Modulo 1 Geostatistica - corso base

Francesco Pirotti

2024-05-30

Contents

1 Introduzione ad R **2**

1.1 Cosa impareremo . . . 2
1.2 Materiale . . . 3
1.3 R: panoramica . . . 3
1.4 RStudio . . . 3
1.5 Righe codice - i comandi . . . 4
1.6 Commenti e sezioni . . . 4
1.7 Documentazione/help . . . 5
1.8 Packages/Librerie . . . 5
1.9 Packages/Librerie . . . 6
1.10 Variabili e funzioni . . . 6
1.11 Variabili e funzioni: Scope . . . 6
1.12 Strutture dati in R . . . 7
1.13 Tipo dati: vettori . . . 7
1.14 Tipo dati: matrix . . . 8
1.15 Tipo dati: array . . . 8
1.16 Tipo dati: List . . . 9
1.17 Tipo dati: List c/nomi . . . 10
1.18 Tipo dati: Data frame . . . 10
1.19 Tipo dati: Data frame . . . 11
1.20 Tipo dati: Tibble . . . 11
1.21 Convertire oggetti . . . 11
1.22 Salvare oggetti R . . . 12
1.23 Operatori di R . . . 12
1.24 Esercizio . . . 12

2 Data in/process/out **13**

2.1 Cosa faremo . . . 13
2.2 Input dati: CSV . . . 13
2.3 Input dati: Fogli excel . . . 13
2.4 Grafici di base . . . 14
2.5 Grafici di base . . . 15
2.6 Input dati: JSON . . . 16

3 Trasformazione dei dati **16**

3.1 Introduzione . . . 16
3.2 Iteratori . . . 17
3.3 Iteratori tidy . . . 17
3.4 Convertire tipologie di colonne . . . 17
3.5 Filtro e selezione colonne . . . 18
3.6 Mutate . . . 18
3.7 Aggregazioni . . . 18
3.8 Unire tabelle (join) . . . 18
3.9 Unire tabelle (join) . . . 19
3.10 Ristrutturazione: wide=>long . . . 20
3.11 Ristrutturazione: long=>wide . . . 21
3.12 Grafici con GGplot . . . 21
3.13 Grafici con GGplot . . . 22
3.14 Grafici e dati geo . . . 23

4 Dati spaziali **24**

4.1 Introduzione . . . 24
4.2 Dati vettoriali . . . 24

1 Introduzione ad R

1.1 Cosa impareremo

- struttura di R (base e pacchetti), potenzialità
- RStudio: come funziona (panoramica)
- documentazione, esempi, snippets per imparare
- principali categorie di dati: oggetti, variabili e funzioni in R
- assegnazione ed utilizzo variabili
- leggere, elaborare e scrivere dati strutturati (tabelle) e non

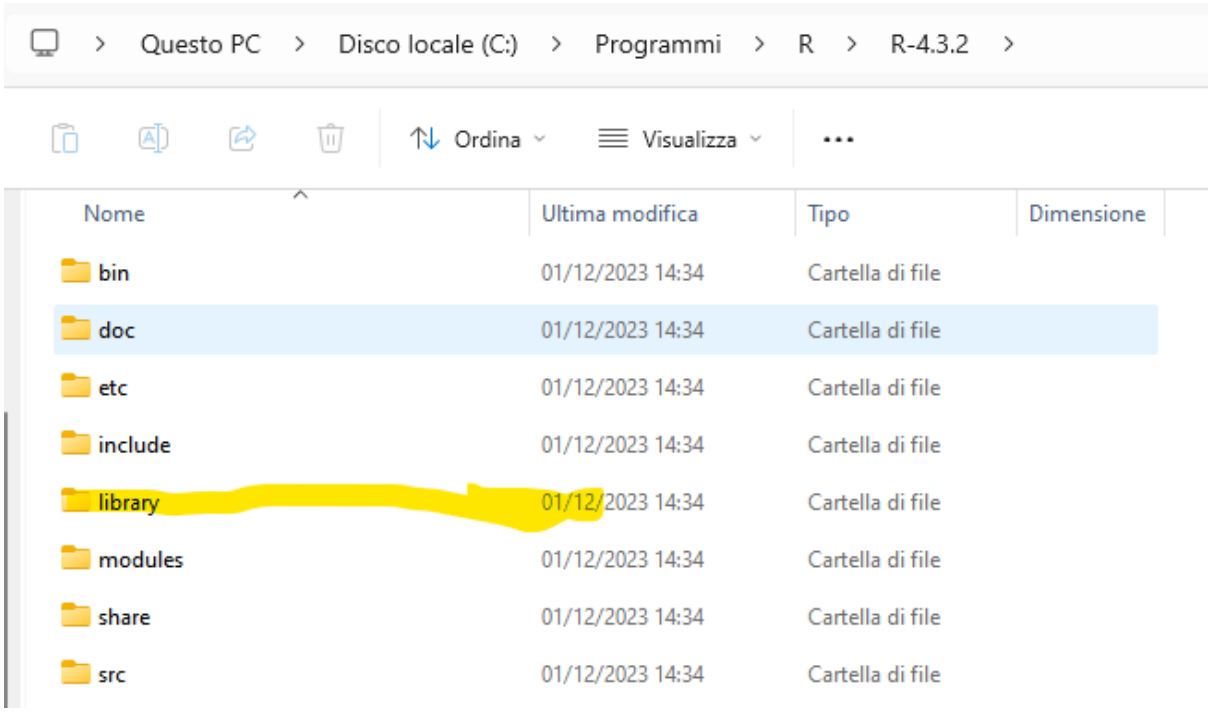
1.2 Materiale

Trovate il materiale in una pagina GitHub QUI ed anche direttamente nello spazio web QUI. Troviamo:

- slides in formato HTML
- slides in formato Power Point
- contenuto slides in formato di documento A4 PDF

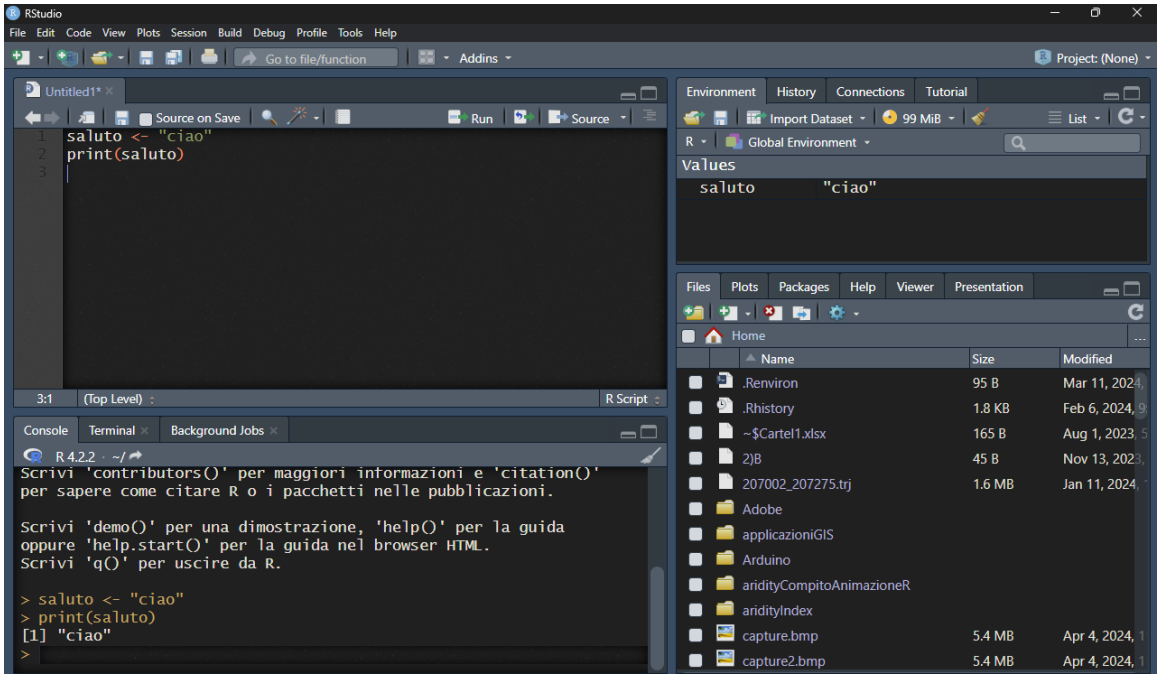
1.3 R: panoramica

- Installazione di R, versioni
- struttura di R - percorso di installazione



1.4 RStudio

- versione desktop/server
- vantaggi interfaccia:
  - console/terminale/background
  - progetti/packages/help
  - Environment/History/Connection/Tutorial
- Salvare un progetto, cartella di progetto, GIT e GitHub
- Working Directory predefinita, uso del tab per richiamare percorsi



1.5 Righe codice - i comandi

R è un programma basato su righe di codice con comandi che eseguono elaborazioni su dati.

L'utente immette i comandi al prompt (>) e ciascun comando viene eseguito uno alla volta andando a capo.

Le righe di comando solitamente vengono salvate in un file "script" con estensione "R" (.R) e vengono eseguite una alla volta mediante "invio" o con selezione multipla e "invio".

Con RStudio è possibile eseguire l'intero file, fermandosi eventualmente in punti specifici "breakpoints" (lo vedremo durante il corso).

Esercizio: esegui comando della figura alla slide precedente.

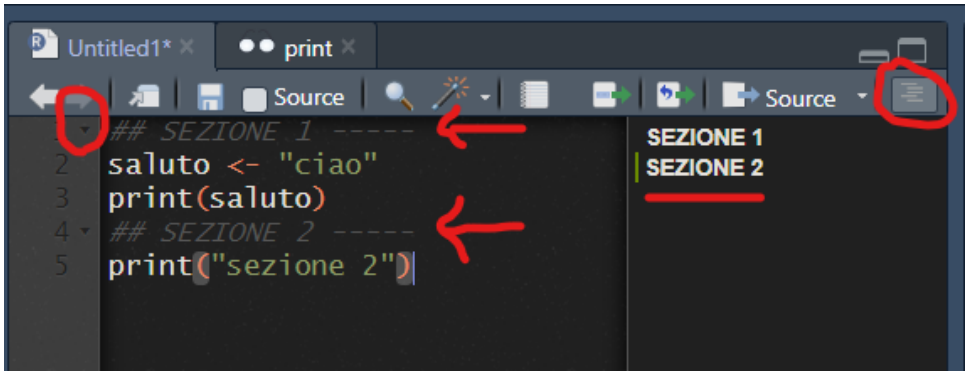
```
saluto <- "ciao"
print(saluto)
```

```
## [1] "ciao"
```

1.6 Commenti e sezioni

Se non si vuole eseguire delle righe, basta mettere un carattere asterisco (#) all'inizio del testo che NON si vuole eseguire.

NB se inserito all'inizio della riga, e seguito da testo e da 4 o più caratteri - o #, il testo diventa una sezione



1.6.1

1.7 Documentazione/help

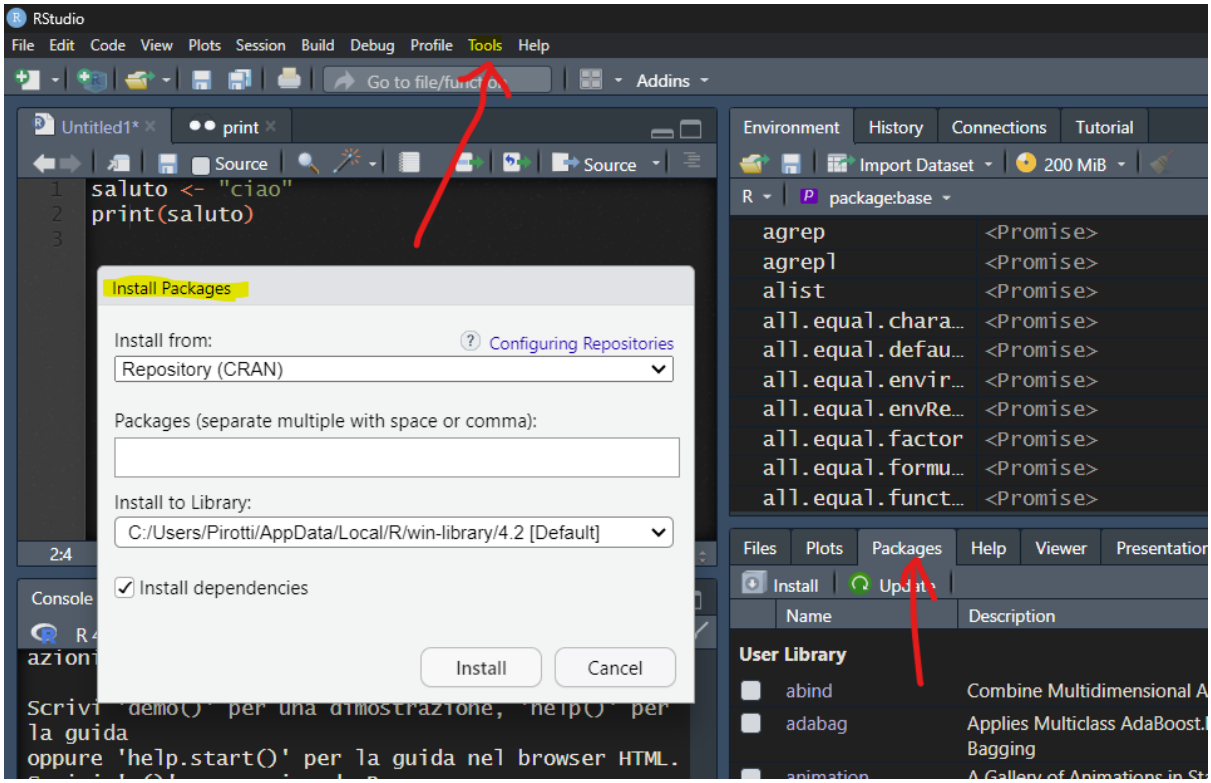
Ogni singola funzione ha ampia documentazione con molti esempi. Chiamando una funzione dopo uno o due punti interrogativi richiama la documentazione. Quasi sempre gli esempi sono eseguibili facendo copia/incolla

Esercizio: esegui il primo esempio dalla documentazione della funzione *print*

```
?print
??print
```

1.8 Packages/Librerie

Tantissime funzionalità aggiuntive sono disponibili su componenti aggiuntivi che vanno installate con il comando `install.packages(<nome libreria>)` e poi caricate con il comando `library(<nome libreria>)`. Da RStudio possiamo caricarle da interfaccia grafica.



1.9 Packages/Librerie

Non tutti sono nella "repository" CRAN, alcuni sono su GitHub o scaricabili da altre fonti - vedi alcune cose sotto

- https://github.com/fpirotti/rPET
- https://github.com/fpirotti/CloudGeometry

Per installare librarie presenti solo in CRAN si utilizza una libreria chiamata "devtools"

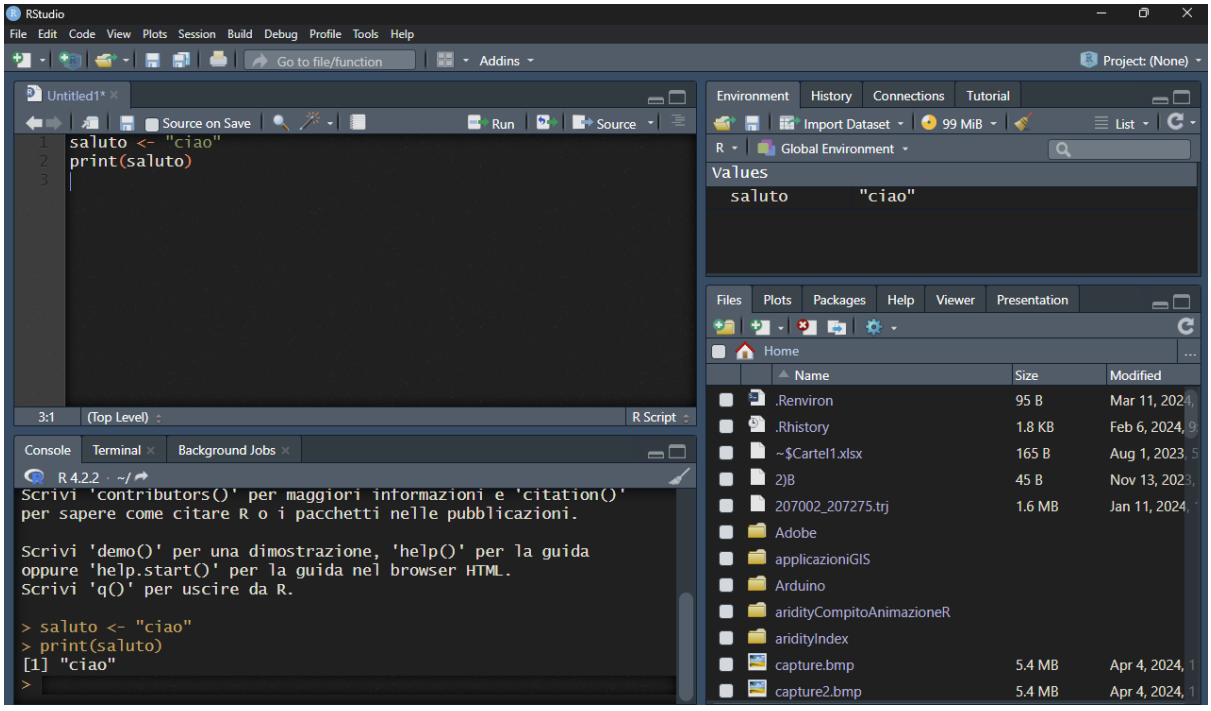
```
# install.packages("devtools")
devtools::install_github("fpirotti/CloudGeometry")
```

1.10 Variabili e funzioni

Qui vediamo una variabile ed una funzione.

NB1 - la variabile "saluto" è nel Environment ("ambito/ambiente/campo") *globale**. La funzione "print" è nel campo del package "base" - tieni premuto il tasto CTRL e seleziona il nome della funzione - vedi cosa succede.

NB2 - operatore di assegnazione `<-` (o `<<-` nel caso si voglia forzare l'assegnazione ad una variabile *globale**)



1.11 Variabili e funzioni: Scope

*Le variabili create al di fuori di funzioni sono note come variabili ***globali*****;** possono essere utilizzate sia all'interno delle funzioni che all'esterno.

Sotto andiamo a creare una nostra funzione "salutami" che esegue il saluto. Provate a modificare l'operatore di assegnazione da `<<-` a `<<-` e rieseguire!!

```r
salutami <- function(){
  saluto <- "ciao ARPA!!!"
  print(saluto)
}

print(saluto)
```

```
## [1] "ciao"
```

```r
salutami()
```

```
## [1] "ciao ARPA!!!"
```

```r
print(saluto)
```

```
## [1] "ciao"
```

1.12 Strutture dati in R

NB ogni elemento in R è considerato (ed è) un VETTORE. Le funzioni di R considerano ogni variabile un vettore. Cosa significa? Che le funzioni elaborano tutti gli elementi di un vettore "by default" e che ogni elemento è indicabile con un numero iniziando da 1 (non da 0 come solitamente succede in altri linguaggi).

- vector
- character
- integer
- numeric

```r
miaVar <- FALSE
class(miaVar)
miaVar[[1]]
miaVar[[2]]
```

1.13 Tipo dati: vettori

Perchè succede quello che vedete sotto (dati *numeric* diventano *character*)?

```r
miaVar <- c(1,4,6,8)
class(miaVar)
```

```
## [1] "numeric"
```

```r
miaVar[[1]]
```

```
## [1] 1
```

```r
miaVar[[2]]
```

```
## [1] 4
```

```r
miaVar[[2]] <- "evviva"
class(miaVar)
```

```
## [1] "character"
```

```r
print(miaVar)
```

```
## [1] "1"      "evviva" "6"      "8"
```

1.14 Tipo dati: matrix

Matrix è un oggetto con struttura di matrice ovvero bidimensionale (righe × colonne); pensate a un gruppo di vettori impilati o affiancati.

Si accede e si assegnano i valori con [r,c] dove r e ci sono gli indici di riga e colonna. Si può lasciare vuoto un indice per accedere alla riga/colonna

```r
mat <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
mat[[2]]
```

```
## [1] 2
```

```r
mat[1,2]
```

```
## [1] 3
```

```r
mat[1,]
```

```
## [1] 1 3
```

```r
mat[1,2] <- 100
```

1.15 Tipo dati: array

Un array è una matrice multidimensionale.

r=rows, c=columns, m=matrice.... etc...

> Esercizio: vedi sotto come creare un array a 3 dimensioni. Nota che duplica 9 valori 2 volte. Prova a dare 8 valori invece che nove. Prova a dare 3 valori. Cosa succede.

```r
# 2 vettori di valori
valori1 <- c(5, 9, 3)
valori2 <- c(10, 11, 12, 13, 14, 15)
column.names <- c("C1", "C2", "C3")
row.names <- c("R1", "R2", "R3")
matrix.names <- c("Matrix1", "Matrix2")

# Crea un array a tre dimensioni
arr <- array(c(valori1, valori2), dim = c(3, 3, 2),
                  dimnames = list(row.names,
                                  column.names,
                                  matrix.names))
print(arr)
```

```
## , , Matrix1
##
##    C1 C2 C3
## R1  5 10 13
## R2  9 11 14
## R3  3 12 15
##
## , , Matrix2
##
##    C1 C2 C3
## R1  5 10 13
## R2  9 11 14
## R3  3 12 15
```

1.16 Tipo dati: List

Le strutture vector/matrix/array, possono contenere solo una tipologia base (numeric, integer, character, logical...). Ma la struttura LIST no!

La struttura *list* è un set di dati eterogenei; opzionalmente, è possibile assegnare dei nomi a ciascun elemento nel set.

```
lista <- list(1,4,6,8)
class(lista)
```

```
## [1] "list"
```

```
lista[[1]]
```

```
## [1] 1
```

```
lista[[2]]
```

```
## [1] 4
```

```
lista[[2]] <- "evviva"
class(lista)
```

```
## [1] "list"
```

```
lista[[1]]
```

```
## [1] 1
```

```
lista[[2]]
```

```
## [1] "evviva"
```

```
print(lista)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "evviva"
##
## [[3]]
## [1] 6
##
## [[4]]
## [1] 8
```

1.17 Tipo dati: List c/nomi

NB, la struttura LIST non è altro che un set, una "lista", di oggetti associata ad un indice. L'indice è un numero intero ma può essere un testo (simile al concetto di coppie "key->value"/chiave->valore).

Può essere assegnato un indice/chiave per riferimento all'elemento nella lista

```
names(lista)
```

```
## NULL
```

```
names(lista) <- c("primo", "secondo")
lista[[1]]
```

```
## [1] 1
```

```
lista[["primo"]]
```

```
## [1] 1
```

Svantaggi di strutture tipo *list*: usa più memoria!

> **Nota su allocazione memoria**: se dovete gestire volumi importanti di dati, considerate la pre-allocazione della memoria SE conoscete la dimensione. Vedi anche blog qui.

```
vettoreMoltoGrande = numeric(1000)
vettoreMoltoGrande[[100]]
```

```
## [1] 0
```

1.18 Tipo dati: Data frame

Un *data frame* è una struttura di tipo *list* ma con un numero uguale di "righe" per ogni colonna di attributi. È possibile manipolare i data frame filtrando sulle righe e operando sulle colonne.

Sia righe che colonne possono avere identificativi.

```
c.lat <- c(45.1, 45.2, 45.3)
c.lon <- c(11.1, 11.2, 11.3)

df.geo.stz <- data.frame(stz=c("A", "B","C"),
                         longitudine=c.lon,
                         latitudine=c.lat)

colnames(df.geo.stz)
```

```
## [1] "stz"         "longitudine" "latitudine"
```

```
rownames(df.geo.stz)
```

```
## [1] "1" "2" "3"
```

```
rownames(df.geo.stz) <- df.geo.stz$stz
rownames(df.geo.stz)
```

```
## [1] "A" "B" "C"
```

1.19 Tipo dati: Data frame

Le colonne sono *vettori* - si può richiamare e assegnare i valori di una colonna con `$` o `[[]]` o `[,"<nomecolonna>"]` o `[ riga,"<nomecolonna>"]`

Come con Tipo dati: matrix si possono usare gli indici di riga/colonna.

```
df.geo.stz$stz
df.geo.stz[,"stz"]
df.geo.stz[["stz"]]
df.geo.stz$stz <- c("A1","B1", "C1")
```

1.20 Tipo dati: Tibble

Un data frame particolare, lo vedremo quando usiamo l'infrastruttura di librerie "tidyverse". Vantaggi:

- Non modifica il nome mettendo un punto al posto degli spazi (Vedi esempio)
- Non utilizza mai `row.names()`. Lo scopo di *tidyverse* è quello di memorizzare le variabili in modo coerente. Quindi non memorizza mai una variabile come attributo speciale (i.e. nomi riga).
- Ricicla solo vettori di lunghezza 1. Questo perché il riciclo di vettori di lunghezza superiore è una fonte frequente di problemi nascosti.

```
library(tidyverse)
geo.stz.tb <- tibble(df.geo.stz)
class(geo.stz.tb)
geo.stz.tb
names(data.frame(`nome colonna con spazi` = 1))
names(tibble(`nome colonna con spazi` = 1))
```

1.21 Convertire oggetti

Oggetti tra loro compatibili si possono convertire; ad esempio tra matrix => data.frame usando il comando `as.data.frame(<oggetto matrix>)` viceversa attenzione che tutti i tipi di vettore vengono resi omogenei.

Provate il seguente esercizio sotto.

```
mat
df.mat <- as.data.frame(mat)
as.matrix(df.mat)
df.mat$nuovaColonna <- "testo"
as.matrix(df.mat)
```

1.22 Salvare oggetti R

Rstudio può salvare l'intero progetto con le variabili e le funzioni che vedete in alto a destra.

Il comando `save` salva in un file con estensione "rda" che viene riconosciuto anche direttamente da RStudio. NB è possibile salvare qualiasi oggetto, anche funzioni!

Prova a cliccare sul file dopo aver lanciato la prima riga del comando seguente!

```
## NB salvo anche la funzione "salutami" fatta all'inizio!
save(df.geo.stz, salutami, file="oggetti.rda")
load("oggetti.rda")
saveRDS(df.geo.stz, file = "geo.stz.RDS")
df.geo.stz <- readRDS("geo.stz.RDS")
```

1.23 Operatori di R

Gli operatori aritmetici e logici di R funzionano sia su singoli scalari che su vettori e strutture come *array* e *matrix*.

NB questo vuol dire che si possono eseguire operazioni su tutti i singoli valori della struttura internamente (*vectorization*)!

```
c.lat + 10
```

```
## [1] 55.1 55.2 55.3
```

```
c.lat == c.lon
```

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

```
c.lat[[1]] <- 44
c.lat <- c.lat - 1
## AND / OR
c.lat == c.lon & c.lat==0
```

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

```
c.lat == c.lon | c.lat==45.1
```

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

```
#c.lat == c.lon && c.lat==c.lat
#c.lat == c.lon || c.lat==c.lat
```

1.24 Esercizio

- Creare una funzione chiamata *crea.df* che esegue la seguente operazione:
  - creazione di un oggetto data.frame con 2 colonne, (Stazione, ValoreX) e 4 righe. Inventate voi i valori. Calcolo una terza colonna con il quadrato di ValoreX.
- Eseguire la funzione per creare un oggetto di tipo "tibble" chiamato *tabella.tb*
- **Consegnate** un file RDA contenente i due oggetti, la funzione *crea.df* e il tibble *tabella.tb* - chiamate il file con: es1_cognome_nome.rda, e.g. *es1_pirotti_francesco.rda*

2 Data in/process/out

2.1 Cosa faremo

Impareremo a leggere/trasformare/scrivere dati da fonti diverse:

- file di testo strutturati (**CSV**)
- fogli di lavoro (**XLSX**)
- dati online non strutturati come **JSON**

2.2 Input dati: CSV

I file CSV sono notoriamente file di testo strutturati come tabelle e molto utilizzati per condividere dati.

Il comando standard è `read.csv` .

NB non è ottimizzato per l'efficienza. Per dati massivi usare il comando `fread` dalla libreria data.table `data.table::fread()`.

Facciamo un esercizio con dati ARPA! Scarichiamo dalla pagina QUI o direttamente da QUI i dati di precipitazione mensile - salvateli in una sottocartella "dati" e caricateli usando sia la funzione read.csv che fread dalla libreria data.table (se l'avete altrimenti vedete a schermo i risultati). Si vuole notare le differenze e come si chiama una funzione SENZA caricare una libreria, usando `::`.

```
prec <- read.csv("dati/Prec_mensili_2020.csv")
prec2 <- data.table::fread("dati/Prec_mensili_2020.csv")
summary(prec2)
```

```
##  Precipitazioni mensili Veneto   2020 (mm)      media 1993-2019 (mm)
##  Length:13                     Min.   :   7.0   Min.   :  57.0
##  Class :character              1st Qu.:  28.0   1st Qu.:  74.0
##  Mode  :character              Median :  89.0   Median :  96.0
##                                Mean   : 180.2   Mean   : 171.5
##                                3rd Qu.: 171.0   3rd Qu.: 114.0
##                                Max.   :1171.0   Max.   :1114.0
```

> Esercizio: Importate i dati inserendo i parametri corretti di read.csv - come noterete, la prima riga NON importa correttamente. La seconda si, ma è una funzione della libreria data.table. Usate la documentazione in R per capire come aggiungere argomenti alla funzione read.csv (vedi sezione QUI). **NB** usate il tasto TAB per richiamare i percorsi, risparmiate tempo e fatica!

2.3 Input dati: Fogli excel

I fogli di calcolo in formato MS Excel sono più strutturati rispetto ad un file CSV, dato che ogni colonna ha anche lo specifico tipo di dati (numero, testo, data ecc. . .).

Per leggere/scrivere questi dati è necessario usare librerie aggiuntive.

```
prec3 <- prec2
## nuova colonna
prec3$norm2020 <- prec3$`2020 (mm)` / prec3$`media 1993-2019 (mm)`
writexl::write_xlsx( list("originale" =prec2, "elaborata"=prec3), path="dati/output.xlsx" )
prec4 <- readxl::read_xlsx("dati/output.xlsx")
```

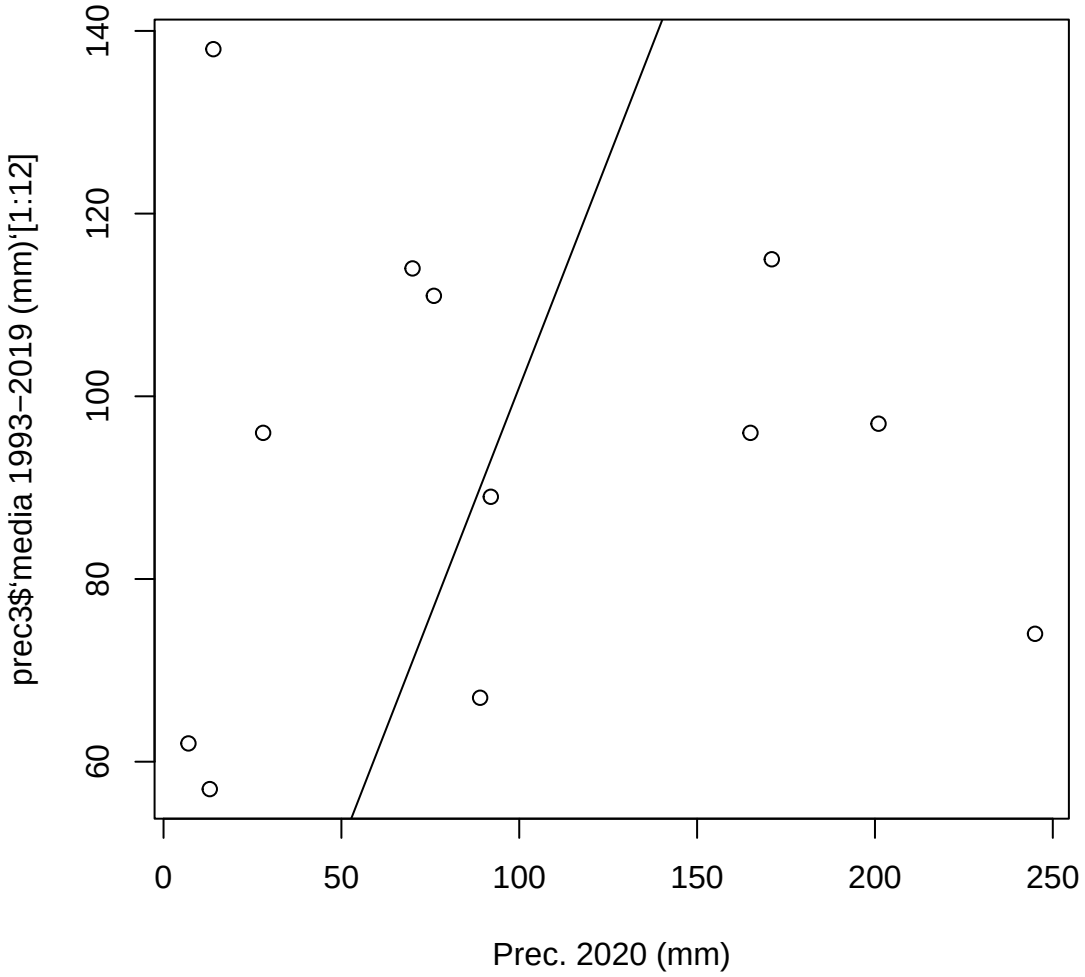
	A	B	C	D	E
1	**Precipitazioni mensili Veneto**	**2020 (mm)**	**media 1993-2019 (mm)**	**norm2020**	
2	Gennaio	13	57	0.228070175	
3	Febbraio	7	62	0.112903226	
4	Marzo	89	67	1.328358209	
5	Aprile	28	96	0.291666667	
6	Maggio	70	114	0.614035088	
7	Giugno	165	96	1.71875	
8	Luglio	92	89	1.033707865	
9	Agosto	201	97	2.072164948	
10	Settembre	76	111	0.684684685	
11	Ottobre	171	115	1.486956522	
12	Novembre	14	138	0.101449275	
13	Dicembre	245	74	3.310810811	

originale elaborata

2.4 Grafici di base

La funzione `plot` riesce a creare grafici di base. Andiamo a confrontare le precipitazioni del 2020 con quelle medie 1993-2019.

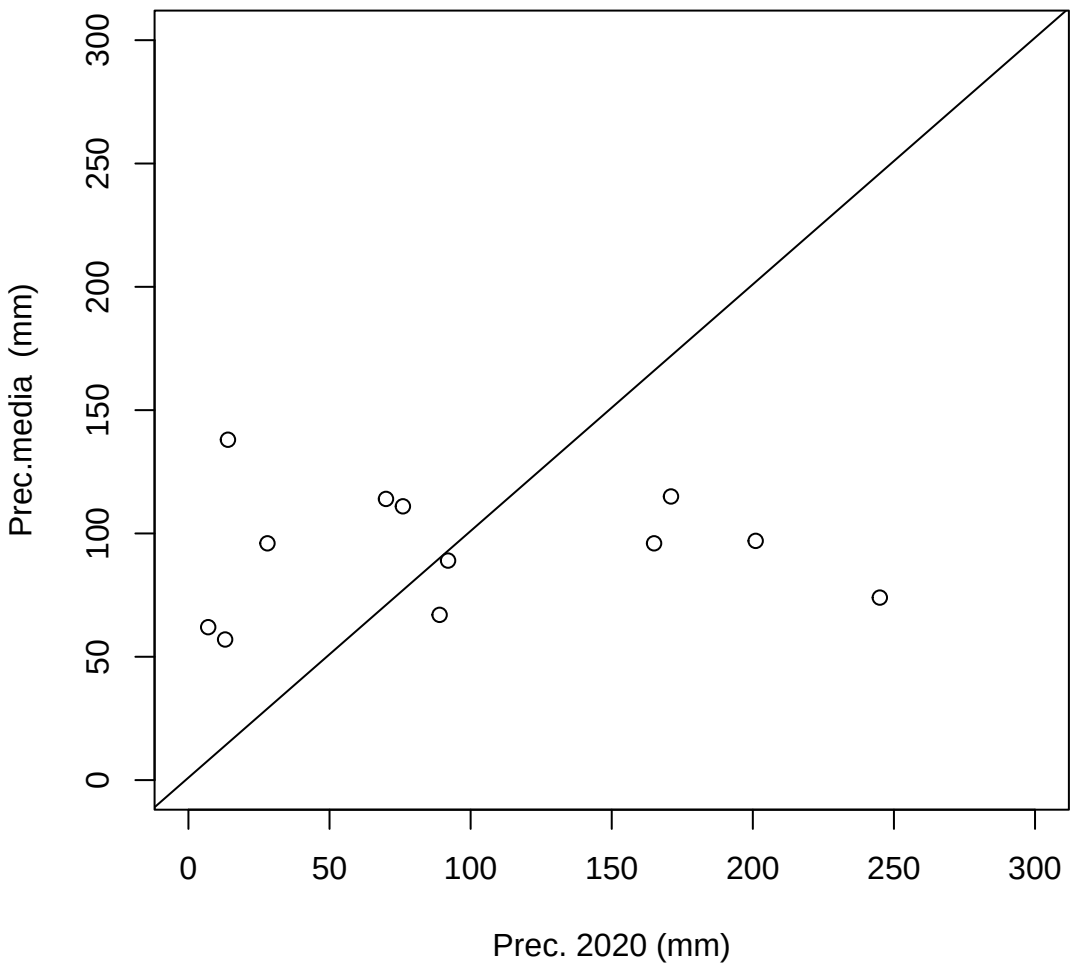
```
plot(prec3$`2020 (mm)`[1:12], prec3$`media 1993-2019 (mm)`[1:12], xlab="Prec. 2020 (mm)")
abline(1,1)
```



2.5 Grafici di base

Miglioriamo un po' il grafico - ma utilizzeremo poi la libreria ggplot2 per ottenere grafici migliori.

```
plot(prec3$`2020 (mm)`[1:12], prec3$`media 1993-2019 (mm)`[1:12],
     xlab="Prec. 2020 (mm)", ylab="Prec.media  (mm)",
     xlim=c(0,300), ylim=c(0,300))
abline(1,1)
```



2.6 Input dati: JSON

I dati JSON (**J**ava**S**cript **O**bject **N**otation) sono molto utilizzati per trasferire dati non strutturati. Vedi QUI e QUI. Anche per dati "geo" con il GeoJSON (vedi esempio QUI).

Usiamo direttamente da API ARPAV "Anagrafica stazioni in formato JSON" dalla pagina QUI

```
## Scarichiamo direttamente da API ARPAV
api.url <- "https://api.arpa.veneto.it/REST/v1/meteo_meteogrammi?rete=MGRAMMI&coordcd=18&orario=0&rn
stazioni.meteo <- jsonlite::read_json(api.url)
```

3 Trasformazione dei dati

3.1 Introduzione

Modificare dati significa aggiungere, togliere, filtrare, manipolare colonne, creare sommari e statistiche di interesse aggreando dati. Insomma il dato viene "manipolato". In inglese viene anche usato "data wrangling"

Ci “alleniamo” con i dati JSON delle stazioni meteo ARPAV scaricati dal JSON.

Obiettivo: creare un grafico con la posizione delle stazioni e aggiungere qualche valore mediante unione con altri dati.

3.2 Iteratori

Fondamentale saper utilizzare operazioni che eseguono su tutti gli elementi di un oggetto. E.g. tutte le colonne di un *data.frame*, tutte le righe di un *data.frame*, tutti gli oggetti di una *list*...

```
dati <- stazioni.meteo$data
length(dati)
## LOOP classico: visto male per
## performance bassa,
## ma non necessariamente vero
for(i in 1:length(dati)){
  print(dati[[i]])
  break / next
}

funz <- function(x){
  length(x)
}
# lapply/mapply/sapply/apply per righe/colonne
# lapply(dati, funz)
```

3.3 Iteratori tidy

L’operatore “map” è simile agli “apply” ma con più funzionalità. Vedi sotto, usando *map_df* i risultati vengono raccolti su una tabella (tibble)!

```
library(tidyverse)
## i dati sono strutturati nell'oggetto "data"
##  come liste dentro altre liste
# as.data.frame(stazioni.meteo$data)
dt <- map(.x = stazioni.meteo$data, .f = function(x){ x } )
dtf <- map_df (.x = stazioni.meteo$data, .f = function(x){ x } )
```

3.4 Convertire tipologie di colonne

Mappando a data.frame con `map` abbiamo solo colonne di tipo character... come mai?* NB con `map_df` questo non succede perchè la funzione `map_df` è “smart” e capisce qualche tipo usare per ogni colonna. TRANNE che per la colonna valore. Perchè?**

NB gli iteratori su tabelle (data.frame o tibble) lavorano per colonne.

```
## richiamo la classe di ogni colonna
# map(.x = dt, .f = function(x){ class(x) } )
map(.x = dtf, .f = function(x){ class(x) } )
## ATTENZIONE - colonna valore è corretta?
dtf$valore <- as.numeric(dtf$valore)
dtf$tipo <- as.factor(dtf$tipo)
levels(dtf$tipo)
```

*Perchè raccoglie ogni elemento della lista in un vettore ed il vettore può avere solo valori di un tipo base (*character, numeric, integer, logical...*) - sceglie sempre il *character* se valori misti

** Nota che nel JSON il numero in "valore" è tra virgolette, dunque viene visto come testo (character) e non come numerico. Confronta con latitudine e longitudine!

3.5 Filtro e selezione colonne

Operazioni usando tidyverse (e altri sistemi) usano operatori di concatenzione di processi. In questo caso si usa %>% ovvero un operatore che indica di usare l'output di un processo come input al processo successivo. Un esempio con un'operazione di filtro con successiva selezione di colonne:

```
## Selezioniamo solo le stazioni in provincia di Belluno
dtf.belluno <- dtf %>% filter( provincia =="BELLUNO")
## Se aggiungiamo criteri di filtro divisi da virgola è come usare AND
dtf %>% filter( provincia !="BELLUNO", valore < 10)
dtf %>% filter( provincia !="BELLUNO" & valore < 10)
## Se aggiungiamo criteri di filtro è come usare AND
dtf %>% filter( provincia =="BELLUNO" | valore < 12) %>% select(nome_stazione, valore)
```

3.6 Mutate

Il comando mutate consente di calcolare nuove colonne in funzione di quelle esistenti.

```
dtf %>% mutate(valore=valore^2)  %>% select(valore)
```

3.7 Aggregazioni

Operazioni di aggregazione con `group_by`

Attenzione - la funzione `mean` restituisce `NA` se tra i valori trova degli `NA`, per evitare questo comportamente mettere "`na.rm = TRUE`".

```
dtf %>% group_by(tipo)  %>% summarise(valore=mean(valore))
```

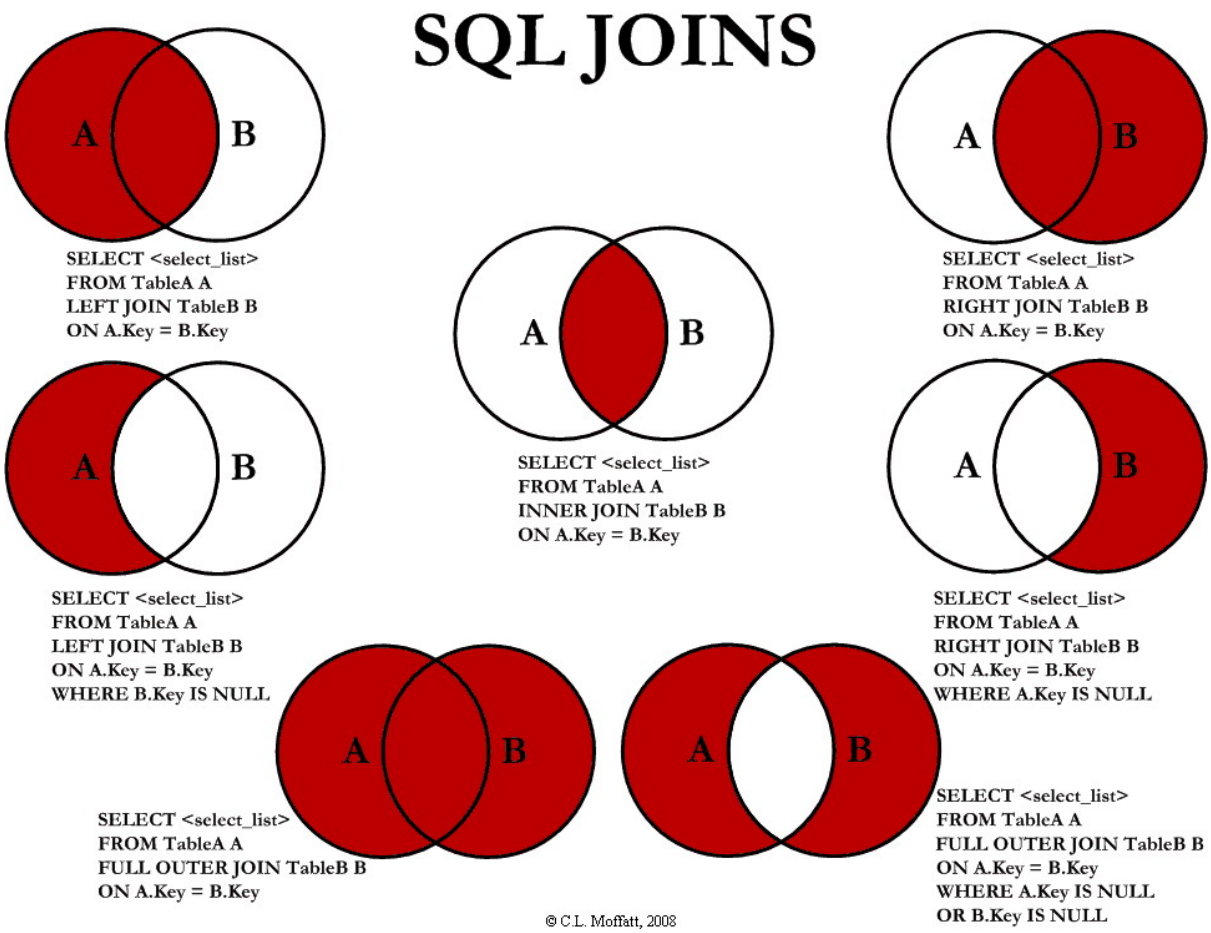
```
## # A tibble: 3 x 2
##   tipo    valore
##   <fct>    <dbl>
## 1 TARIA2M  NA
## 2 TARIA5M   5.54
## 3 TARIA8M  NA
```

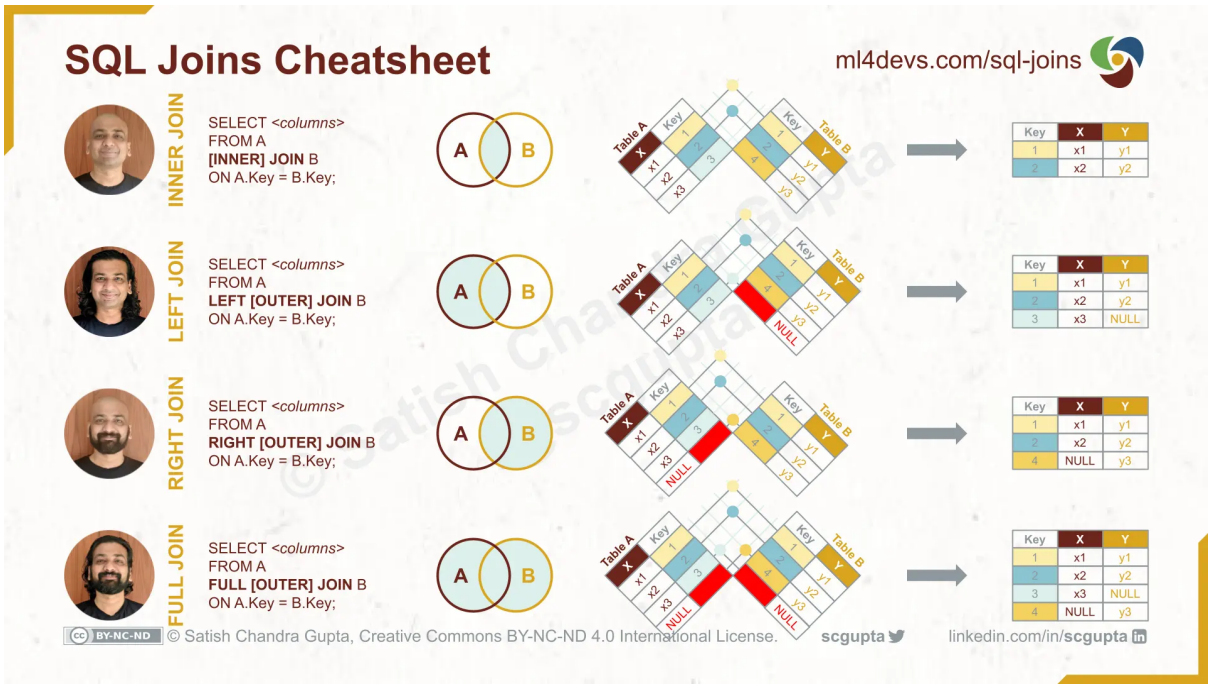
```
dtf %>% group_by(tipo)  %>% summarise(valore=mean(valore, na.rm = TRUE))
```

```
## # A tibble: 3 x 2
##   tipo    valore
##   <fct>    <dbl>
## 1 TARIA2M  15.4
## 2 TARIA5M   5.54
## 3 TARIA8M   8.65
```

3.8 Unire tabelle (join)

I join sono operazioni note in ambito di gestione dei dati, servono per aggiungere in senso orizzontale valori usando una o più colonne con valori corrispondenti.





3.9 Unire tabelle (join)

Vediamo un esempio pratico: andiamo ad unire i valori di oggi a quelli di alcuni giorni fa alle nostre stazioni. Scarica qui il dato CSV.

```
dt.past <- read.csv("dati/ARPA_temperatura_20240528.csv")
dt.past <- dt.past %>% select(statnm, statcd, dataora, media)
dt.joined <- dt.past %>%  left_join(dtf, by=c("statcd"="codice_stazione") )
```

3.10 Ristrutturazione: wide=>long

Questo tipo di modifica è molto comune per rappresentare i dati in modo differente. Vediamo un esempio pratico:

```
## andiamo a creare classi di temperatura
classi <- cut(dtf$valore, breaks = c(-10,0,10,20,30))
## mappatura con tabella a doppia entrata
tb.wide <- table(dtf$provincia, classi)
## nota che è un oggetto "table", trasformiomo in data.frame
tb.wide <- as.data.frame.matrix(tb.wide)
rownames(tb.wide)
```

```
## [1] "BELLUNO" "PADOVA"  "ROVIGO"  "TREVISO" "UDINE"   "VENEZIA" "VERONA"
## [8] "VICENZA"
```

```
## nota ancora che le provincie sono rownames - se converto
## a tibble non ci sono più
rownames(as_tibble(tb.wide) )
```

```
## [1] "1" "2" "3" "4" "5" "6" "7" "8"
```

```
## creo colonna e relacate la mette all'inizio (provate senza "relocate")
tb.wide <- tb.wide %>% mutate(Provincie = rownames(tb.wide))%>%  relocate(Provincie)
## tabella con numero di stazioni per classe espressa "wide"
tb.wide
```

```
##         Provincie (-10,0] (0,10] (10,20] (20,30]
## BELLUNO   BELLUNO       0     10       9       0
## PADOVA     PADOVA       0      0      10       0
## ROVIGO     ROVIGO       0      0       5       1
## TREVISO   TREVISO       0      1       6       0
## UDINE       UDINE       0      1       0       0
## VENEZIA   VENEZIA       0      0       7       1
## VERONA     VERONA       0      1      11       0
## VICENZA   VICENZA       0      2      25       0
```

```
## trasformo
tb.long <- tb.wide  %>% pivot_longer(!Provincie,
    names_to = "Classe Temp. °C", values_to = "Numero"
    )
tb.long
```

```
## # A tibble: 32 x 3
##    Provincie `Classe Temp. °C` Numero
##    <chr>     <chr>              <int>
##  1 BELLUNO   (-10,0]                0
##  2 BELLUNO   (0,10]                10
##  3 BELLUNO   (10,20]                9
##  4 BELLUNO   (20,30]                0
```

```
##  5 PADOVA    (-10,0]               0
##  6 PADOVA    (0,10]                0
##  7 PADOVA    (10,20]              10
##  8 PADOVA    (20,30]               0
##  9 ROVIGO    (-10,0]               0
## 10 ROVIGO    (0,10]                0
## # i 22 more rows
```

3.11 Ristrutturazione: long=>wide

Operazione inversa per tornare alla tabella di origine

```
tb.long2wide <- tb.long  %>% pivot_wider(
  names_from = "Classe Temp. °C", values_from = "Numero"
  )
tb.long2wide
```

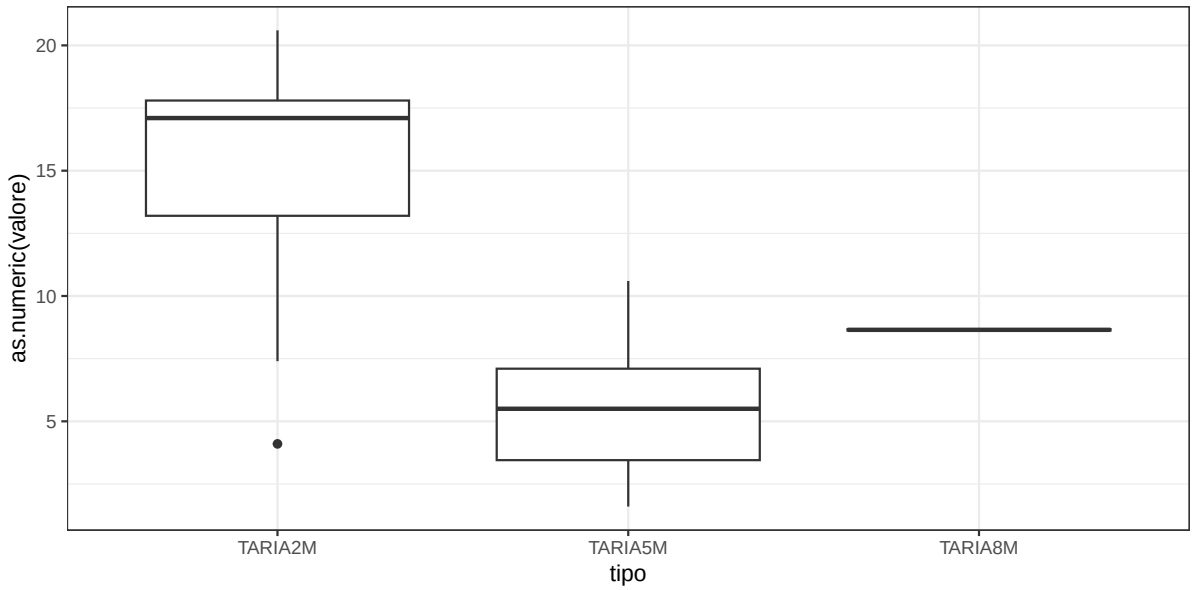
```
## # A tibble: 8 x 5
##   Provincie `(-10,0]` `(0,10]` `(10,20]` `(20,30]`
##   <chr>         <int>    <int>     <int>     <int>
## 1 BELLUNO           0       10         9         0
## 2 PADOVA            0        0        10         0
## 3 ROVIGO            0        0         5         1
## 4 TREVISO           0        1         6         0
## 5 UDINE             0        1         0         0
## 6 VENEZIA           0        0         7         1
## 7 VERONA            0        1        11         0
## 8 VICENZA           0        2        25         0
```

3.12 Grafici con GGplot

Le funzioni di base "`plot`" sono utili ma ci sono librerie che forniscono più funzionalità. Introduciamo "ggplot". Notate che viene creato un grafico concatenando con l'operatore "`+`" gli oggetti del grafico. "`aes`" sta per *aestethics* e indica colori e valori degli assi, aggregazioni ecc...

```
library(ggplot2)
ggplot(dtf) + geom_boxplot(aes(x=tipo, y=as.numeric(valore) )) + theme_bw()
```

```
## Warning: Removed 56 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```

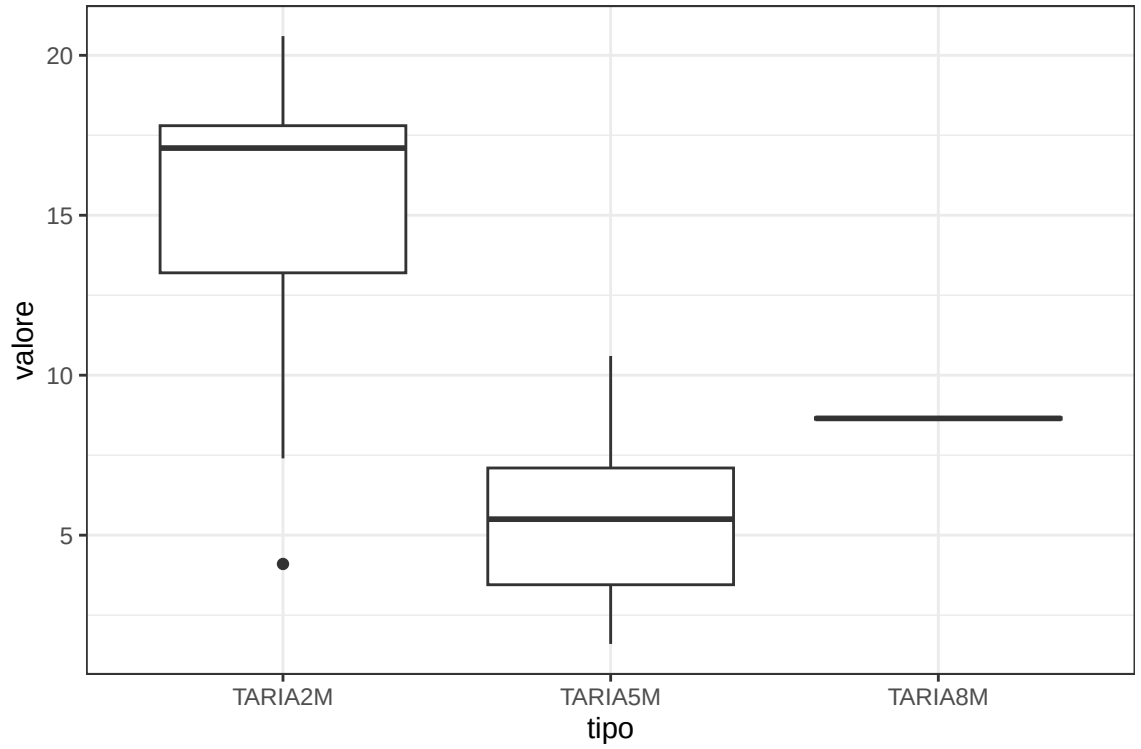


3.13 Grafici con GGplot

NB. la parte "as.numeric" in quanto la colonna anche contiene numeri è di tipo "character". Modifichiamola con "as.numeric", ma vedrete due valori mancanti (NA=not available).

```
dtf$valore <- as.numeric(dtf$valore)
ggplot(dtf) + geom_boxplot(aes(x=tipo, y=valore )) + theme_bw()
```

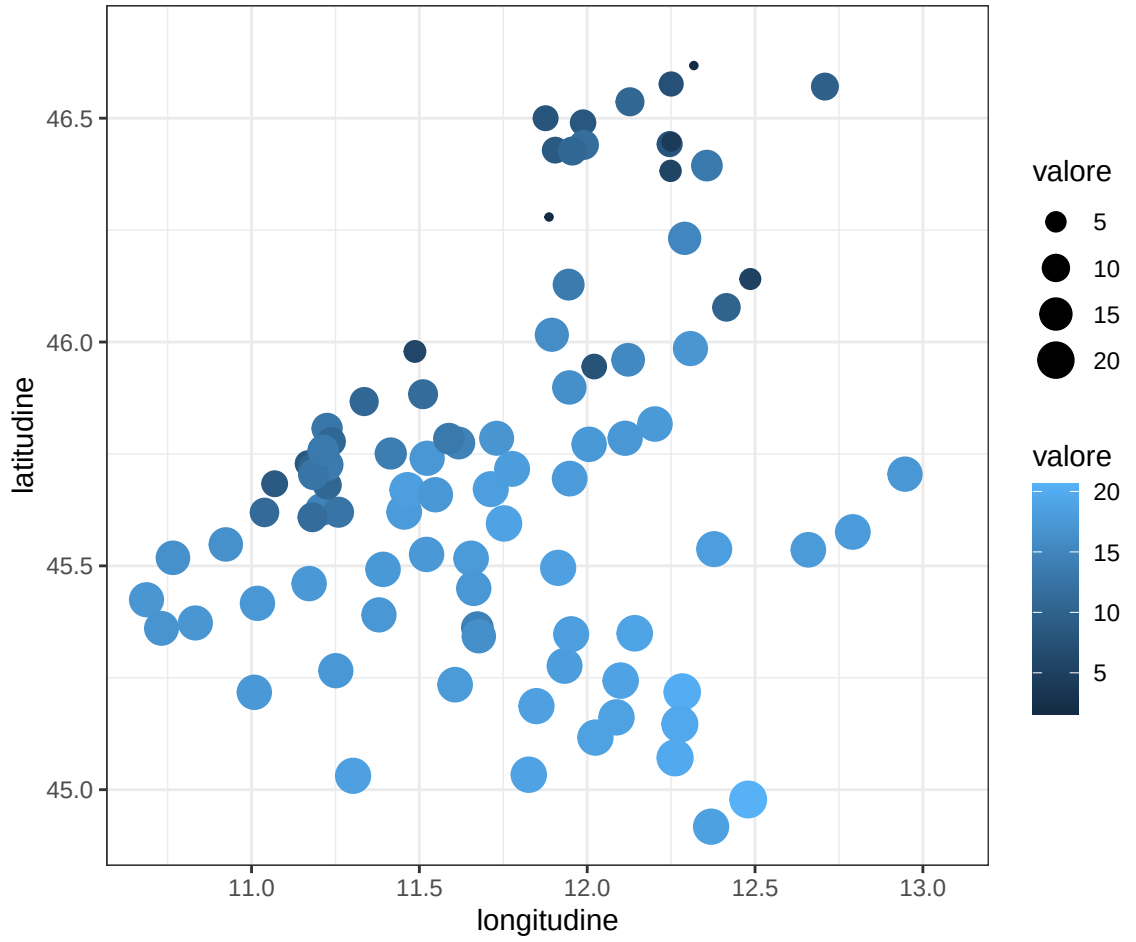
```
## Warning: Removed 56 rows containing non-finite outside the scale range
## (`stat_boxplot()`).
```



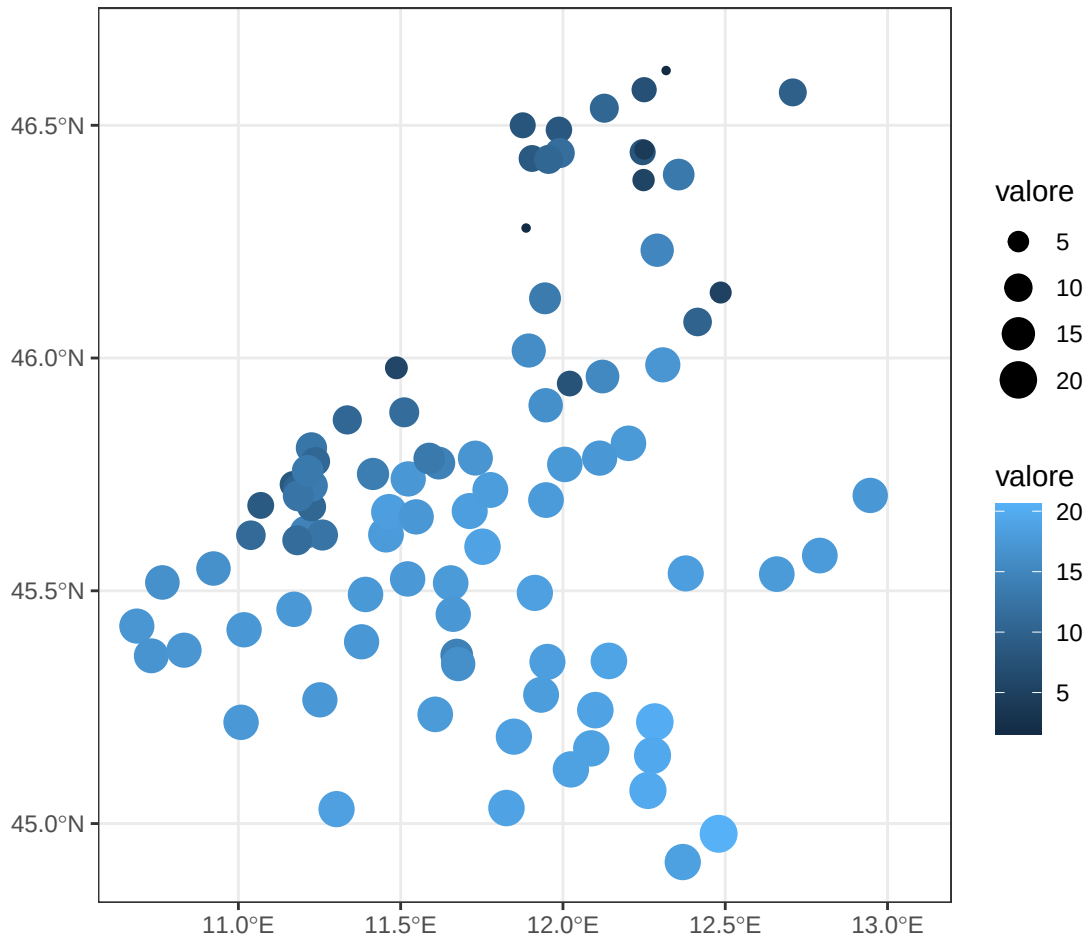
3.14 Grafici e dati geo

Abbiamo colonne con coordinate: possiamo creare un grafico con uno sfondo territoriale e considerare delle colonne come coordinate? Per dati geo usiamo libreria *sf* e *terra*. Non indispensabili per geostatistica, ma fortemente consigliati.

```
library(ggspatial, quietly = T)
library(sf)  ## se non l'avete installata non vi preoccupate
theme_set(theme_bw())
ggplot(dtf) + geom_point(aes(x=longitudine, y=latitudine, color=valore, size=valore ))
```




```
dtf.geo <- sf::st_as_sf(dtf,coords = c("longitudine","latitudine"), crs=4326 )
ggplot(dtf.geo) + geom_sf(aes(color=valore, size=valore )) + coord_sf()
```



4 Dati spaziali

4.1 Introduzione

4.2 Dati vettoriali