# Introduction to Statistical Programming

Timothy Wong[1]
James Gammerman[2]

August 22, 2018

[1] timothy.wong@hotmail.co.uk
[2] jgammerman@gmail.com

# Outline

# R Ecosystem

# What is the R language?

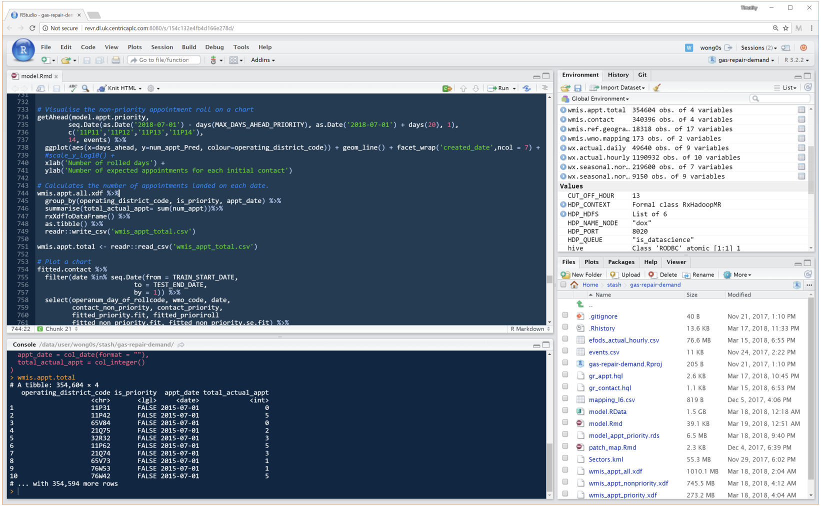
- Offers modern and sophisticated statistical algorithms
- Used by millions of analysts and researchers worldwide
- Has a thriving open-source community
- Enables big data analytics

# Easy to Use

- `PROC REG` = `lm()` or `glm()`
- `PROC SQL` = `%>%`
- `PROC SORT` = `arrange()`
- `PROC MEANS` = `mean()`, `sd()`
- `PROC GPLOT` = `plot()`, `ggplot()`, `autoplot()`

# RStudio Server Pro

- RStudio is your integrated development environment (IDE)



# Packages

- **CRAN** is the Comprehensive R Archive Network.
- User-contributed packages: source code, binaries, documentation.

```r
# Return all installed packages
installed.packages()
# Install a new package and all its dependencies from CRAN
# This will install to the default library location
install.packages("ggplot2", dependencies = TRUE)
# Load an installed package
# (both lines are identical)
library(ggplot2)
library("ggplot2")
```

# Packages

- **CRAN Task View** is a curated list of packages.
- Useful guide to get started with `R`.
- `https://cran.r-project.org/web/views/`

# Variable Assignment

- Assign variables using the <- symbol.
- Do not use reserved words. This will confuse the interpreter.

```
# Assign variables
myVarX <- 5
myVarY <- 20
# Perform multiplication
myVarX * myVarY
# Look at the reserved words
?Reserved
```

# Vectors

- R is a vectorised programming language.
- Vector contains elements of the same data type.

```
myVec1 <- 1:10
# Find out the length of vector
length(myVec1)

## [1] 10

# Reverse the vector
# This does not change the value of myVec1
rev(myVec1)

##  [1] 10  9  8  7  6  5  4  3  2  1

# Create a custom vector of 10, 15, 20, 25, 30
myVec2 <- c(10, 15, 20, 25, 30)
myVec2

## [1] 10 15 20 25 30

# Create a vector of sequential numbers with increment 0.5
myVec3 <- seq(from = -2, to = 2, by = 0.5)
myVec3

## [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0

# Elements of a vector can be named
myVec4 <- c(`New York` = 8.5,
            `London` = 8.6,
            `Moscow` = 11.9)
myVec4

## New York   London   Moscow
##      8.5      8.6     11.9
```

# Subsetting a Vector

- You can subset elements from a vector.

```
# Select the second element of the vector
myVec2[2]

## [1] 15

# Subset a range from the vector
myVec2[2:4]

## [1] 15 20 25

# Subset specified elements
myVec2[c(4,2,3)]

## [1] 25 15 20

# Subset named element of a vector
myVec4["New York"]

## New York
##      8.5
```

# Vectorised Operations

- Operations in R are vectorised.

```
# Arithmetic operations
myVec1 + 10

##  [1] 11 12 13 14 15 16 17 18 19 20

myVec1 - 10

##  [1] -9 -8 -7 -6 -5 -4 -3 -2 -1  0

myVec1 * 2

##  [1]  2  4  6  8 10 12 14 16 18 20

myVec1 / 2

##  [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

myVec1 ^ 2

##  [1]   1   4   9  16  25  36  49  64  81 100

log(myVec1)

##  [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
##  [8] 2.0794415 2.1972246 2.3025851
```

# Character Vectors

- Vector can also contain character objects.

```
# Vector can contain character objects
myVec4 <- c("Bill", "Mark", "Steve", "Jeff", "Larry")
myVec4

## [1] "Bill"  "Mark"  "Steve" "Jeff"  "Larry"

# Constant character vectors in R
LETTERS

##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

letters

##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"

month.name

##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "July"      "August"    "September" "October"
## [11] "November"  "December"

month.abb

##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
## [12] "Dec"
```

# Date and Date/Time Vectors

- `Date` is a data type in base `R`.
- The package `lubridate` extends date/time functionalities.
- Enables easier date/time manipulation.

```
# This is a vector of Date objects
myVec5 <- as.Date(c("2017-07-13",
                    "2017-10-11",
                    "2017-11-21",
                    "2018-01-16",
                    "2018-03-27"))
# Load the lubridate package
# Use the function ymd_hms() to parse date/time with timezone
# Returns a vector of POSIXct (date/time) object
library(lubridate)
myVec6 <- ymd_hms(c("2017-07-13 09:30:00",
                    "2017-10-11 08:00:00",
                    "2017-11-21 10:00:00",
                    "2018-01-16 11:30:00",
                    "2018-03-27 12:00:00"),
                  tz = "Europe/London")
# Date/time manipulation applied to a vector
myVec7 <- myVec6 + hours(1) + minutes(30)
# Compute the day of week - returns a vector of characters
weekdays(myVec7)

## [1] "Thursday"  "Wednesday" "Tuesday"   "Tuesday"   "Tuesday"
```

# Logical Operators

- Apply logical operators on vector objects.

```
# Find all values greater than 5 - returns a vector of logical values
myVec1 > 5

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

# Find all values equal to 7
myVec1 == 7

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE

# Find all values matching 2,4,6 and 8
myVec1 %in% c(2,4,6,8)

##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE

# Find all values between 2 and 7
myVec1 >= 2 & myVec1 <= 7

##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE

# Find all values equal to 7 or equal to 8
myVec1 == 7 | myVec1 == 8

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

# Special Numbers in R

```r
# Pi is constant 3.14159...
pi

## [1] 3.141593

# One divided by zero is infinity
1/0

## [1] Inf

# Negative number divided by zero is negative infinity
-1/0

## [1] -Inf

# Infinity divided by infinity is Not-a-Number (NaN)
Inf/Inf

## [1] NaN

# Not available (NA) plus one is still NA
NA + 1

## [1] NA

# Effects of different special numbers
c(5, 10, 15, NA, 25, 30, NaN, 35, 40, Inf, 50, -Inf, 60) / 5

##  [1]    1    2    3   NA    5    6  NaN    7    8  Inf   10 -Inf   12
```

# List

- List is a generic container for objects of different data types

```
myFavBook <- list(title = "R for Data Science",
                  authors = c("Garrett Grolemund", "Hadley Wickham"),
                  publishDate = as.Date("2016-12-12"),
                  price = 18.17,
                  currency = "USD",
                  edition = 1,
                  isbn = 1491910399)
myFavBook

## $title
## [1] "R for Data Science"
##
## $authors
## [1] "Garrett Grolemund" "Hadley Wickham"
##
## $publishDate
## [1] "2016-12-12"
##
## $price
## [1] 18.17
##
## $currency
## [1] "USD"
##
## $edition
## [1] 1
##
## $isbn
## [1] 1491910399
```

# Subsetting a List

- You can subset a particular member from a list

```
# Select a named element of a list
# Use the dollar sign, followed by name without bracket
myFavBook$title

## [1] "R for Data Science"

# Use double squared brackets with element's name as character
myFavBook[["authors"]]

## [1] "Garrett Grolemund" "Hadley Wickham"

# Select the fourth element in the list
myFavBook[[4]]

## [1] 18.17
```

# Data Frame

- Table with rows (observations) and columns (variables).
- Analogous to an Excel workbook.

```
myFavMovies1 <- data.frame(title = c("Dr. No",
                                     "Goldfinger",
                                     "Diamonds are Forever",
                                     "Moonraker",
                                     "The Living Daylights",
                                     "GoldenEye",
                                     "Casino Royale"),
                           year = c(1962, 1964, 1971, 1979,
                                    1987, 1995, 2006),
                           box = c(59.5, 125, 120, 210.3,
                                   191.2, 355, 599),
                           bondActor = c("Sean Connery",
                                         "Sean Connery",
                                         "Sean Connery",
                                         "Roger Moore",
                                         "Timothy Dalton",
                                         "Pierce Brosnan",
                                         "Daniel Craig"))
```

# Data Frame

```
myFavMovies1

##                     title year   box      bondActor
## 1                  Dr. No 1962  59.5   Sean Connery
## 2              Goldfinger 1964 125.0   Sean Connery
## 3 Diamonds are Forever 1971 120.0   Sean Connery
## 4               Moonraker 1979 210.3    Roger Moore
## 5 The Living Daylights 1987 191.2 Timothy Dalton
## 6               GoldenEye 1995 355.0 Pierce Brosnan
## 7           Casino Royale 2006 599.0   Daniel Craig
```

# Tibble

- Similar to traditional `data frame`.
- `tibble` is the modern standard in R.

```r
library(tibble)
myFavMovies2 <- tibble(title = c("Dr. No",
                                 "Goldfinger",
                                 "Diamonds are Forever",
                                 "Moonraker",
                                 "The Living Daylights",
                                 "GoldenEye",
                                 "Casino Royale"),
                       year = c(1962, 1964, 1971, 1979,
                                1987, 1995, 2006),
                       box = c(59.5, 125, 120, 210.3,
                               191.2, 355, 599),
                       bondActor = c("Sean Connery",
                                     "Sean Connery",
                                     "Sean Connery",
                                     "Roger Moore",
                                     "Timothy Dalton",
                                     "Pierce Brosnan",
                                     "Daniel Craig"))
# Append an extra row at the end of the tibble
# Rewrite the original tibble object
myFavMovies2 <- add_row(myFavMovies2,
        title = "Spectre", year = 2015, box = 880.7,
        bondActor = "Daniel Craig")
```

# Tibble

```
myFavMovies2

## # A tibble: 8 x 4
##   title                    year   box bondActor
##   <chr>                   <dbl> <dbl> <chr>
## 1 Dr. No                   1962  59.5 Sean Connery
## 2 Goldfinger               1964 125   Sean Connery
## 3 Diamonds are Forever     1971 120   Sean Connery
## 4 Moonraker                1979 210.  Roger Moore
## 5 The Living Daylights     1987 191.  Timothy Dalton
## 6 GoldenEye                1995 355   Pierce Brosnan
## 7 Casino Royale            2006 599   Daniel Craig
## 8 Spectre                  2015 881.  Daniel Craig
```

# Subsetting a Tibble

```r
# Get one column by name
myFavMovies2[["title"]]
myFavMovies2$title
# Get a range of columns by position ID
myFavMovies2[, 1:2]
myFavMovies2[1:2]
# Get rows 1 to 3
myFavMovies2[1:3, ]
# Get the "year" variable of row 1-3
myFavMovies2[1:3, "year"]
# Get the "title" and "year" variables of row 4-7
myFavMovies2[4:7, c("title","year")]
```

# Functions

- Functions are vectorised.

```
is.odd <- function(x) {
  # The modulo operator %% returns the remainder
  # If a number divide by 2 gives remainder 1, then it is an odd number
  remainder <- x %% 2
  equalToOne <- remainder == 1
  return(equalToOne)
}
# Execute the function with one input
is.odd(5)

## [1] TRUE

# Execute the function with an integer vector
is.odd(1:10)

##  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE

# Define another function
is.even <- function(x) {
  !is.odd(x)
}
# Return true for even numbers
is.even(1:10)

##  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
```

# If-Else

```r
library(lubridate)
# Find out what day is today
myWeekday <- weekdays(today())
# Check whether today is Saturday or Sunday
if (myWeekday %in% c("Saturday","Sunday")) {
  myGreeting <- "Have a nice weekend"
} else {
  myGreeting <- "Go back to work"
}
# Prints the message
myGreeting
```

# If-Else

- Multiple conditions are allowed.

```r
library(lubridate)
myWeekday <- weekdays(today())
# Checks multiple conditions
if (myWeekday %in% c("Saturday","Sunday")) {
  myGreeting <- "Have a nice weekend"
} else if(myWeekday == "Friday"){
  myGreeting <- "It's Friday!"
} else if(myWeekday == "Monday"){
  myGreeting <- "Oh no..."
} else {
  myGreeting <- "Go back to work"
}
myGreeting
```

# While

- Loops as long the condition stays `TRUE`.

```
myCounter <- 100
while (myCounter > 0) {
  myCounter <- myCounter - 5
  print(myCounter)
}
```

# While

- Skip an iteration using `next`
- Early exit using `break`

```
myCounter <- 100
while (myCounter > 0) {
  myCounter <- myCounter - 5
  if (myCounter > 50) {
    # Skips all iterations if the counter value is greater than 50
    next
  }
  if (myCounter == 10) {
    # Early stop if the counter value matches 10
    break
  }
  print(myCounter)
}
```

```
## [1] 50
## [1] 45
## [1] 40
## [1] 35
## [1] 30
## [1] 25
## [1] 20
## [1] 15
```

# For

```r
myResult <- 0
for (i in 1:100) {
  myResult <- myResult + i ^ 2
}
myResult

## [1] 338350
```

# Apply

- `apply()` takes a 'grid' input: `data frame`, `tibble`, or `matrix`
- Margin 1 = apply over rows
- Margin 2 = apply over columns

```
# The second argument 1 inidicates iterate over rows
myMessages1 <- apply(myFavMovies2, 1, function(row){
  sprintf("%s was released in %s.", row["title"], row["year"])
})
myMessages1

## [1] "Dr. No was released in 1962."
## [2] "Goldfinger was released in 1964."
## [3] "Diamonds are Forever was released in 1971."
## [4] "Moonraker was released in 1979."
## [5] "The Living Daylights was released in 1987."
## [6] "GoldenEye was released in 1995."
## [7] "Casino Royale was released in 2006."
## [8] "Spectre was released in 2015."
```

# lapply

- `lapply()` always returns a list

```
# The second argument 1 inidicates iterate over rows
myMessages2 <- lapply(myFavMovies2$title,
                      function(x){ sprintf("%s is a great movie!", x) })
# Checks the data type of the result
typeof(myMessages2)

## [1] "list"

# Check whether it is a list
is.list(myMessages2)

## [1] TRUE

# Select the 6th element of the list
myMessages2[[6]]

## [1] "GoldenEye is a great movie!"
```

# sapply

- `sapply()` always returns a vector

```
myMessages3 <- sapply(myFavMovies2$title,
                      function(x){ sprintf("%s is a great movie!", x) })
# Check whether it is a list
is.list(myMessages3)

## [1] FALSE

myMessages3

##                                  Dr. No
##                "Dr. No is a great movie!"
##                              Goldfinger
##            "Goldfinger is a great movie!"
##                    Diamonds are Forever
## "Diamonds are Forever is a great movie!"
##                               Moonraker
##             "Moonraker is a great movie!"
##                    The Living Daylights
## "The Living Daylights is a great movie!"
##                               GoldenEye
##             "GoldenEye is a great movie!"
##                           Casino Royale
##         "Casino Royale is a great movie!"
##                                 Spectre
##               "Spectre is a great movie!"
```

# Looping

- Looping can be slow.
- Always try to vectorise your code.

```
# Vectorised operation is fast
system.time({
  myResult <- 1:10000 * 2
})

##    user  system elapsed
##       0       0       0

# Looping is quite slow
system.time({
  myResult <- sapply(1:10000, function(x){ x * 2 })
})

##    user  system elapsed
##       0       0       0

# Appending to vector is much slower
system.time({
  myResult <- c()
  for(i in 1:10000){
    myResult <- c(myResult, i * 2)
  }
})

##    user  system elapsed
##    0.17    0.00    0.17
```

# UK User Communities

- LondonR - LONDONR
- ManchesterR - MANCHESTER R
- CaRdiff - CaRdiff
- SheffieldR
- EdinbR - EdinbR
- CambR -
- NottinghamR
- BirminghamR -
- Oxford RUG -
- Bristol Data Scientists - BRISTOL DATA SCIENTISTS

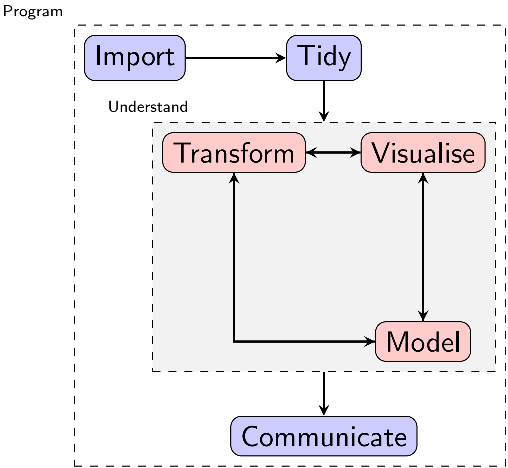
# International User Communities

- International R User Conference (useR!)
- Enterprise Application of the R Language (EARL)
- European R User Meeting (ERUM)

# Data Transformation

# Tidyverse

- A coherent system of packages for data manipulation, exploration and visualisation.
- Typical workflow of a project:



# Tidyverse: Packages

**Import** Reading datasets from various data sources.

- `readr`
- `readxl`
- `haven`
- `httr`
- `rvest`
- `xml2`

**Tidy** Clean up datasets.

- `tibble`
- `tidyr`

**Transform** Aggregate, change variable format and derive new variables.

- `dplyr`
- `forcats`
- `hms`
- `lubridate`
- `stringr`

**Visualise** Creating charts using the Grammar of Graphics.

- `ggplot2`

**Model** Train and test statistical models.

- `broom`
- `modelr`

**Program** Coding in pipeline-style.

- `magrittr`
- `purrr`

# Data Transformation

- Subset the observations by condition - `filter()`
- Reorder the observations - `arrange()`
- Pick variables by name - `select()`
- Compute new variable as a function of existing variables - `mutate()`
- Aggregate many values into one - `summarise()` or `summarize()`
- All of the above functions can be used with `group_by()`
- Syntax: `function(data, actions_to_take)`

# Loading Dataset

```
# Load the package
library(nycflights13)
# View the flights dataset interactively
View(flights)
# Print out the dataset on the console
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

# dplyr Query

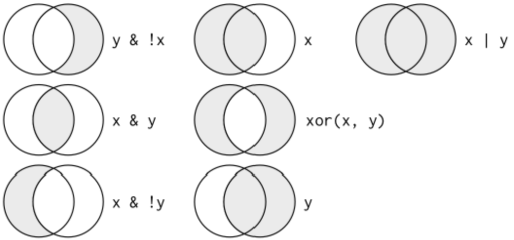
- Finding average arrival delay time for each route and sort it. (i.e. which route are delayed the most?)

```
library(dplyr)
flights %>%
  group_by(tailnum) %>%
  summarise(delay = mean(arr_delay),
            n = n()) %>%
  arrange(desc(delay)) %>%
  filter(n > 100)
```

```
## # A tibble: 1,201 x 3
##    tailnum delay     n
##    <chr>   <dbl> <int>
##  1 N826UA   19.6   120
##  2 N929DL   17.6   108
##  3 N431UA   16.3   104
##  4 N342NW   16.0   111
##  5 N567UA   15.0   105
##  6 N850UA   14.9   130
##  7 N179JB   14.3   311
##  8 N803UA   13.2   129
##  9 N561UA   12.8   103
## 10 N11206   12.7   111
## # ... with 1,191 more rows
```

# Filtering Observations

- Pick a subset of observations based on their values
- Use logical operators: `>`, `>=`, `<`, `<=`, `!=`, `==`
- More complex combinations of logical operators:




- Only returns observations where the condition is `TRUE`. All `FALSE` and `NA` values are excluded.
- Use the function `is.na()` to check if the value is missing.

# Examples - filter()

```
# Select all flights on January 1st
# Assigning them to a new variable jan1
jan1 <- filter(flights, month == 1, day == 1)
jan1
# Select all flights from November or December
filter(flights, month == 11 | month == 12)
# A useful shorthand for this problem is x %in% y
# It selects every row where x is one of the values in y
filter(flights, month %in% c(11, 12))
# Let's select only the flights that weren't delayed
# (on arrival or departure) by more than two hours:
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

# Arranging Observations

- Changes the order of observations in a dataset.
- Sorts the observations by a set of variables in ascending order
  - Use `desc()` to sort in descending order
  - If more than one variable is supplied, each additional variable will break ties in the values of the preceding variable.
  - `NA` values are placed at the end

# Examples - arrange()

```r
# Arrange flights by year, then month, then day
arrange(flights, year, month, day)
# Use desc() to reorder by a column in descending order
arrange(flights, desc(arr_delay))
# Missing values are always sorted at the end:
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
arrange(df,desc(x))
```

# Selecting Variables

- Returns specific variables in the dataset, dropping the others.
- Comes with a number of helper functions you can use within `select()` to pick out variables based on their names:
  1. `starts_with("abc")` matches variable names that begin with `"abc"`.
  2. `ends_with("xyz")` matches variable names that end with `"xyz"`.
  3. `contains("ijk")` matches variable names that contain `"ijk"`.
  4. `num_range("x", 1:3)` matches `x1`, `x2`, and `x3`.
- Can be used to rename variables, but better to use the `rename()` function
- To move several variables to the start of a data frame use `select()` in conjunction with `everything()`

# Examples - select()

```
# Select columns by name
select(flights, year, month, day)
# Select all columns between 'year' and 'day' (inclusive)
select(flights, year:day)
# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
# Rename a variable using rename()
rename(flights, tail_num = talinum)
# Reorder columns using the everything() helper
select(flights, time_hour, air_time, everything())
```

# Compute New Variables

- Create new variables which are functions of existing ones
- New variables are always added at the end of the dataset
- To keep only the newly-computed variables and remove the old ones, use `transmute()`.
- Many functions can be used with `mutate()` to create new variables:

  **Arithmetic operators** `+`, `-`, `*`, `/`, `^`

  **Modular arithmetic** `%/%` for integer division and `%%` for modulo

  **Logs** Very useful for data ranging across multiple orders of magnitude

  **Comparison** `<`, `<=`, `>`, `>=`, `!=`, `==`

  **Ranking** There are several of these, the most common one is `min_rank()`

# Examples - summarise()

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

## # A tibble: 97 x 4
##    dest  count  dist delay
##    <chr> <int> <dbl> <dbl>
##  1 ABQ     254 1826   4.38
##  2 ACK     265  199   4.85
##  3 ALB     439  143  14.4
##  4 ATL   17215  757. 11.3
##  5 AUS    2439 1514.  6.02
##  6 AVL     275  584.  8.00
##  7 BDL     443  116   7.05
##  8 BGR     375  378   8.03
##  9 BHM     297  866. 16.9
## 10 BNA    6333  758. 11.8
## # ... with 87 more rows
```

# Grouped Summaries

- `summarise()` and `summarize()` aggregates a set of values into one.
- Commonly used with `group_by()` to analyse properties of individual groups.
- Examples of summary functions:

  Measures of location: Arithmetic average `mean()` and median `median()`.

  Measures of spread: Standard deviation `sd()` and interquartile range `IQR()`.

  Measures of rank: Minimum value `min()`, maximum value `max()` as well as the quantiles `quantile()`

  Measures of position: `first()` and `last()`
- Use the pipe operator %>% to combining several operations

# Examples - mutate()

```r
# Select several columns only
flights_sml <- select(flights,
                      year:day,
                      ends_with("delay"),
                      distance,
                      air_time)
# Use these the smaller data frame derive new columns
mutate(flights_sml,
        gain = arr_delay - dep_delay,
        speed = distance / air_time * 60)

## # A tibble: 336,776 x 9
##     year month   day dep_delay arr_delay distance air_time  gain speed
##    <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
##  1  2013     1     1         2        11     1400      227     9  370.
##  2  2013     1     1         4        20     1416      227    16  374.
##  3  2013     1     1         2        33     1089      160    31  408.
##  4  2013     1     1        -1       -18     1576      183   -17  517.
##  5  2013     1     1        -6       -25      762      116   -19  394.
##  6  2013     1     1        -4        12      719      150    16  288.
##  7  2013     1     1        -5        19     1065      158    24  404.
##  8  2013     1     1        -3       -14      229       53   -11  259.
##  9  2013     1     1        -3        -8      944      140    -5  405.
## 10  2013     1     1        -2         8      733      138    10  319.
## # ... with 336,766 more rows
```

# Regression Models

# Simple Regression

Univariate linear regression model

$$\hat{y}_i = \beta_0 + \beta_1 x_i$$

- Analogous to a straight line $y = mx + c$
- Can be chained with $M$ dependent variables (Multivariate)

$$\hat{y}_i = \beta_0 + \sum_{m=1}^{M} \beta_m x_{m,i}$$

- Residual term $\epsilon_i = y_i - \hat{y}_i$ assumed to be Gaussian

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

- Also known as ordinary least squared (OLS) regression

# Linear Regression in R

```r
# Build a univariate linear model
# These two lines are equivalent
myModel1 <- lm(mpg ~ wt, mtcars)
myModel1 <- lm(formula = mpg ~ wt, data = mtcars)
# Read the model summary
summary(myModel1)
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.5432 -2.3647 -0.1252  1.4096  6.8727
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
## wt           -5.3445     0.5591  -9.559 1.29e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.046 on 30 degrees of freedom
## Multiple R-squared:  0.7528,Adjusted R-squared:  0.7446
## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

# More Linear Regression Models

**Multivariate linear model** Additional independent variables can be chained using the + symbol. Categorical variables can be encoded as dummy on-the-fly using `factor()`.

```
myModel2 <- lm(mpg ~ wt + hp + qsec + factor(am), mtcars)
```

**Polynomial term** Model can become more flexible when an independent variable is converted into polynomial terms. Use the `poly()` function.

```
myModel3 <- lm(mpg ~ wt + qsec + factor(am) +
                 poly(hp, 3), mtcars)
```

**Interaction term** Two variables can be combined to create synergy effect. The * symbol is used to combine variables together.

```
myModel4 <- lm(mpg ~ wt * hp + qsec + factor(am), mtcars)
```

# Regression Diagnostics

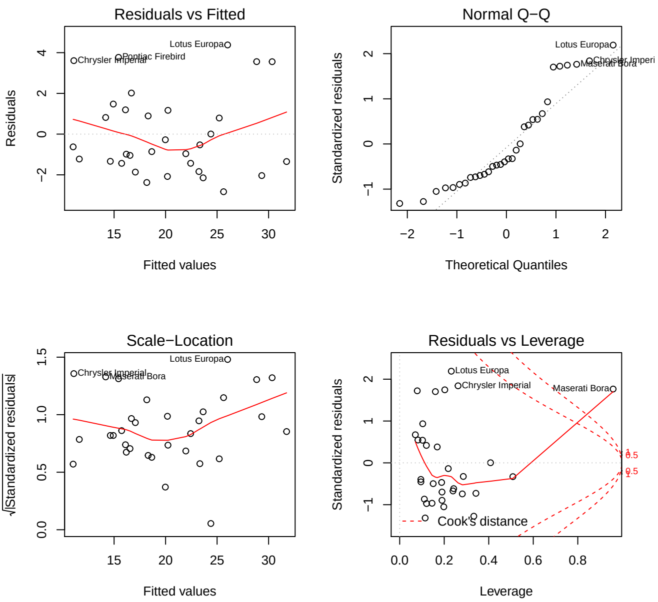
**Residuals vs Fitted** Checks for non-linear relationship. Look for a near horizontal line.

**Normal Quantile-Quantile** It aligns model residuals against a theoretical normal distribution. If the residuals spread along a straight diagonal line on the Q-Q plot, it suggests that the residuals are normally distributed.

**Scale-Location** Checks for homoscedasticity and heteroscedasticity. It is homoscedastic if observations scatter without any observable pattern.

**Residual vs Leverage (Cook's Distance)** Identifies observations having strong influence to the model.

# Diagnostics Plots

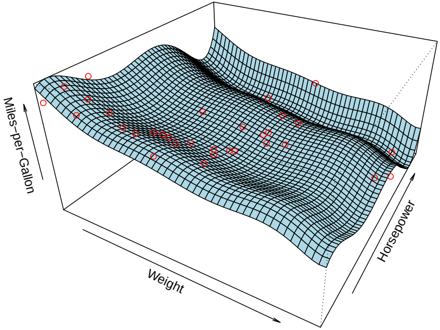


# Overfitting

- Flexible models are prone to overfitting.
- Overfitting makes the model less generalisable.
- Solution
  - Use less flexible methods.
  - Impose regularisation.

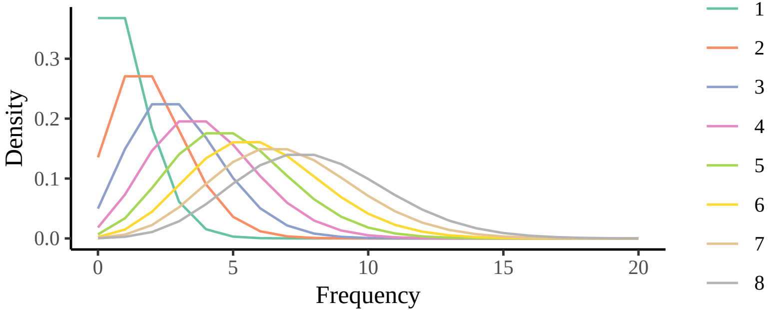
# Overfitting: Visual Explaination

$$\hat{y} = \beta_0 + \sum_{j=1}^{8} \beta_{wt_j} x_{wt}^{j} + \sum_{k=1}^{5} \beta_{hp_k} x_{hp}^{k}$$



# Poisson Distribution

- Count of distinct events are drawn from Poisson distribution.
  - Always positive.
  - In most cases they are integers.
- e.g. Number of people in a room, number of flights delayed per day... etc.



# Testing for Poisson Distribution

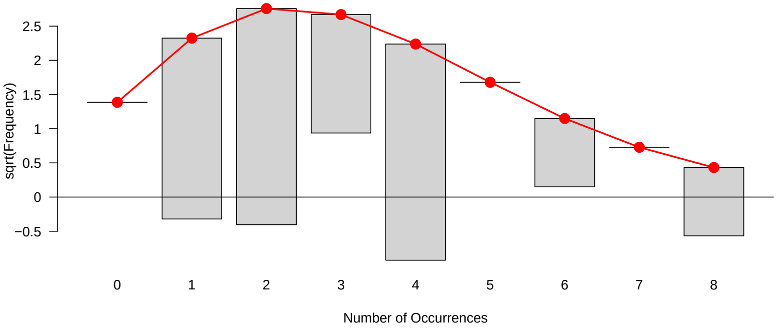
- Chi-square goodness of fit test.
- Fits the data against theoretical Poisson distribution.
- Look for statistical significance.

```
# Performs the Chi-squared goodness-of-fit test.
# It checks whether the variable is drawn from a Poisson distribution.
library(vcd)
gf <- goodfit(mtcars$carb, type= "poisson", method= "ML")
# Checks the statistical p-value of the goodness-of-fit test.
# If p<=0.05 then it is safe to say that the variable is Poisson.
summary(gf)
```

```
##
##   Goodness-of-fit test for poisson distribution
##
##                       X^2 df     P(> X^2)
## Likelihood Ratio 20.53973  4 0.0003906369
```

# Goodness of Fit Plot

```r
# Plots the observed frequency vs theoretical Poisson distribution.
# The hanging bars should fill the space if it is perfectly Poisson.
plot(gf)
```



# Poisson Regression

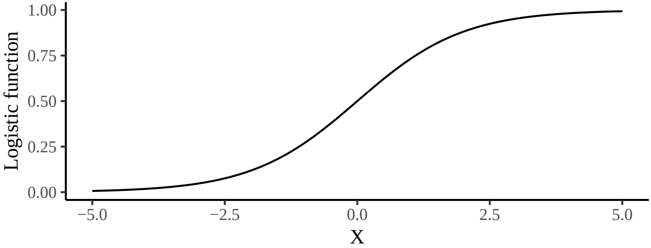
```
# Build a Poisson model to predict the number of carburetors in a car.
myPoissonModel <- glm(carb ~ hp + wt + factor(am),
                      family="poisson",
                      data=mtcars)
# Read the model summary
summary(myPoissonModel)

##
## Call:
## glm(formula = carb ~ hp + wt + factor(am), family = "poisson",
##     data = mtcars)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -0.91420  -0.48423  -0.07246   0.19252   1.26155
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.418081   0.604211  -0.692   0.4890
## hp           0.004316   0.001880   2.296   0.0217 *
## wt           0.179583   0.191352   0.938   0.3480
## factor(am)1  0.393750   0.324978   1.212   0.2257
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 27.043  on 31  degrees of freedom
## Residual deviance: 10.798  on 28  degrees of freedom
## AIC: 108.16
##
## Number of Fisher Scoring iterations: 4
```

# Binomial Distribution

- There are only two possible outcomes $\{Y, \neg Y\}$.
  - Toss a coin (Head or tail)
  - Taking an examination (Pass or fail)
  - Selling a product (Sold or unsold)
- Likelihood of event $Y$ and $\neg Y$ expressed as probablity $P(Y) + P(\neg Y) = 1$.
- Logistic function squeezes real value range $X$ into $(0, 1)$ to express probablity.

$$P(Y) = \frac{1}{1 + e^{-X}}$$



# Logistic Regression

- Equation for logistic regression

$$P(Y) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_M x_M)}}$$

- Coefficients $\beta_1, \beta_2, \beta_3, ..., \beta_M$ can be converted into odds ratios $OR(x_1), OR(x_2), OR(x_3), ..., OR(x_M)$.

$$OR(x_1) = \frac{odds(x_1 + 1)}{odds(x_1)} = \frac{e^{\beta_0 + \beta_1 (x_1 + 1) + \beta_2 x_2 + ... + \beta_M x_M}}{e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_M x_M}} = e^{\beta_1}$$

- $OR(x_1)$ Represents the change in probability when $x_1$ increases by 1 unit.

# Training a Logistic Regression Model

- Build a logistic regression model to predict the dependent variable `am`. (1=manual; 0=auto)

```
myLogisticModel <- glm(am ~ mpg + hp + disp,
                       family="binomial",
                       data=mtcars)
summary(myLogisticModel)
```

Calculate the odds ratios for this model.

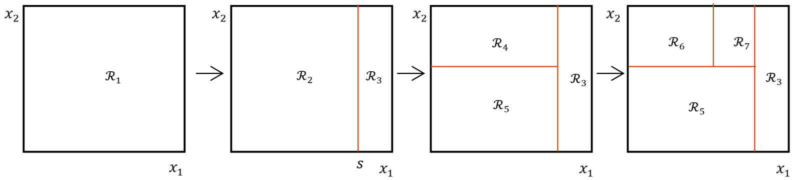
```
# Calculate the odds-ratios by taking the exponential of the coefficients
# You may also calculate the 95% confidence interval of the odds-ratio
exp(cbind(oddsratio = myLogisticModel$coefficients,
          confint(myLogisticModel)))

##                 oddsratio        2.5 %      97.5 %
## (Intercept) 2.066683e-15 9.104693e-49  0.02539238
## mpg         3.614582e+00 1.198283e+00 61.43800377
## hp          1.161095e+00 1.049568e+00  1.50974362
## disp        9.366496e-01 8.197068e-01  0.98685201
```

# Tree-based Methods

# Recursive Partitioning

- Cut off point is denoted as $s$.
- Divides data into regions (leaves) $\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \ldots$ recursively.
- Works with real values as well as categorical variables.
- Large tree risks overfitting
  - Removes weaker leaves.
  - Regularisation.



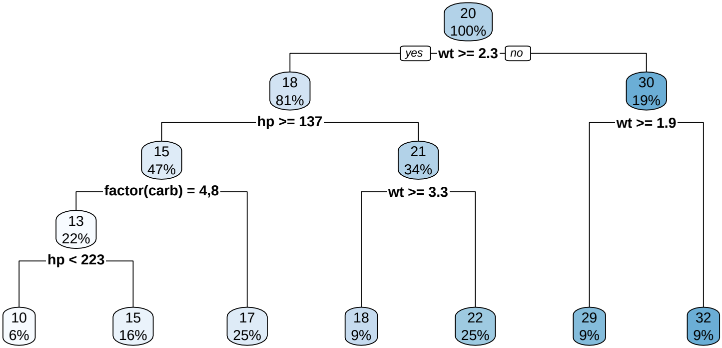
# Decision Tree

- Trees can be trained with a formula and optional control parameters.

```
# Load the rpart package for recursive partitioning
library(rpart)
# Build a decision tree to predict mpg
myTree <- rpart(formula = mpg ~ wt + hp +
                    factor(carb) +
                    factor(am),
                  data = mtcars,
                  control = rpart.control(minsplit=5))
# Read the detailed summary of the tree
summary(myTree)
```

# Decision Tree: Visualisation

```r
# Load the rpart.plot package for tree visualisation
library(rpart.plot)
rpart.plot(myTree)
```

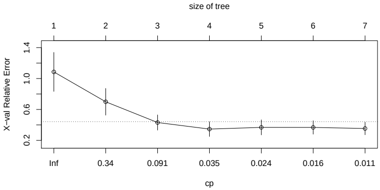


# Tree Pruning

```
printcp(myTree)

##
## Regression tree:
## rpart(formula = mpg ~ wt + hp + factor(carb) + factor(am), data = mtcars,
##     control = rpart.control(minsplit = 5))
##
## Variables actually used in tree construction:
## [1] factor(carb) hp           wt
##
## Root node error: 1126/32 = 35.189
##
## n= 32
##
##          CP nsplit rel error  xerror     xstd
## 1 0.652661      0  1.000000 1.08570 0.252677
## 2 0.178618      1  0.347339 0.69922 0.172943
## 3 0.046269      2  0.168721 0.43098 0.098563
## 4 0.026109      3  0.122453 0.34614 0.095951
## 5 0.022593      4  0.096343 0.36810 0.097621
## 6 0.011989      5  0.073751 0.36754 0.087290
## 7 0.010000      6  0.061762 0.35358 0.081453

plotcp(myTree)
```

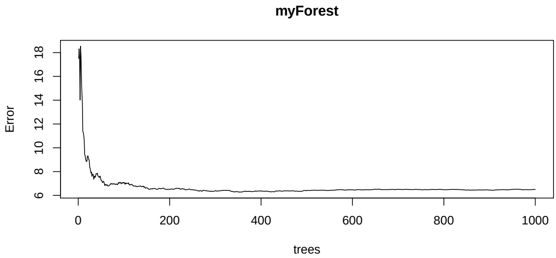


# Random Forest

- Consists of many decision trees
  - Randomly selected variables will be used in each split
  - Usually no need to prune them (all trees are allowed to grow big)
- $M$ trees in a random forest produces $M$ predictions
  - Final prediction is calculated as mean value for regression problem
  - Classification problem will use most the common label (majority voting)

# Training a Random Forest

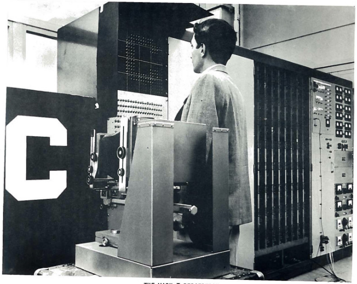
```
library(randomForest)
library(dplyr)
# Build a random forest with 1000 trees
# Each tree has 2 randomly selected variables
# You can change the parameters
myForest <- randomForest(mpg ~ wt + hp + carb + am,
                         ntree = 1000,
                         mtry = 2,
                         data = mtcars %>% mutate(carb = factor(carb),
                                                  am = factor(am)))
# Plot the error as the forest expands
plot(myForest)
```



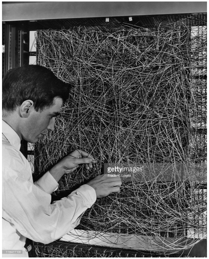
# Neural Networks

# Artificial Neurons

- Inspired by neurons in biological brain.
- McCulloch and Pitts described neuron as a logical process.
  - Neuron takes several inputs $\{x_1, x_2, x_3, ..., x_M\}$
  - Fires (activates) if the combined weighted input $\sum_{m=1}^{M} w_m x_m$ exceeds threshold.
  - Output can either be fire 1 or not fire 0.
- Rosenblatt's Mark I Perceptron



THE MARK I PERCEPTRON



# Modern Neural Networks

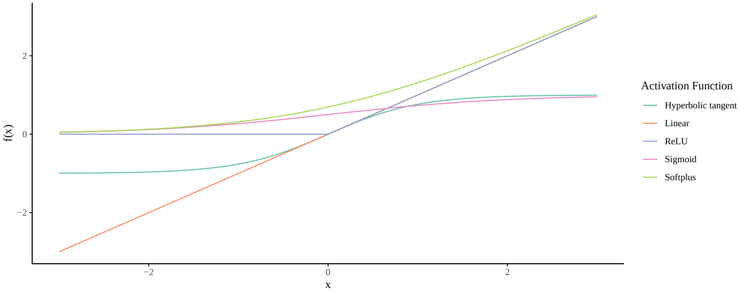
- Neural nets are based on non-linear processing power.
- Trained via gradient descent optimisers (backpropagation).
  - Network initialise randomly.
  - Requires differentiable loss function.
  - Requires strong gradient in order to improve.
  - Model weights improve iteratively.
  - Converge at local minimum.
- Neurons can be stacked as layers.
  - Can either be shallow (1 layer) or deep (many layers).
  - State-of-the art neural networks have highly bespoke topologies.

# Activation

- Weighted inputs are combined linearly.

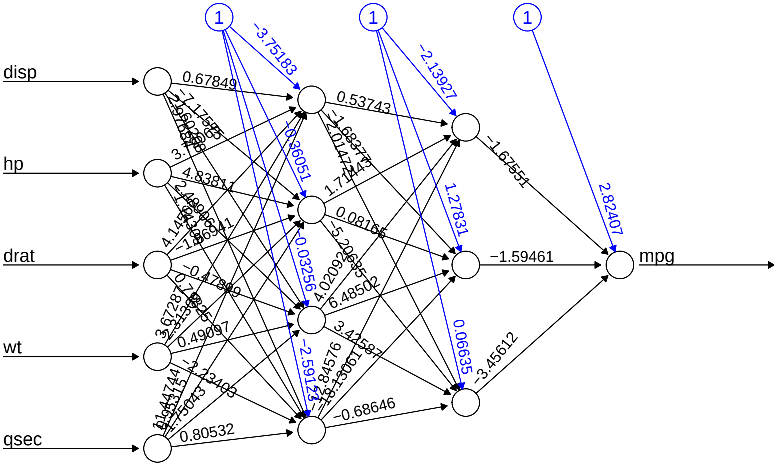
$$X = \sum_{m=1}^{M} w_m x_m$$

- Non-linear activation functions
  - Sigmoid
  - Hyperbolic tangent
  - etc...



# Multilayer Perceptron: Topology

- Layers are fully-interconnected.
- Usually having two or more layers.




Error: 0.045594   Steps: 2263

# Time Series Analysis

# Time Series Data

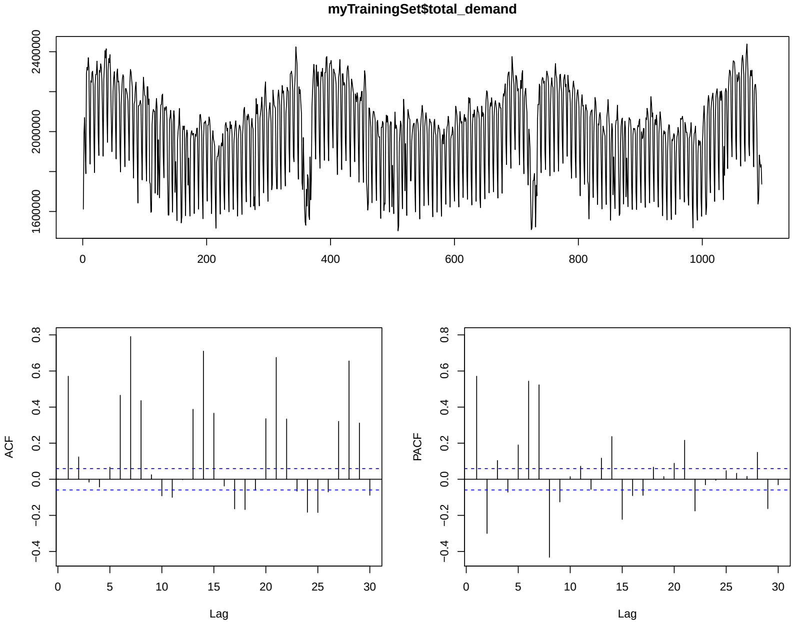
- Observations repeatedly taken at regular interval.
- Explore variable relationship across temporal space.

  **Auto-correlation Function (ACF)** Measures the correlation of a single variable along the temporal dimension between $x_t$ and $x_{t+h}$. It shows the correlation of the variable over different lag periods. For most time series variables, correlation is usually strong at lag $h = 1$ and it gradually diminishes as lag period increases. Cyclic pattern in the correlogram suggests possible seasonality which you can analyse further.

  **Partial Auto-correlation Function (PACF)** Also measures the correlation between different lag periods, but it controls the correlation across the temporal dimsnion so that only the contribution of an individual lag is reflected.

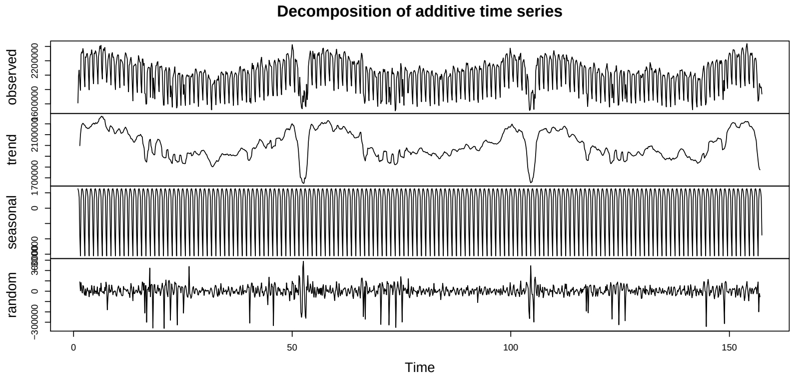
  **Cross Correlation Function (CCF)** Analyses the temporal correlation between two variables.

# ACF and PACF Correlograms



# Decomposition

- Time series can be decomposed into:
  - Seasonal component $S_t$
  - Trend component $T_t$
  - Residual component $\epsilon_t$
- Additive time series is expressed as $X_t = S_t + T_t + \epsilon_t$
- Multiplicative time series is expressed as $X_t = S_t \times T_t \times \epsilon_t$



# Time Series Linear Regression Model

- Decomposed components can be used as independent variable in linear regression:

$$X_t = \beta_0 + \beta_{trend} T_t + \beta_{seasonal} S_t + \sum_{m=1}^{M} (\beta_m x_{mt}) + \epsilon_t$$

- Components can also form interaction terms with other independent variables.

# Auto-regressive Moving Average Model (ARMA)

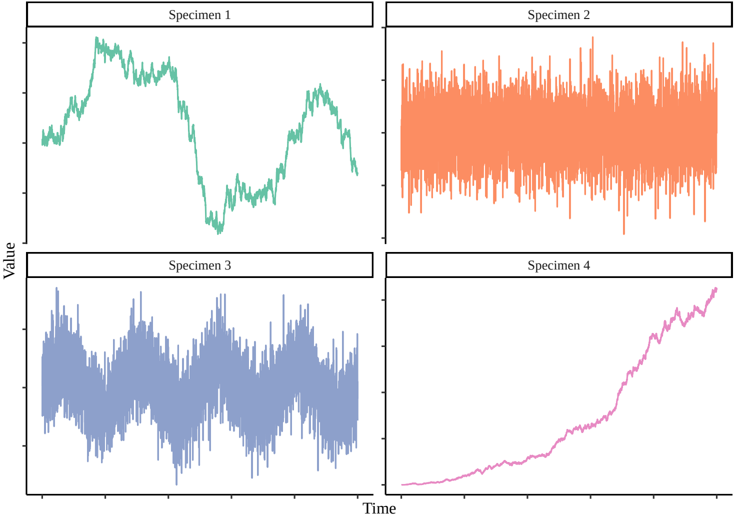
- $ARMA(p, q)$ model

$$\underbrace{X_t}_{\text{Observation}} = \underbrace{\beta_0}_{\text{intercept}} + \underbrace{\sum_{i=i}^{p}(\phi_i X_{t-1})}_{\text{AR(p)}} + \underbrace{\sum_{i=1}^{q}(\theta_i \epsilon_{t-i})}_{\text{MA(q)}} + \underbrace{\epsilon_t}_{\text{residual}}$$

- $ARIMA(p, d, q)$ model
  - Auto-regressive Integrative Moving Average
  - $I(d)$ $d^{th}$ order integration can be added.
    - Integration refers to the difference from previous time step
    - First order differencing $I(1) : X_t^{'} = X_t - X_{t-1}$
    - To satisfy **stationarity** requirement.

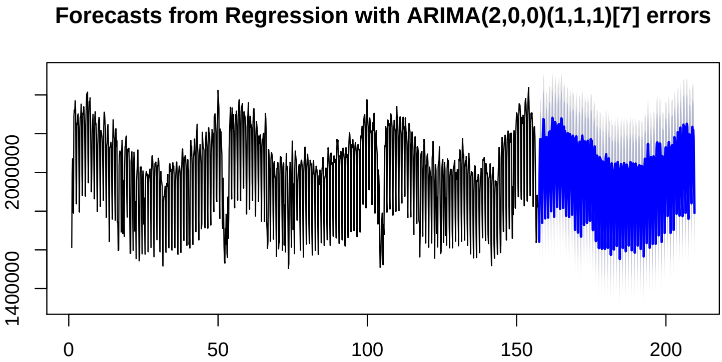
# Stationarity

- Equal properties (mean and variance) across time.
  - Only one below is a stationary time series.
  - Which one is it?



# ARIMA with Seasonality (SARIMA)

- $ARIMA(p, d, q)(P, D, Q)_m$
  - All parameter values can be automatically identified.
  - Simple models are always preferred
  - Intend to keep $p + q + P + Q$ small.



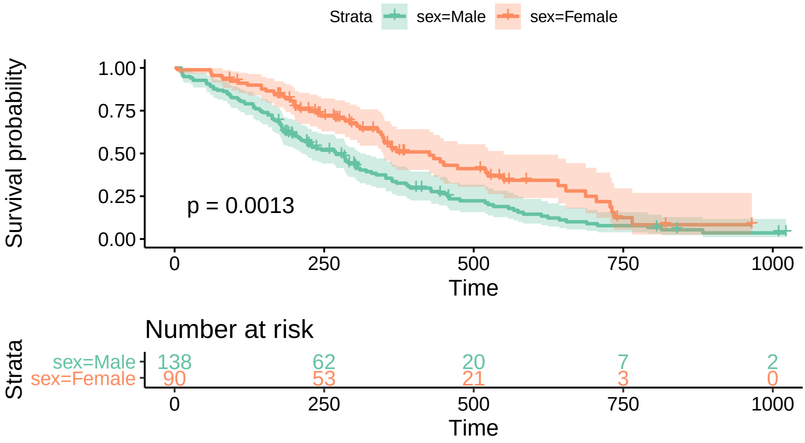
# Survival Analysis

# Survival Analysis

- Event occuring at irregular intervals.
  - e.g. Patients gets sick, machine failing... etc
- Also known as **time-to-event analysis** or **event history analysis**.

# Kaplan-Meier Estimator

- It is used to measure how many subjects survives in a clinical trial since treatment began.
- Categorical variables only.



# Cox Proportional Harzard Model

- It is a regression method which can take into account categorical and numeric variables.
- Assumes effects are time-independent (proportional harzard assumption).
- Harzard function $h_t$ is defined as:

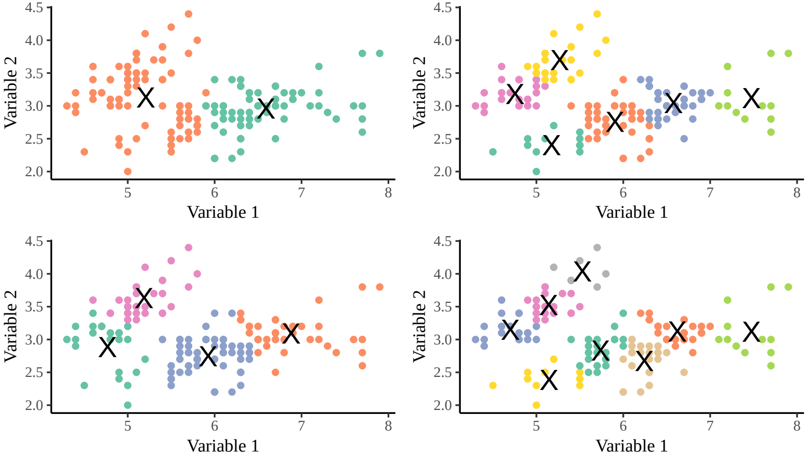
$$h_t = h_{0,t} \times e^{\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \ldots + \beta_M x_M}$$

```
# Build a Cox model with predictor variables
myCoxModel1 <- coxph(Surv(time, status==2) ~ factor(sex) + age +
                     ph.ecog + ph.karno +
                     pat.karno +
                     meal.cal + wt.loss, data = lung)
# Read the model summary
summary(myCoxModel1)
```

# Unsupervised Learning

# K-means Clustering

- Works with unlabelled data.
- Arrange objects into groups (clusters)
- Objective interpretation of results as $K$ is arbitrarily selected.



# Agglomerative Hierarchical Clustering

- $N$ objects can form maximum $N$ clusters, each having 1 member object.
- Identify distance between closest cluster pair.
- Merge them.
- Repeat until there are no more clusters left.

# Agglomerative Hierarchical Clustering: Example