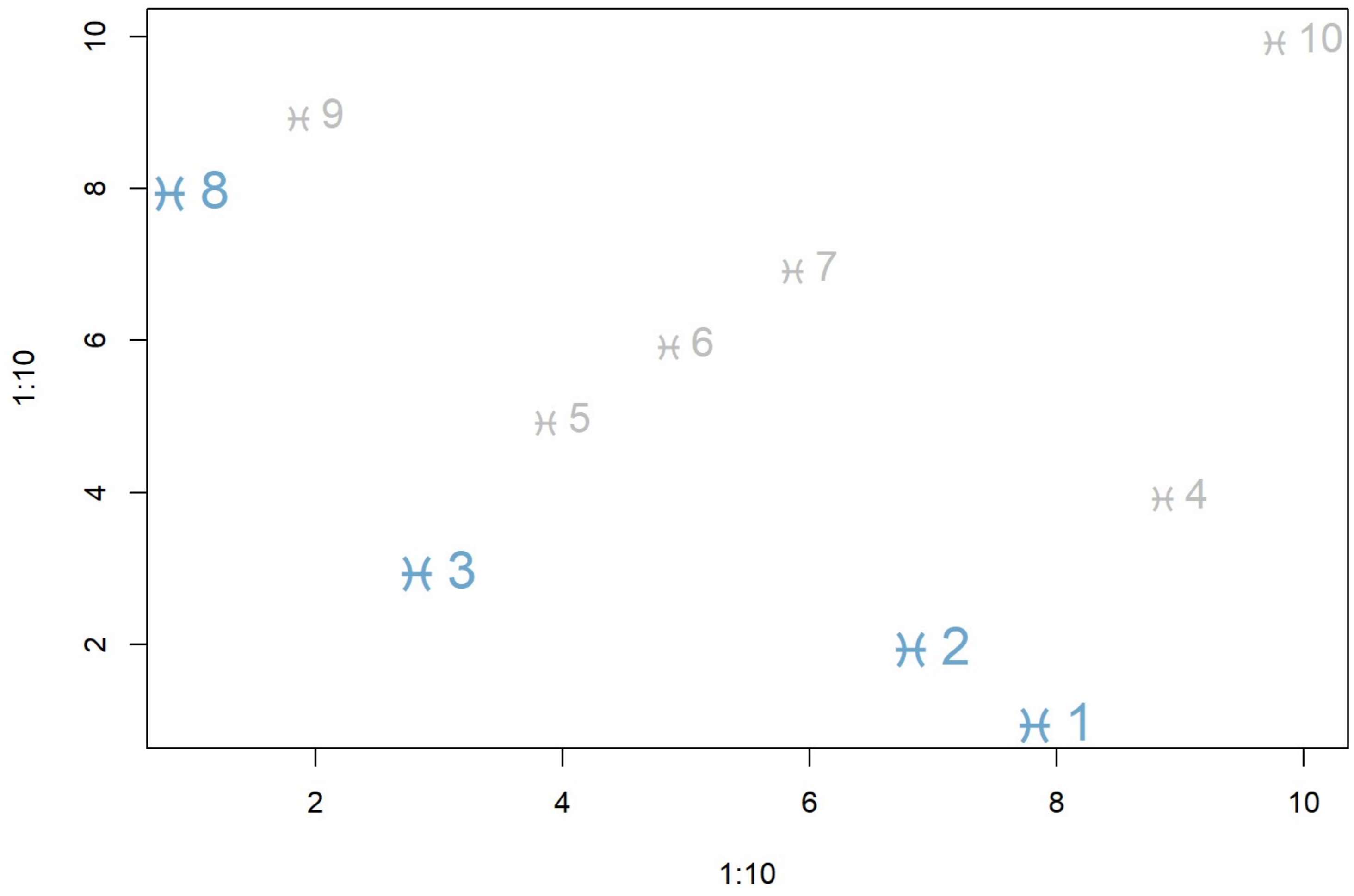
Birthday Problem

- With different weights to the month
- N simulation

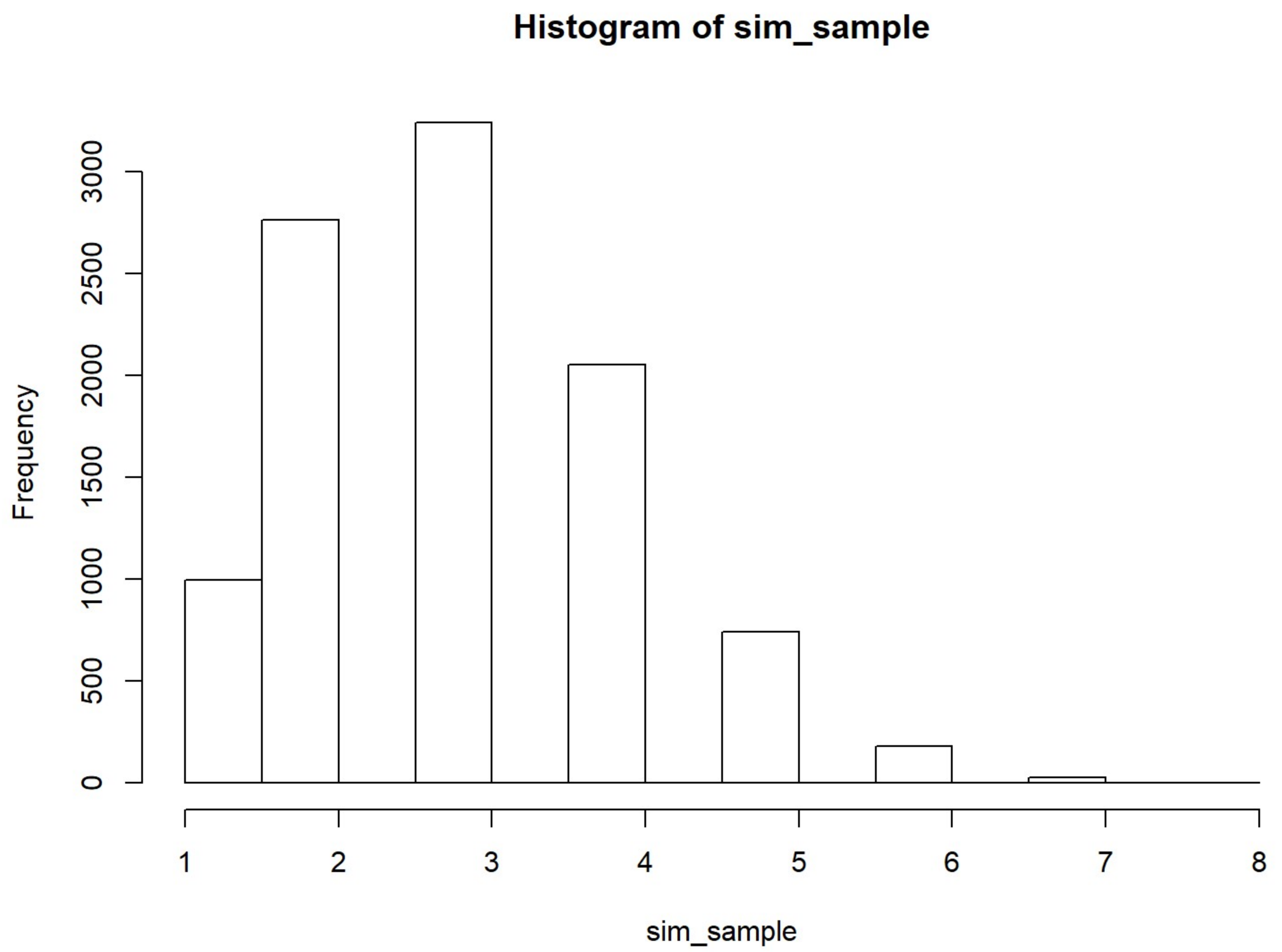
```
## [1] 14
## [1] 101
## [1] 37.084
```



Fastest Fish Problem



```
## res_sim: 2.9431
## res_ana: 2.92896825396825
```


Histogram of sim_sample
Frequency
0
500
1000
1500
2000
2500
3000
1
2
3
4
5
6
7
8
sim_sample

Lecture 05: Shiny

Minimalist

```
library(shiny)
ui <- fluidPage("Hello World")
server <- function(input, output, session) { }
shinyApp(ui = ui, server = server)
```

Think around Input and Outputs

```
ui <- fluidPage(
  titlePanel("Hello World with a Histogram"),
  # Input() functions
  numericInput("num", "Number of Sample", value = 30),
  # Output() functions
  plotOutput("hist")
)
```

Input

All input function follow such function signature except for input-specific parameters.

```
inputXXX(inputId = "input name", label = "label to display", ...)
```

- numericInput
- textInput
- passwordInput
- slideInput
- selectInput
- dateInput

Reference: https://shiny.rstudio.com/reference/shiny/1.0.5/

Output

All output function follow such pattern.

```
yyyOutput(outputId = "output name")
```

- textOutput("text")
- verbatimTextOutput("text_orignal")
- tableOutput("t1")
- dataTableOutput("t2")
- plotOutput(outputId = "hist", width = "400px", height = "400px")
- uiOutput("uiX")

plotOutput: I suggest to set width and height to fixed size so we need extra parameters. For others, outputId is good enough.

Server

Sever is to fill the content of output

```
server <- function(input, output, session) {
  # Enable either one of two
  output$hist <- renderPlot({ hist(rnorm(100)) })

  if (FALSE) {
    output$hist <- renderPlot({
      title("a normal random number histogram")
      hist(rnorm(input$num))
    })
  }
}
```

shinyApp = UI + Server

UI and Server combines to be a ShinyApp. UI is to run the same for each browser/client. Server is separate between different users.

```
shinyApp(ui, server)
```

Reactivity Kicks In

- Reactivity: `input$num ------> output$p1`
- Reactivity links input to the output like a data flow.

Reactive values work together with reactive functions.

1. Reactive function responds. `input$x => output$y`
2. Reactive value notifies.
   `input$x => expression() => output$y`

Reactivity - I

Reactivity is enabled by placing inputXXX in renderXXX function.

```
library(shiny)

ui <- fluidPage(
  numericInput("num", "Num", 100),
  # numericInput("mean", "Mean", 5),
  # numericInput("sd", "SD", 3),
  numericInput("lambda", "Lambda", 1),
  plotOutput("p1")
)

server <- function(input, output, session) {
  output$p1 <- renderPlot({
    # hist(rnorm(input$num, mean = input$mean, sd = input$sd))
    hist(rpois(n = input$num, lambda = input$lambda))
  })
}

shinyApp(ui, server)
```

Reactivity - 2

- We use `observeEvent` to observe button action, and `isolate` to cut down the link of `inputXXX` in `renderXXX`, so button can work.
- If we remove `isolate`?

```
library(shiny)

ui <- fluidPage(
  numericInput("num", "Num", 10),
  actionButton("go", "Go"),
  plotOutput("p1")
)

server <- function(input, output, session) {
  observeEvent(input$go, {
    output$p1 <- renderPlot({
      hist(rnorm(isolate(input$num)))
    })
  })
}

shinyApp(ui, server)
```

Reactivity - 3

We can add a reactiveValue with `eventReactive`

```
library(shiny)

ui <- fluidPage(
  numericInput("num", "Num", 10),
  actionButton("go", "Go"),
  plotOutput("p1")
)

server <- function(input, output, session) {
  data <- eventReactive(input$go, {
    hist(rnorm(input$num))
  })

  output$p1 <- renderPlot({ data() })
}

shinyApp(ui, server)
```

Output

For tableOutput

```
output$t1 <- renderTable(iris)

output$t1 <- renderTable({
  some input..
  output is a data frame.
})
```

For dataTableOutput (Dynamic table)

```
output$t2 <- renderDataTable(iris)
```

For plotOutput

```
output$p2 <- renderPlot({ plot(runif(1000), runif(1000)) })
```

For textOutput and verbatimTextOutput

```
output$t3 <-  renderText({ "foo" })
output$t4 <- renderPrint({
  print("foo")
  print("bar")
})
```

Example: Shiny-24

```
library(shiny)
library(DT)

ui <- fluidPage(
  h3("t1"),
  tableOutput("t1"),
  hr(),
  fluidRow(
    column(9, h3("dt1"),
           dataTableOutput("dt1")),
    column(3,   h3("x4"),
           verbatimTextOutput("x4"))),
  hr(),
  fluidRow(
    column(8, h3("dt2"),
           dataTableOutput("dt2")),
    column(4, h3("p5"),
              plotOutput("p5")))
)

options(error = function() traceback(2))

server <- function(input, output, session) {
  output$t1 <- renderTable(iris[1:10,], striped = T, hover = T)
  output$dt1 <- renderDataTable(iris, options = list( pageLength = 5))
  output$x4 <- renderPrint({
      s = input$dt1_rows_selected
      if (length(s)) {
        cat('These rows were selected:\n\n')
        cat(s, sep = ', ')
      }
    })

  output$dt2 <- renderDataTable(iris,
                                options = list(pageLength = 5),
                                server = FALSE)
  output$p5 <- renderPlot({
    s <- input$dt2_rows_selected
    plot(iris$Sepal.Length, iris$Sepal.Width)
    if (length(s)) {
      points(iris[s, c("Sepal.Length", "Sepal.Width"), drop = F],
             pch = 19, cex = 1, col = "red")
    }
  })
}

shinyApp(ui, server)
```

Debug Shiny

- Debug in R Studio
- Clear all variable to run Shiny in R Studio
- debugSource, if you use other source code

Shiny Summary

- Reactive is about wiring input and output
- Connect from receiver: plot/tabulate for data
- Connect from trigger: button, isolate to create a Chinese wall

Shiny Assignment

1. Add a selectInput for different color names, returned from `colors()`.

   plot(1:10, pch = 19, cex = 1, col = “skyblue1”)

2. Create a Bond Schedule

- Inputs: start date, tenor, coupon rate, yield to maturity.
- Output: coupon schedule (ignore public holidays), amount in table and plot. NPV

$$NPV = \frac{Cashflow1}{(1+yield)^1} + \frac{Cashflow2}{(1+yield)^2} + \ldots + \frac{LastCashflow}{(1+yield)^n}$$

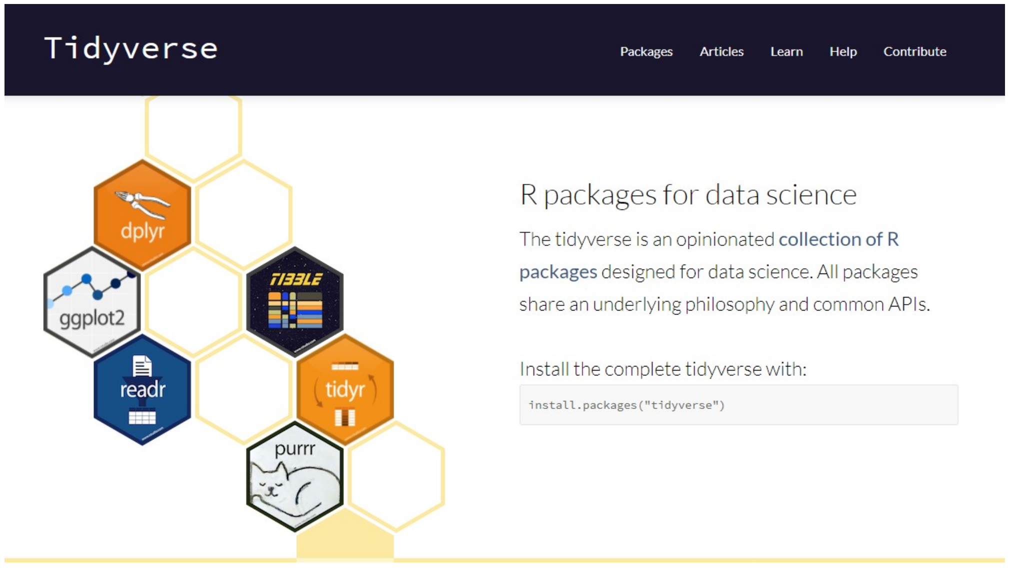
For a Bond with fixed coupon

$$BondPrice = Coupon * \frac{1-(\frac{1}{(1+yield)^n})}{yield} + \left[MaturityValue * \frac{1}{(1+yield)^n}\right]$$

Lecture 06: Data

Tidyverse

install.packages("tidyverse")



R packages for data science

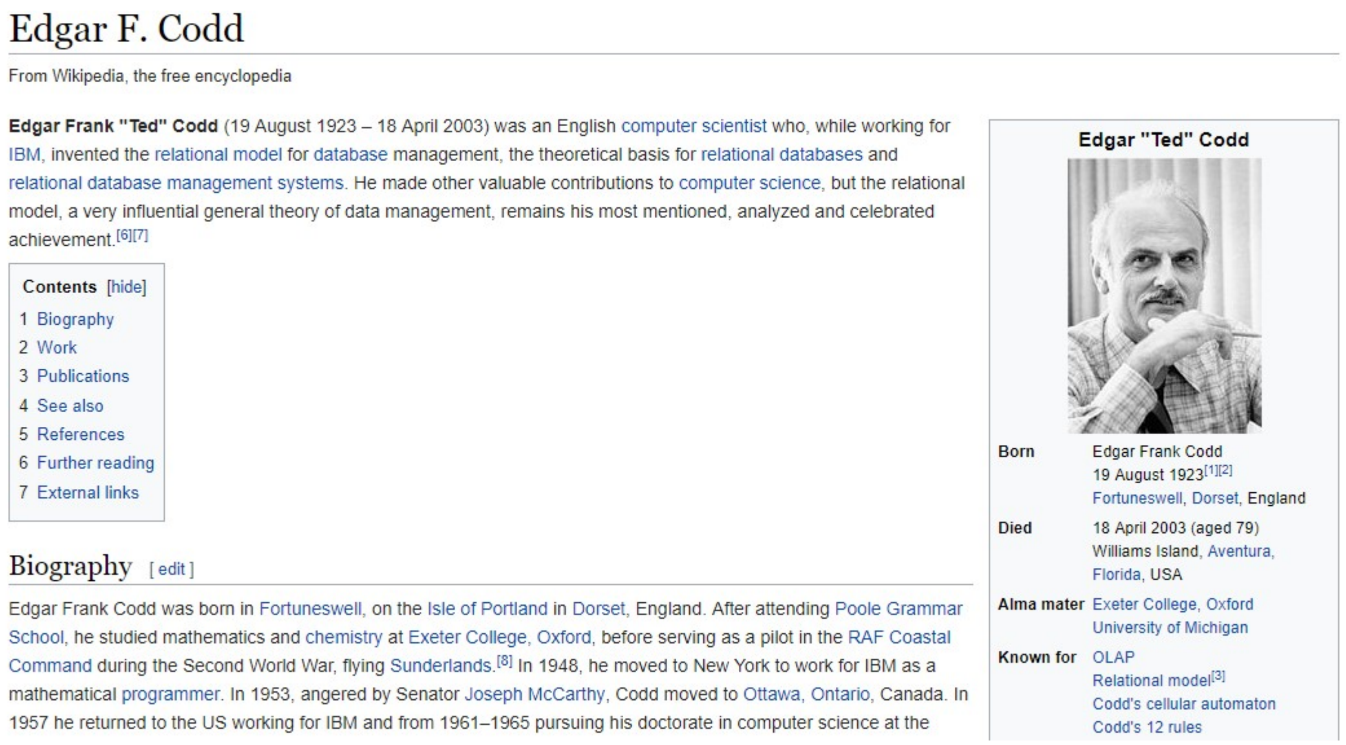
The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying philosophy and common APIs.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

SQL

Let’s start from SQL, which first appeared in 1974; 43 years ago.



Edgar F. Codd

From Wikipedia, the free encyclopedia

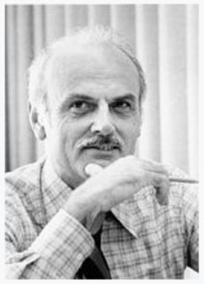
Edgar Frank "Ted" Codd (19 August 1923 – 18 April 2003) was an English computer scientist who, while working for IBM, invented the relational model for database management, the theoretical basis for relational databases and relational database management systems. He made other valuable contributions to computer science, but the relational model, a very influential general theory of data management, remains his most mentioned, analyzed and celebrated achievement.[6][7]

Contents [hide]

Biography [edit]

Edgar Frank Codd was born in Fortuneswell, on the Isle of Portland in Dorset, England. After attending Poole Grammar School, he studied mathematics and chemistry at Exeter College, Oxford, before serving as a pilot in the RAF Coastal Command during the Second World War, flying Sunderlands.[8] In 1948, he moved to New York to work for IBM as a mathematical programmer. In 1953, angered by Senator Joseph McCarthy, Codd moved to Ottawa, Ontario, Canada. In 1957 he returned to the US working for IBM and from 1961–1965 pursuing his doctorate in computer science at the

Edgar "Ted" Codd



Born	Edgar Frank Codd 19 August 1923[1][2] Fortuneswell, Dorset, England
Died	18 April 2003 (aged 79) Williams Island, Aventura, Florida, USA
Alma mater	Exeter College, Oxford University of Michigan
Known for	OLAP Relational model[3] Codd's cellular automaton Codd's 12 rules

CRUD: Create Read Update Delete

Data engineering was born around 70s with SQL.

SQL does CRUD

```
# Select everything from Shops.
SELECT * FROM Shops;

# Select with a filter
SELECT * FROM Shops WHERE size = "Big";

# Select with a filter and order
SELECT * FROM Shops WHERE size = "Big" ORDER BY Name;

# Select with a filter, order, group and summary function `sum`
SELECT Region, sum(Sales) FROM Shops WHERE size = "Medium" GROUP BY Region;

# Insert a new record to Shops.
INSERT into Shops (Name, Region, Sales) VALUES ("Costco", "North", 123456, ...);

# Update a field
UPDATE Shops SET Sales = Sales + 1000 WHERE Name = "Costco";

# Delete from Shops with a filter
DELETE from Shops WHERE Sales < 1000
```

Data frame does CRUD

```
df <- data.frame(a = 1:10, b = 10:1)
# Filter:
df[which(df$a == 3 | df$b == 3), , drop = T]
```

```
df[match(3, df$a), , drop = T]
## $a
## [1] 3
##
## $b
## [1] 8
df[, match("b", colnames(df)), drop = T]
##  [1] 10  9  8  7  6  5  4  3  2  1

# Insert
rbind(df, df)
```

```r
# Delete
df[-(which(df$a == 3 | df$b == 3)), , drop = T]
```

```r
# Update
df[which(df$a == 3 | df$b == 3), 2] <- 3
```

dplyr

dplyr package from tidyverse is a high-performance package to deal with data frame.

```
# tidyverse is a bundle of packages.
# I usually load them all with library(tidyverse, instead of library(dplyr) individually.
library(tidyverse)
# Loading tidyverse: ggplot2
# Loading tidyverse: tibble
# Loading tidyverse: tidyr
# Loading tidyverse: readr
# Loading tidyverse: purrr
# Loading tidyverse: dplyr

# Note:
# filter(): dplyr, stats
# lag():    dplyr, stats
# Use dplyr::lag and dplyr::filter when it doesn't work.
```

How dplyr works

`dplyr` provides functions in "verbs", which is functions that does one thing only. We will learn to use the following.

- Key
  - *select: return a subset of the columns of a data frame*
  - *filter: extract a subset of rows based on logical conditions*
  - *arrange: reorder rows*
  - *rename: rename variables*
  - *mutate: add new variables/columns or transform existing variables*
- Group
  - *group_by / rowwise / ungroup: stratify the data*
  - *summarise / summarize: generate summary statistics of different variables in the data frame, possibly within strata*
  - *do: process data within the strata*

- Combine
  - *left_join / right_join / anti_join / full_join*
  - *bind_rows / bind_cols*
- Helpers
  - *%>%: the “pipe” operator is used to connect multiple verb actions together into a pipeline*
  - *ifelse / case_when*
  - *lag*
  - *n*

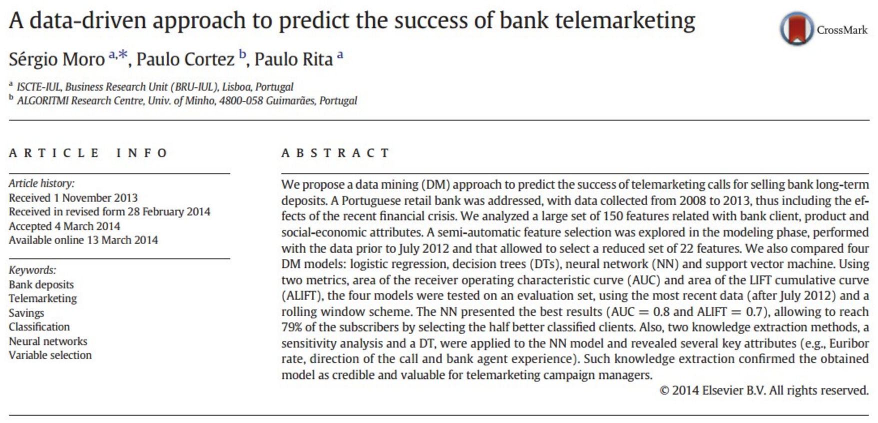
Sample dataset

A data-driven approach to predict the success of bank telemarketing

Author: Sérgio Moroa; Paulo Cortezb; Paulo Ritaa

http://dx.doi.org/10.1016/j.dss.2014.03.001

I chose this data set of a Portuguese retail bank clients profile



A data-driven approach to predict the success of bank telemarketing

CrossMark

Sérgio Moro [a,*], Paulo Cortez [b], Paulo Rita [a]

[a] *ISCTE-IUL, Business Research Unit (BRU-IUL), Lisboa, Portugal*
[b] *ALGORITMI Research Centre, Univ. of Minho, 4800-058 Guimarães, Portugal*

A R T I C L E   I N F O

Article history:
Received 1 November 2013
Received in revised form 28 February 2014
Accepted 4 March 2014
Available online 13 March 2014

Keywords:
Bank deposits
Telemarketing
Savings
Classification
Neural networks
Variable selection

A B S T R A C T

We propose a data mining (DM) approach to predict the success of telemarketing calls for selling bank long-term deposits. A Portuguese retail bank was addressed, with data collected from 2008 to 2013, thus including the effects of the recent financial crisis. We analyzed a large set of 150 features related with bank client, product and social-economic attributes. A semi-automatic feature selection was explored in the modeling phase, performed with the data prior to July 2012 and that allowed to select a reduced set of 22 features. We also compared four DM models: logistic regression, decision trees (DTs), neural network (NN) and support vector machine. Using two metrics, area of the receiver operating characteristic curve (AUC) and area of the LIFT cumulative curve (ALIFT), the four models were tested on an evaluation set, using the most recent data (after July 2012) and a rolling window scheme. The NN presented the best results (AUC = 0.8 and ALIFT = 0.7), allowing to reach 79% of the subscribers by selecting the half better classified clients. Also, two knowledge extraction methods, a sensitivity analysis and a DT, were applied to the NN model and revealed several key attributes (e.g., Euribor rate, direction of the call and bank agent experience). Such knowledge extraction confirmed the obtained model as credible and valuable for telemarketing campaign managers.

Sample dataset columns

variable = column = field

Personal profile

- 1 - age (numeric)
- 2 - job : type of job (categorical: "admin.","unknown","unemployed","management","housemaid","entrepreneur" "blue-collar","self-employed","retired","technician","services")
- 3 - marital : marital status (categorical: "married","divorced","single"; note: "divorced" means divorced or widowed)
- 4 - education (categorical: "unknown","secondary","primary","tertiary")
- 5 - default: has credit in default? (binary: "yes","no")
- 6 - balance: average yearly balance, in euros (numeric)
- 7 - housing: has housing loan? (binary: "yes","no")

- 8 - loan: has personal loan? (binary: “yes”,“no”)

Related with the last contact of the current campaign:

- 9 - contact: contact communication type (categorical: “unknown”,“telephone”,“cellular”)
- 10 - day: last contact day of the month (numeric)
- 11 - month: last contact month of year (categorical: “jan”, “feb”, “mar”, …, “nov”, “dec”)
- 12 - duration: last contact duration, in seconds (numeric)

Other attributes:

- 13 - campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
- 14 - pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric, -1 means client was not

previously contacted)

- 15 - previous: number of contacts performed before this campaign and for this client (numeric)
- 16 - poutcome: outcome of the previous marketing campaign (categorical: "unknown","other","failure","success")

Output variable (desired target):

- 17 - y - has the client subscribed a term deposit? (binary: "yes","no")

Read data

Use RStudio's File -> Import Dataset, you may choose either "From Text (base)" or "From Text (readr)". Either way loads the data.

`base` comes with R. `readr` is a package from tidyverse that provides more options and functionality. Copy the generated code to your script file.

I place it at https://goo.gl/fFQAAm (for Download), https://goo.gl/PBQnBt (for direct use).

You may download it and save it to local.

```
# Use base
bank <- read.csv("example/data-bank/bank.csv", sep=";") # or,
bank <- read.csv("https://goo.gl/PBQnBt", sep = ";")

# use readr
library(readr)
bank <- read_delim("example/data-bank/bank.csv",
                   ";", escape_double = FALSE, trim_ws = TRUE)
## Parsed with column specification:
## cols(
##   age = col_integer(),
##   job = col_character(),
##   marital = col_character(),
##   education = col_character(),
##   default = col_character(),
##   balance = col_integer(),
##   housing = col_character(),
##   loan = col_character(),
##   contact = col_character(),
##   day = col_integer(),
##   month = col_character(),
##   duration = col_integer(),
##   campaign = col_integer(),
##   pdays = col_integer(),
##   previous = col_integer(),
##   poutcome = col_character(),
##   y = col_character()
## )
```

```
View(bank)
```

`select`

`select(df, ...)`, … can be

- variable name
- numeric to indicate nth column (– means exclude)
- a range
- a function

`select` - Examples

```
subset <- select(bank, marital)
subset <- select(bank, 1)
subset <- select(bank, -1)
subset <- select(bank, -job)
subset <- select(bank, -(job:education))
subset <- select(bank, starts_with("p"))
subset <- select(bank, ends_with("p"))
subset <- select(bank, contains("p"))
```

`select` as a re-arrangement of columns.

```
job_first <- select(bank, job, everything())
bank
```

age	job	marital	education	default	balance	housing	lo
<int>	<chr>	<chr>	<chr>	<chr>	<int>	<chr>	<
30	unemployed	married	primary	no	1787	no	n
33	services	married	secondary	no	4789	yes	ye
35	management	single	tertiary	no	1350	yes	n
30	management	married	tertiary	no	1476	yes	ye
59	blue-collar	married	secondary	no	0	yes	n
35	management	single	tertiary	no	747	no	n
36	self-employed	married	tertiary	no	307	yes	n
39	technician	married	secondary	no	147	yes	n
41	entrepreneur	married	tertiary	no	221	yes	n
43	services	married	primary	no	-88	yes	ye

1-10 of 4,521 rows | 1-10 of 17 columns Previous 1 2 3 4 5 6 ... 453 Next

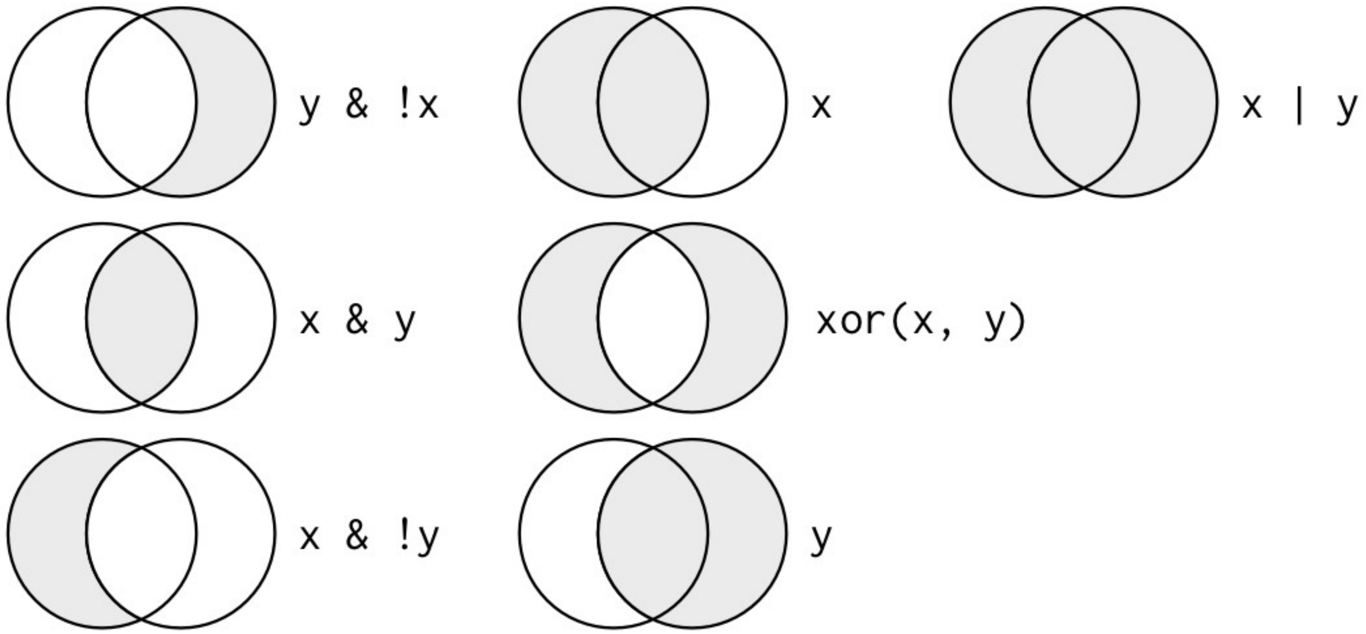
filter

```
colnames(bank)
##  [1] "age"       "job"       "marital"   "education" "default"
##  [6] "balance"   "housing"   "loan"      "contact"   "day"
## [11] "month"     "duration"  "campaign"  "pdays"     "previous"
## [16] "poutcome"  "y"

young <- dplyr::filter(bank, age < 40)
another_young <- dplyr::filter(bank, age < 20 & marital == "married")
just_young <- dplyr::filter(bank, age < 20 & marital == "single")

young2 <- dplyr::filter(bank, age >= 20 & age < 30)
another_young2 <- dplyr::filter(bank, age >= 20 & age < 30 & marital == "married")
just_young2 <- dplyr::filter(bank, age >= 20 & age < 30 & marital == "single")
```

filter - logic operators



filter - string operations

```r
# %in% to match multiple
second_upper <- dplyr::filter(bank, education %in% c("tertiary", "secondary"))

# filter out NA value.
no_na <- dplyr::filter(bank, is.na(balance) | balance > 0)
```

Exercise

- How many bank client have a loan while doesn’t have a housing?
- How many bank client have a job between 20 to 40?

rename

```r
# rename(new name = old)
# Use tick to quote special strings.
df <- rename(bank, young_age = age)
df <- rename(bank, `Age in Bank` = age)
```

arrange

```r
# arrange is sort
arrange(bank, job)
```

```r
arrange(bank, default, job)
```

```
# descending for day
arrange(bank, desc(day))
arrange(bank, desc(as.Date(day, format="%d", origin = Sys.Date())))
```

NB: Missing values are always sorted at the end.

Exercise

- How could you use arrange() to sort all missing values to the start? (Hint: use is.na()).

  arrange(bank, !is.na(a), a)

- Find the longest duration?

- Find the eldest?

mutate

```r
# Replace existing
# ifelse is to check condition.
df1 <- mutate(bank, y = ifelse(y == "yes", T, F))

# Add a new column.
df2 <- mutate(bank, duration_diff = duration - mean(duration, na.rm = TRUE))

# case_when is a function to deal multiple choices.
mutate(bank, age_group = case_when(
  age < 20 ~ "youth",
  age < 40 ~ "middle-age",
  age < 50 ~ "senior",
  TRUE ~ "happy"
))
```

```
firstup <- function(x) {
  substr(x, 1, 1) <- toupper(substr(x, 1, 1))
  x
}

# month.abb is a built-in array of month names.
df3 <- mutate(bank, month_name = factor(firstup(as.character(month)), levels = month.abb))

# transmute would remove all other columns after mutation, only keeping the new variable.
df5 <- transmute(bank,
                 duration_trend = duration - mean(duration, na.rm = TRUE),
                 balance_trend = balance - mean(balance, na.rm = TRUE))
```

What you can do with `mutate`

- +, -, *, /: ordinary arithmetic operator
- %/% (integer division) and %% (remainder), where x == y * (x %/% y) + (x %% y)
- x / sum(x) amd y - mean(y): computes the difference from the mean.
- log2(), log(), log10():
- lead(), lag(): compute running differences (e.g. x - lag(x)) or find when values change (x != lag(x))
- rolling sum, prod, min, max: cumsum(), cumprod(), cummin(), cummax(); and dplyr provides cummean()
- row_number()/min_rank()/ntile(,n)

```
y <- c(1, 2, 2, NA, 3, 4)
row_number(y)
## [1]  1  2  3 NA  4  5
min_rank(y)
## [1]  1  2  2 NA  4  5
```

```
ntile(y, 2)
## [1]  1  1  1 NA  2  2
```

%>%

We may write such code.

```
df <- select(df, x)
df <- mutate(df, a = 1)
df <- rename(df, a = b)
df <- arrange(df, x)

# This is effectively,
arrange(rename(mutate(select(df, x), a = 1), a = b), x)

third(second(first(x)))
```

How about this?

```
df %>% select %>% mutate %>% rename %>% arrange
```

%>% Benefits

`%>%` operator allows you to transform the flow from nesting to left-to-right fashion, i.e.

```
first(x) %>% second() %>% third()

x %>% first() %>% second() %>% third() # this could also do.

x %>% first(.) %>% second(.) %>% third(.) # . represents the input
```

What's the output of below?

```
c(1, 3, 7, 9) %>% {
  print(.)
  mean(.)
} %>% { . * 3 } %>% {
  print(.)
  sample(round(., 0))
}
## [1] 1 3 7 9
## [1] 15
##  [1]  5  8 13 14  4 11 15  2  9  7  1 12  6 10  3
```

Work with Pipe

%>% … %>%

```
# Feed the data for multiple processing
{
  v <- .
  cn <- colnames(v)

  v <- select(v, u, z)
  colnames(v) <- cn[1:3]
  v
}

# How to return multiple value

%>%
  assign("data_name", data, envir = parent.env(environment()) )
} %>% {
  select(., z < 0.4)
}

# or, we use list
%>% {
  list(a, b)
}  %>% {
  v <- .
  v$a
  v$b
}
```

Code pattern with Pipe

```
df %>%
... %>%
... %>%
... %>%
{
  v <- .
  ggplot(data = v) +
    # full data is used here
    geom_line(data = v) +
    # partial data needs to be hightlighted.
    geom_line(data = filter(., some condition), color = "red")
}
```

Use of Caution

Pros:

- We don't need to keep intermediate result, sames memory and also variable names.

Cons:

- Difficult to debug, to find something in the middle of the chain.
- Use `{ print(.); filter(., ...) }` to print intermediate resuls.
- Separate the long pipes into shorter pipes, adding more intermediate variables.
- Your pipes are longer than (say) ten steps. In that case, create intermediate objects with meaningful names. That will make debugging easier, because you can more easily check the intermediate results, and it

makes it easier to understand your code, because the variable names can help communicate intent.

- You have multiple inputs or outputs. If two or more objects being combined together, don’t use the pipe.

- Pipes are fundamentally linear and expressing complex relationships with them will typically yield confusing code.

Environment

Environment is where your data resides. Use `local()` to isolate.

```
# local stores the data wihtin the boundary of {}
x <- 3
local({
  print(x)
  x <- 1
  print(x)
})
## [1] 3
## [1] 1
print(x)
## [1] 3
```

```
# local stores the nearest environment
x <- 3
{
  print(x)
  x <- 1
  print(x)
}
## [1] 3
## [1] 1
x
## [1] 1
```

Environment

Use `assign()` to space-jump.

```
# assign data to global environment
x <- 1
pass_out_global <- function() {
  assign("x", 3, envir = .GlobalEnv)
}

# assign data to just one level up
pass_out <- function() {
  assign("x", 2, envir = parent.env(environment()))
}
```

```
x <- 1
pass_out()
x
## [1] 2

# assign data to pass it out of function
extra_layer <- function() {
  pass_out()
}

x <- 1
extra_layer()
x
## [1] 2
```

```
extra_layer_g <- function() {
  pass_out_global()
}

x <- 1
extra_layer_g()
x
## [1] 3
```

Summary

- We learned the key “verbs” from dplyr. Let’s pick up the rest next week.