# FE8828 Programming Web Applications in Finance

Week 2: 5. R Programming/2 6. R Shiny/2: Building a Web App 7. dplyr/1: Data Manipulation

Dr. Yang Ye yy@runchee.com

Nanyang Business School

Sep 24, 2020

Section 1

## Lecture 5: R Programming/2

## Object - S3 Object System in R

```
# 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

```r
set.seed(1234)
# Generate a vector of option objects
opts <- sapply(1:1000,
                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) {
  obj <- GBSOption(o@type, o@underlying, o@strike,
             Time = 1, r = 0.01, b = 0, sigma = 0.3)
  obj@price
})
cat(paste0("Time used: ", as.numeric(Sys.time() - start)))
## Time used: 0.223999977111816
```

## Objects or Data Frame?

We can re-write above example using a data frame. We can notice a few differences but largely the same.

Note: `tibble` is to create data frame in Tidyverse.

```r
set.seed(1234)
# Generate a vector of options
df_opts <- tibble(type = sample(c("c", "p"), 1000, replace = TRUE),
                  strike = round(runif(1000) * 100, 0),
                  underlying = round(runif(1000) * 100, 0))

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

start <- Sys.time()
# GBSOption also returns an object. We just need its price attribute.
res2 <- by(df_opts, 1:nrow(df_opts), function(r) {
  obj <- GBSOption(r$type, r$underlying, r$strike,
            Time = 1, r = 0.01, b = 0, sigma = 0.3)
  obj@price
}, simplify = TRUE)
cat(paste0("Time used: ", as.numeric(Sys.time() - start)))
## Time used: 0.395026922225952
```

## Objects or Data Frame? - My take

- Arising from data science, most calculations is around data frame(s). Does it make object obsolete?
- No. Please consider object to be used as “data in transition” which individual attention is needed, frequent internal status change, upgrade, transform, etc.
- Data frame is to process “data in finish”. We will apply group-based action and study the data inside to gain insights.
- Consider both data frame-oriented programming and object-oriented programming.
- Data frame requires us to know what’s inside, the name and data types. Object has a definition that helps us to store data in one format.
- Besides, object can be used to organize functions, etc.

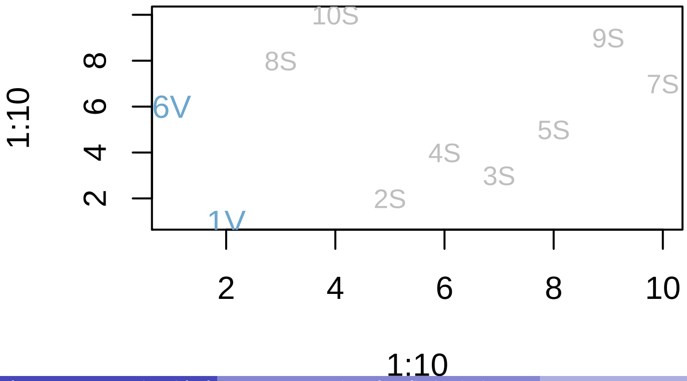
## Read/Write data

```r
# set working directory
setwd("C:/TEMP")

# Save this_is_var1 to a file
this_is_var1 <- 112131
saveRDS(this_is_var1, file = "C:/TEMP/DATA/data.Rds")
# Load data from a file into a new variable `new_var`
new_var <- readRDS(file = "C:/TEMP/DATA/data.Rds")
print(new_var) # gives 112131
```

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

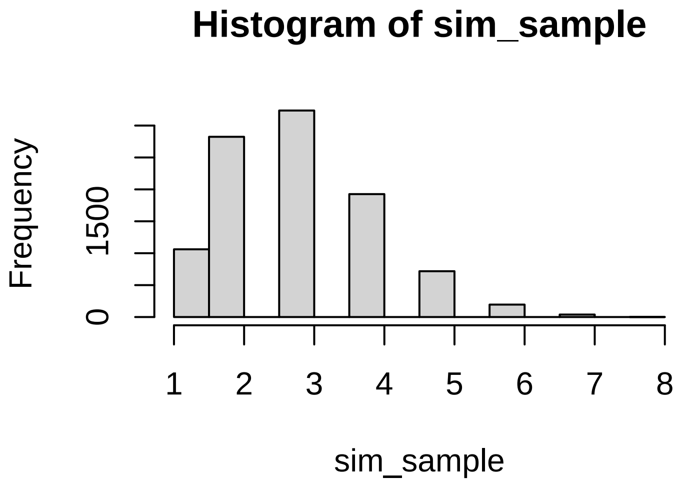
## Question: Fastest Fish Problem

We have ten fishes releases in a very long lane at fixed time interval but random order. They swim at different speed. The fast fish would eat the slow fish. On average, How many fishes would survive?



## Histogram of Number of Survived Fishes

- Result from simulation: 2.9162
- Result from analytics: 2.9289683



## Assignment: How far to make a choice?

- Secretary Problem https://en.wikipedia.org/wiki/Secretary_problem
  - If we have 100 secretaries ranked from best to worst, coming to interview at random order.
  - Our selection strategy, use a small group to establish our selection criteria, for the subsequent ones, we pick the first that's better than our selection criteria.
  - What's the size of the split so that the probabiliy of our picked is the best is maximized? What's the probabiliy that we can pick the best?
- Hint: use simulation method
  - Step 1: create function `make_choice <- function(N, split_number)`
    1. Generate a list `input_list` of `N` long with integer 1 to N at random position
    2. Split the list `input_list` into two: evaluation group and selection group.
    3. Remember the best number from evaluaton group and match the first number in selection group, >= than best. Return it.
    4. Run this function for a few (hundred) times and find the probability of getting `N`.
  - Step 2: create function `find_optimal()`, calls `make_choice` for each of the split number from `1` to `N/2`. So we can find the optimal value for the split for the `N`.
  - Step 3: Find the solution for `N = 3`, then, `N = 10`, then move on to `N = 100`.

## Take-home

- How many data types have you remembered? Have you used?
  - Use more vector, list in ordinary programming.
  - Use matrix, data frame for data programming.
  - Use as.Date()/as.character() to convert date. Use lubridate/bizdays packages
- Write anonymous functions.
- Use apply()/purrr:map() function.
- Try to use object when appropriate to organize code (reduce copy and duplicates).

## Section 2

## Lecture 6: R Shiny/2: Building a Web App

## Start with minimalism

In Shiny/1, we worked on `ui`, now we move on to `server`. Recall, this was our mimimal Shiny.

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

The default `server` function created by R Studio has two inputs only, add `session` as 3rd input.

## Think around Input and Outputs

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

## Input controls

Input controls follow such function signature except for input-specific parameters.

Note: UI elements (User-Interface) for user interactin (edit or display content) is called “control”, or widgets.

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

More types of controls. You can now use layout containers to place them in the Shiny App.

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

Reference: https://shiny.rstudio.com/reference/shiny/1.5.0/

## Output controls

Output controls 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")`

For plotOutput, I suggest to set width and height to fixed size, though they are optional input to the function. For other kinds of outputs, only `outputId` is good enough (usually).

## Server

`server` function is to add dynamic action.

```r
server <- function(input, output, session) {
  # Enable either one from below.
  output$hist <- renderPlot({ hist(rnorm(100)) })

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

## shinyApp = UI + Server

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

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 - 1

Reactivity is enabled by placing input `inputXXX` inside `renderXXX` function. (shiny-21.R)

This is an automatic way to add the reactive linkage.

```
library(shiny)

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

# input$num    --> output$p1
# input$lambda -/

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))
  })
```

## Reactivity - 2

- Button represents a manual trigger of the action.
- 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`? (shiny-22.R)

```
library(shiny)

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

# input$go -> output$p1
# (input$num)


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

      # Detailed code
      # To make code in good clarity, I re-write above one line as below.
```

## Reactivity - 3

We can add a reactiveValue with `eventReactive`. (shiny-23.R) This would make reactive link for “1 - N”.

```
library(shiny)

ui <- fluidPage(
  numericInput("num", "Num", 100),
  actionButton("go", "Go"),
  plotOutput("p1"),
  plotOutput("p2")
)

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

  # input$go <-> data() --> output$p1
  #                     \-> output$p2

  # Variable data becomes a reactive variable.
  # What changes to it will trigger the output.
  output$p1 <- renderPlot({ hist(data()) })

  output$p2 <- renderPlot({ hist(-data()) })
```

## Reactivity - 4

We can add a reactiveValue with `reactiveValue`. (shiny-24.R) This would make reactive link for “N - 1”, or “N - N”.

```
library(shiny)

ui <- fluidPage(
  numericInput("num1", "Num", 100),
  numericInput("num2", "Num", 100),
  h4("Sum"),
  textOutput("t1")
)

server <- function(input, output, session) {
  sum_v <- reactiveVal(0)

  # Instead of anonymous function, we use a named function
  calc_sum <- function() {
      sum_new <- isolate(input$num1) + isolate(input$num2)
      sum_v(sum_new)
  }

  # input$num1 --> sum_v -> output$t1
  # input$num2 -/

  observeEvent(input$num1, {
    sum_new <- isolate(input$num1) + isolate(input$num2)
    sum_v(sum_new)
  })
  observeEvent(input$num2, {
```

## 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-25.R)

```
library(conflicted)
library(shiny)
library(DT)
conflict_prefer("dataTableOutput", "DT")
conflict_prefer("renderDataTable", "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"),
              selectInput("plot_color", "Highlight", choices = colors()),
              plotOutput("p5")))
)

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

server <- function(input, output, session) {
  output$t1 <- renderTable(iris[1:10,], striped = TRUE, hover = TRUE)
  output$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)
```

## Debug Shiny

- Use print to check certain code has been run.
- Clear environment to run Shiny in R Studio, so you can check whether your App has all the data it can load.
- Use stop point

## Shiny: Take-home

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

## Shiny Assignment - 1

1. Create a Bond Schedule
   - Inputs: start date, tenor, coupon rate, coupon frequency (Annual, Q, Semi-Annual), and yield to maturity.
   - Output:
     - A table of coupon schedule (date, ignoring public holidays) and payment amount
     - A plot of payment vs schedule

NPV

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

For a Bond with fixed coupon

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

## Shiny Assignment - 2

2. Create a data downloader

- Register at https://www.alphavantage.co/support/#api-key
- Install R package `alphavantager`
- Write an App to let user input a US stock ticker, save it in RDS format and plot it.
- Help:
  - Sample code to download data.
  - https://www.alphavantage.co/documentation/
  - https://cran.r-project.org/web/packages/alphavantager/alphavantager.pdf

## Shiny Assignment - 2 - alphavantager sample

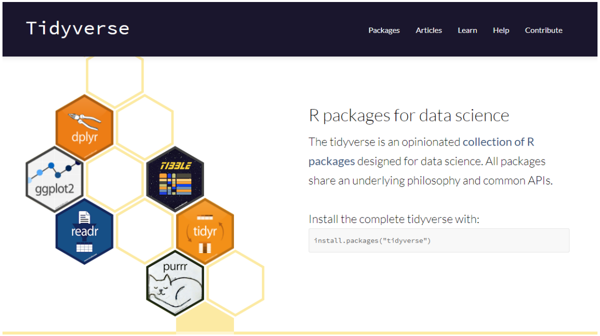
```
library(alphavantager)
av_api_key("Your Key")

# To speed up download, we use compact to download recent 100 days.
# outputsize is default to "compact"
df_res <- av_get("MSFT",av_fun = "TIME_SERIES_DAILY_ADJUSTED",outputsize="compact")

# Below code can return NA if bad code is passed.
df_res <- tryCatch({
  df_res <- av_get("BadStockCode", av_fun = "TIME_SERIES_DAILY_ADJUSTED")
  df_res
  }, error = function(e) {
    NA
  })
is.na(df_res) # TRUE

# plots
plot(df_res$timestamp, df_res$adjusted_close)
lines(df_res$timestamp, df_res$adjusted_close)
```

## Section 3

## Lecture 7: dplyr/1: Data Manipulation

## Tidyverse

install.packages(“tidyverse”)



R packages for data science

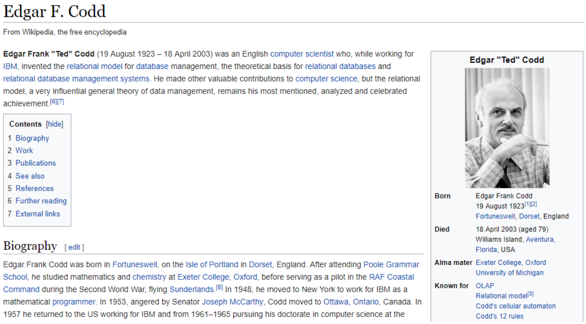
The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying philosophy and common APIs.

Install the complete tidyverse with:

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

# SQL

- It was invented by Edgar Codd
- It first appeared in 1974, which is 46 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 ]

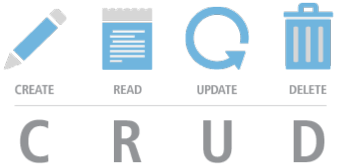
Edgar Frank Codd was born in Fortuneswell, on the Isle of Portland in Dorset, England. After attending Poole Grammar School, he studied mathematics and chemistry at Exeter College, Oxford, before serving as a pilot in the RAF Coastal Command during the Second World War, flying Sunderlands.[8] In 1948, 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<br>19 August 1923[1][2]<br>Fortuneswell, Dorset, England |
| **Died** | 18 April 2003 (aged 79)<br>Williams Island, Aventura, Florida, USA |
| **Alma mater** | Exeter College, Oxford<br>University of Michigan |
| **Known for** | OLAP<br>Relational model[3]<br>Codd's cellular automaton<br>Codd's 12 rules |

## CRUD: Create | Read | Update | Delete

- The combination of these operations can create complete programs.
- Data engineering was born around 70s with SQL.
- Nowadays, `dplyr` inherites the thoughts to do data manipulation with **verbs** not SQL.



## Data frame does CRUD

```r
df <- tibble(a = 1:10, b = 10:1)

# Select (aka Filter)
df[which(df$a == 3 | df$b == 3), , drop = T]
df[match(3, df$a), , drop = T]
df[, match("b", colnames(df)), drop = T]

# Insert
rbind(df, df)

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

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

## dplyr

dplyr package from tidyverse is a high-performance package to manipulate data in data frame.

```
# tidyverse is a bundle of packages.
# I usually load them all with library(tidyverse, instead of library(dplyr) individually.
library(conflicted) # help to resolve name conflicts
library(tidyverse)
# -- Attaching packages --------------------------------------- tidyverse 1.2.1 --
# v ggplot2 3.2.1     v purrr   0.3.2
# v tibble  2.1.3     v dplyr   0.8.3
# v tidyr   0.8.3     v stringr 1.4.0
# v readr   1.3.1     v forcats 0.4.0
# -- Conflicts ------------------------------------------ tidyverse_conflicts() --
# x dplyr::filter() masks stats::filter()
# x dplyr::lag()    masks stats::lag()
# There are other filter() or lag() functions in packages.
# Following code prefer the ones from dplyr pacakge.
conflict_prefer("filter", "dplyr")
## [conflicted] Removing existing preference
## [conflicted] Will prefer dplyr::filter over any other package
conflict_prefer("lag", "dplyr")
## [conflicted] Removing existing preference
## [conflicted] Will prefer dplyr::lag over any other package
# Alternative, use dplyr::lag and dplyr::filter with their package names
```

## How dplyr works

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

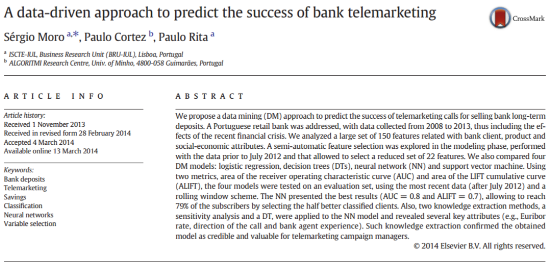
- Key
  - select: return a subset of the columns of a data frame
  - filter: extract a subset of rows based on logical conditions
  - arrange: reorder rows
  - rename: rename variables
  - mutate: add new variables/columns or transform existing variables
- Group
  - group_by / rowwise / ungroup: stratify 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/distinct
  - n

# Sample dataset

```
A data-driven approach to predict the success of 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.

- Real data collected from a Portuguese retailbank, from May 2008 to June 2013, in a total of 52,944 phone contacts.



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*

ARTICLE INFO

*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

ABSTRACT

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 (also called variable, field or feature)

- Personal profile
  1. age (numeric)
  2. job : type of job (categorical: "admin.","unknown","unemployed","management","housemaid","entrepreneur","student" "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)

## Sample dataset columns - 2

- 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

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

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.

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)
##
## -- Column specification --------------------------------------------------------
## cols(
##   age = col_double(),
##   job = col_character(),
##   marital = col_character(),
##   education = col_character(),
##   default = col_character(),
##   balance = col_double(),
##   housing = col_character(),
##   loan = col_character(),
##   contact = col_character(),
##   day = col_double(),
##   month = col_character(),
##   duration = col_double(),
```

## `select`

`select(df, ...)`, ... can be

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

## select - Examples

```r
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"))
```

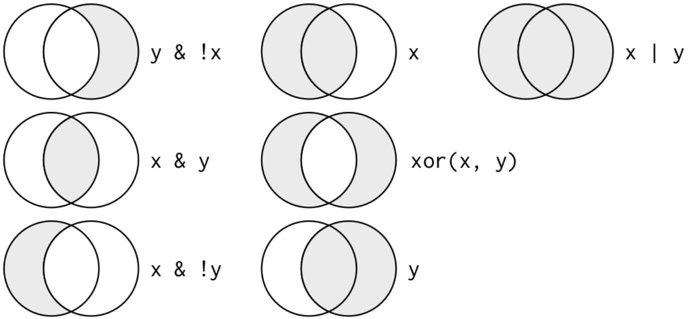
## `select` as a re-arrangement of columns.

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

# filter

```r
colnames(bank)
##  [1] "age"       "job"       "marital"   "education" "default"   "balance"
##  [7] "housing"   "loan"      "contact"   "day"       "month"     "duration"
## [13] "campaign"  "pdays"     "previous"  "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)
```

### Question

- 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)
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.

### Question

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

```r
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.
df2_age_group <- mutate(bank, age_group = case_when(
  age < 20 ~ "youth",
  age < 40 ~ "middle-age",
  age < 50 ~ "senior",
  TRUE ~ "happy"
))

df2_age_group_res <-
  group_by(df2_age_group, age_group) %>%
  summarise(mean_age = mean(age)) %>%
  transmute(mean_age_diff = mean_age - lag(mean_age))
## `summarise()` ungrouping output (override with `.groups` argument)
```

## mutate - 2

```r
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): compute the proportion of all things
- 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)
- cumulative sum, prod, min, max: cumsum(), cumprod(), cummin(), cummax(); and dplyr provides cummean()
- rolling sum:
- 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
```

## Take-Home

- We learned the key “verbs” from dplyr. Review them and try to remember each.
- select, filter, rename, arrange, mutate, etc.
- Let’s pick up the rest next week.

## Special Topic: Environment

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

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

```
# 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
```

## Environment - Isolation from outside

```r
get_sum <- function(i) {
  v <- 0
  for (i in 1:10) {
    v <- v + i
  }
  v
}

get_sum(10)
## [1] 55

# Error with line below: object 'v' not found
# v
```

## Environment

Use assign() to do cross-environment-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(env) {
  print(env)
  assign("x", 2, envir = env)
}
```

# Environment

## Usage of `pass_out()/pass_out_global()`

```r
x <- 1
pass_out(environment())
## <environment: R_GlobalEnv>
x
## [1] 2

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

x <- 1
extra_layer(env = environment())
## <environment: R_GlobalEnv>
x
## [1] 2

extra_layer_g <- function() {
  pass_out_global()
}

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

## Special Topic: Pipe %>%

We may write such code.

```r
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?

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

## %>% Benefits

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

```
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]  4 15  1  8  2  9 10 11 12  6  7 14 13  3  5
```

## Work with Pipe - Techniques

Feed the data for a bit complicatd processing

```
%>% {
  v <- .
  cn <- colnames(v)

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

## Work with Pipe - Techniques

How to return multiple value

```
%>% {
  assign("new_data", filter(., group == "1"),
         envir = parent.env(environment()) )
  filter(., group == "2")
} %>% {
  select(., z < 0.4) # on group 2
  select(new_data, z > 0.4) # on group 1
}

# or, we use list
%>% {
  a <- filter(., group == "1")
  b <- filter(., group == "2")
  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 for Pipe (%>%)

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.
- Pipes are fundamentally linear and expressing complex relationships with them will typically yield confusing code.
- Separate the long pipes into shorter pipes, adding more intermediate variables. Because you can more easily check the intermediate results, and it makes it easier to understand your code.