

Tommaso Russo, Giuliano Colosimo, Ciro Fanelli,
Simone Franceschini, Paolo Gratton, Antonio Parisi, Alice Romeo

UN VIAGGIO NELL'AMBIENTE R



TOR VERGATA
UNIVERSITY
PRESS

STEM

1

Tommaso Russo, Giuliano Colosimo, Ciro Fanelli, Simone
Franceschini, Paolo Gratton, Antonio Parisi, Alice Romeo

Un viaggio nell'ambiente *R*

Questo libro è stato realizzato grazie al supporto del Consorzio Interuniversitario per le Scienze del Mare (CoNISMa).

Il CoNISMa è stato costituito il 21 febbraio 1994 ed è attualmente formato da 36 Università italiane. Realizza progetti di ricerca, di formazione e altre attività scientifiche ed applicative nel campo delle Scienze del Mare. Il Consorzio è nato per favorire la collaborazione non solo tra le Università consorziate, ma anche tra Università, organismi pubblici e privati di ricerca nazionali e internazionali, Enti locali e territoriali, strutture produttive.

Gli ambiti dell'attività del CoNISMa vanno dalla ricerca di base (biodiversità, ecosistemi, sistema fisico, ambienti estremi, cambiamenti climatici), a quella applicata (pesca, acquacoltura, opere marittime) dallo sviluppo (biotecnologie, innovazione tecnologica strumentale), al trasferimento tecnologico (riproduzione e diete per specie ittiche, bioremediation, piantumazione di specie marine vegetali, produzione di energia dal mare), al supporto alle decisioni (gestione integrata della fascia costiera, marine strategy).

La versione digitale dell'opera è disponibile in modalità Open Access sul sito web tvupress.uniroma2.it, secondo i termini della licenza internazionale Attribution-ShareAlike 4.0 International (CC BY-SA 4.0).

Opera soggetta a *double-blind peer review*

DOI: 10.35948/TVUP/979-12-82347-17-4

ISBN 979-12-82347-17-4

Copyright © 2026 Authors

Copyright © 202 Tor Vergata University Press

Università degli Studi di Roma Tor Vergata

Via Cracovia, 50 - 00133 Roma

tvupress.uniroma2.it

Realizzazione editoriale di Edimill Srl I www.edimill.it

Indice

Prefazione	9
Gli autori	11
1. Le basi	13
Che cosa è l'ambiente <i>R</i>	13
A cosa serve/non serve <i>R</i>	18
“What You See Is What You Mean”: dallo <i>Script</i> al terminale (e viceversa)	19
Le Espressioni	23
Le Assegnazioni	26
Sintassi di base (I): l'uso delle parentesi	28
Tipologie di dati: gli attributi e le classi	34
Strutture di dati	38
Sintassi di base (II): le funzioni	48
Il mondo “a matrisoske”	57
La frontiera del calcolo: NA e NaN	59
Operatori relazionali e algebra di Boole	60
L'uso dell'Help	65
Una cassetta degli attrezzi essenziali	69
Funzioni per generare oggetti	70
Un po' di grafici di base	109
plot()	110
hist()	119
barplot()	121
boxplot()	122
plot()(2)	126
jpeg(), bmp(), png(), tiff() e pdf()	127
2. I modelli lineari	129
Un esempio (molto) stupido	130
Valore atteso	131
Residui	134
Likelihood	139
“Fittare” un modello	142

Confronto tra modelli	146
Modelli più o meno semplici	151
Likelihood ratio test	154
Predittori o non predittori?	158
Una <i>regressione lineare</i>	161
Un predittore che predice...	165
E le assunzioni?!	169
 3. Machine learning in <i>R</i>	 181
Introduzione	181
Supervised Learning	182
Unsupervised Learning	184
Dati	185
Divisione dei dati	186
Campionamento casuale semplice	187
Campionamento stratificato	188
Metodi di ricampionamento	190
Cross-validation	191
Bootstrapping	193
Stima della performance del modello	194
Modelli di regressione	195
Modelli di classificazione	196
Concludendo...	198
Algoritmi di Machine Learning in <i>R</i>	198
Decision Trees	199
Random Forests	202
 4. (Cenni di) Analisi genetiche in <i>R</i>	 207
Pacchetti da installare	210
Software di terze parti	212
Uccelli canori del genere <i>Sylvia</i>	213
mtDNA Gatto Dorato Asiatico	215
Microsatelliti giaguaro	216
Approccio descrittivo	218
Variabilità in marcatori altamente polimorfici	218
Variabilità aplotipica	221
Alberi filogenetici	226
 5. (Cenni di) Analisi di serie storiche in <i>R</i>	 233
Il <i>dataset</i>	238
Obiettivi della ricerca	241
Stimatore dei minimi quadrati	242
Previsioni	244

Previsioni puntuali	244
Intervalli di previsione	244
Verifica di ipotesi	245
Modelli con <i>trend</i>	246
Modello con <i>trend</i> lineare	246
Previsioni puntuali	250
Intervalli di previsione	251
Validazione delle previsioni	252
Diagnostiche del modello	254
Altri modelli di <i>trend</i>	256
Modelli stagionali	257
Regressione con variabili <i>dummy</i> stagionali	259
Previsioni	263
Diagnostiche del modello stagionale	265
Un modello più complesso	268
Diagnostiche	270
Modelli con cicli	271
Correlogramma	273
Il modello autoregressivo	275
Previsioni	277
Un modello con <i>trend</i> , stagionalità e cicli	279
Previsioni	281
Diagnostiche del modello	282
Selezione del modello	284
Conclusioni	285
Il modello <i>ADL</i>	286
6. Il mondo di <i>ggplot</i>	289
7. Analisi di dati spaziali e cenni di geostatistica	315
Introduzione e considerazioni generali	315
Dati geospaziali vettoriali	316
Dati geospaziali raster	317
Il concetto di <i>Coordinates Reference System (CRS)</i>	318
<i>EPSG (SRID)</i>	319
<i>proj4string</i>	320
La libreria <i>sf</i> per i dati vettoriali	320
La libreria <i>terra</i> per i dati raster	362
Un po' di Geostatistica	381
Metodi di interpolazione	390
Metodi deterministici	393
Metodi statistici: Il <i>Kriging</i> ordinario e quello universale	405

Questo manuale è dedicato alla mia amica e collega Laura Bruno, che nel 2014 mi chiese di farmi avanti come docente del corso di Metodi Informatici per la Biologia, e che mi ha sempre sostenuto e incoraggiato con la sua grande umanità, e allo storico Direttore del CoNISMa, l'ing. Annibale Cutrona (per gli amici Nini), per aver aiutato me e tanti altri giovani a diventare ricercatori adulti.

Tommaso Russo

Prefazione

R è un ambiente per la programmazione e l'analisi dei dati, sviluppato con particolare attenzione agli strumenti e ai metodi della statistica. È una piattaforma gratuita e aperta sostenuta da una comunità di utenti in tutto il mondo. L'utilizzo di *R* è molto diffuso nell'ambito della comunità scientifica, principalmente perché consente ai ricercatori di condividere rapidamente e in completa trasparenza procedure per l'analisi dei dati e prodotti digitali di vario tipo. Negli ultimi decenni è diventato una delle applicazioni di riferimento nell'ambito delle scienze teoriche ed applicate, e molti corsi di studio relativi a discipline scientifiche ne richiedono, ad un certo punto, l'apprendimento. Infatti, attraverso l'uso di *R* è possibile semplificare il proprio lavoro (rinunciando alla necessità di usare una varietà di altri software) e raffinare il proprio livello di competenza e autonomia rispetto all'analisi dei dati. Questo volume raccoglie il lavoro e le esperienze degli autori come docenti del Corso di **Metodi Informatici per la Biologia** e di alcuni cicli di lezioni sull'uso avanzato di *R* tenuti nell'ambito del programma didattico della **Scuola di Dottorato in Biologia Evoluzionistica ed Ecologia** dell'**Università degli Studi di Roma Tor Vergata**.

Sicuramente questo volume non rappresenta la prima risorsa, realizzata in lingua italiana, che punti a facilitare l'apprendimento e l'utilizzo dell'ambiente *R*. Una veloce ricerca nel web consente di trovare decine di predecessori, e altre iniziative analoghe seguiranno. Infatti, una delle tante

cose belle di *R* è che la sua **comunità**, in continua espansione, è inclusiva e coinvolgente... e questo perché molti utenti sono catturati dal fascino di *R*, che diventa uno dei loro principali strumenti di lavoro. Infatti, tra le ragioni principali che hanno spinto gli autori a intraprendere questa iniziativa c'è il fascino che questo strumento esercita su di loro e la grande importanza che riveste nelle attività di ricerca che portano avanti. Ma, come tanti altri ambienti di sviluppo e programmazione, *R* ha le sue peculiarità e imparare a usarlo sfruttandone appieno il suo potenziale richiede tempo e impegno.

Per spiegare meglio il senso di questo progetto è utile partire da quello che è già disponibile in rete (tralascieremo volutamente opere e materiali non disponibili liberamente online). A prima vista ci sono due principali tipologie di risorse:

- Dispense, manuali, e libri che forniscono una **guida di base** per i neofiti.
- Monografie tematiche e manuali avanzati che affrontano nel dettaglio argomenti più complessi come la logica grafica del pacchetto **ggplot2**.

Quest'opera non vuole essere nessuna delle due... ed entrambe. Ci piacerebbe introdurre all'uso di *R* chi è digiuno di concetti di programmazione, ma anche mostrare quanto lontano si può arrivare...

Il volume è articolato in due sezioni principali. La prima, rappresentata dal Capitolo 3 (Le Basi), vuole essere una introduzione ai rudimenti dell'ambiente *R* per chi non abbia nessuna esperienza di programmazione. Contiene una descrizione della sintassi di base e delle principali regole di programmazione in *R*. Gli altri capitoli compongono la seconda sezione, e sono più specifici: insieme a funzioni e metodi più avanzati, ciascun capitolo sviluppa un tema mediante una serie di esempi teorici o applicativi. Gli argomenti trattati comprendono i modelli lineari, l'analisi dei dati genetici e di quelli spaziali, l'analisi delle serie temporali, e la rappresentazione dei dati mediante la filosofia di ggplot2. Nessuno dei temi toccati nei vari capitoli è trattato in maniera esaustiva. Nei vari capitoli sono descritti solo alcuni dei tanti argomenti e metodi presenti in *R* e, sia nella parte di base che nei capitoli applicativi, gli autori hanno cercato un compromesso tra completezza del tema trattato e necessità di accompagnare i neofiti lungo un percorso di apprendimento progressivo e rapido. Molte descrizioni ed esempi sono presi da altre fonti liberamente accessibili online. In ciascuno di questi casi (salvo involontarie omissioni) è riportato il riferimento originale.

Gli autori

Tommaso Russo

È professore associato di Ecologia presso il Dipartimento di Biologia dell'Università degli Studi di Roma Tor Vergata ed è affiliato al CONiSMA. Si occupa di modellistica ecologica applicata alle scienze della pesca e studia gli impatti dell'uomo sugli ambienti marini e lo sfruttamento sostenibile delle risorse dei mari. Quello che sa di *R* lo ha imparato dagli altri... *in primis* da Antonio Parisi (vedi sotto)... ed ha lanciato l'iniziativa che ha portato a questo "manuale", grazie al fondamentale contributo (e alla pazienza) degli altri autori. Ha scritto i capitoli 1. *Le basi*, 6. *Il mondo di ggplot* e *Analisi di dati spaziali e cenni di geostatistica*.

Giuliano Colosimo

È uno zoologo, ricercatore presso il Dipartimento di Biologia dell'Università di Roma Tor Vergata. Si occupa di genetica ed ecologia orientate alla conservazione di specie, rettili in particolare, a rischio d'estinzione. Ha scritto il capitolo 4. *(Cenni di) Analisi genetiche in R* ed è un classico esempio di self learning, avendo imparato ad usare *R* spendendo lunghe ore davanti al computer.

Ciro Fanelli

È pittore, illustratore, tatuatore e scrittore. I suoi lavori sono apparsi su «La Lettura – Corriere della Sera», «Vice», «Esquire» e altre prestigiose riviste europee e giapponesi. Ha pubblicato *Pinocchio* e *Les corbeaux pleurent la merde* per Le Dernier Cri (Marsiglia), per Rizzoli *Lizard* e *Nel bosco del nostro splendore*. È l'autore di tutte le figure di questo libro (tranne quelle generate in *R*), che ha realizzato attraverso un costante dialogo con gli altri autori, provando a interpretare e visualizzare i concetti di programmazione e statistica spiegati nel testo.

Simone Franceschini

È un ricercatore che, come ecologo, si è focalizzato sull'applicazione di metodi statistici e computazionali negli ambienti di acqua dolce e marini, specializzandosi sull'analisi spaziale e lo sviluppo di algoritmi

di machine learning. Si è formato presso il Laboratorio di Ecologia Sperimentale ed Acquacoltura dell'Università degli Studi di Roma Tor Vergata sullo studio delle dinamiche di popolazioni marine e gli impatti antropici sugli ecosistemi del mare. Negli ultimi anni ha studiato le comunità di pesci d'acqua dolce nella foresta amazzonica utilizzando modelli di deep learning. Attualmente lavora come ricercatore presso l'Hawaii Institute of Marine Biology dell'University of Hawaii a Manoa. Ha scritto il capitolo 3. *Machine learning in R*.

Paolo Gratton

È professore associato di Zoologia presso il Dipartimento di Biologia dell'Università degli Studi di Roma Tor Vergata. Si è occupato di biogeografia, del problema della delimitazione delle specie, e di misure del comportamento, in animali molto diversi come farfalle, trote e scimpanzé. Ultimamente sta provando a reinventarsi anche come ornitologo ed esperto di anfibi. Ha scoperto la statistica relativamente da poco (vabbè... una decina d'anni...), ma non riesce più a ricordare come si possa pensare a un problema senza chiedersi con quale modello rappresentarlo. Ha scritto il capitolo 2. *I modelli lineari*.

Antonio Parisi

È un ricercatore presso il Dipartimento di Economia e Finanza dell'Università degli Studi di Roma Tor Vergata, ma è prima di tutto un esperto di statistica che si diverte da tanti anni a collaborare con gli ecologi (usando *R*, ovviamente!). È autore del capitolo 5. *(Cenni di) Analisi di serie storiche in R*.

Alice Romeo

È una ricercatrice in Bioinformatica presso il Dipartimento di Biologia dell'Università degli Studi di Roma Tor Vergata. Si è avvicinata al mondo di *R* da un paio di anni ed è entrata in gioco nelle fasi finali della stesura del libro, occupandosi di revisionare e armonizzare il codice e i contenuti.

1



Le basi

Che cosa è l'ambiente *R*

In questa sezione saranno introdotte alcune nozioni generali che, spesso, sono date per scontate e quindi non chiaramente delineate. Nella nostra esperienza, dare per scontati questi concetti rende ostico il primo impatto con *R* e le prime fasi del suo utilizzo, cioè quelle durante le quali i nuovi utenti devono assimilare la sintassi, le diverse tipologie di dati e le strutture all'interno delle quali i dati possono essere organizzati e gestiti. Come tutti gli ambienti e i linguaggi di programmazione, *R* ha una sua rigida sintassi e, sebbene essa non sia particolarmente complessa e articolata, sicuramente rappresenta il primo grande ostacolo a un utilizzo rapido ed efficace. Al contrario, se resi comprensibili e chiari, i capisaldi della sintassi possono essere dei punti di riferimento quando guardiamo un codice scritto da altri o da noi stessi un po' di tempo fa... E dunque la prima parte di questo manuale è concepita per illustrare le regole generali che sono alla base

delle procedure nell'ambiente *R*. Dunque, impareremo la differenza sostanziale tra un'**espressione** e un'**assegnazione**, l'utilizzo delle **parentesi** e le diverse tipologie di **oggetti** di base. Successivamente, impareremo a comporre dei semplici codici combinando espressioni, assegnazioni, e oggetti. Nel farlo, cercheremo subito di introdurre dei **suggerimenti di stile** e delle **buone pratiche** che ci aiuteranno a essere più ordinati, efficienti ed efficaci.

R è un ambiente per *Object-oriented programming*, cioè uno stile di programmazione diventato molto popolare negli ultimi anni. Questo stile si basa sull'utilizzo (la manipolazione mediante comandi) di **oggetti**, un approccio che rende molto semplice accedere (leggere e scrivere) ai dati in memoria e combinarli in strutture che possono essere anche molto complesse. Aspetti fondamentali dei linguaggi *object-oriented* sono il concetto di **classe** e quello di **metodo**, sui quali torneremo in seguito in maniera approfondita. **Classi** e **metodi** sono legati dal fatto che, in *R*, la computazione si basa sull'applicazione di specifici metodi a oggetti di specifiche classi. Non esiste una corrispondenza biunivoca tra classi e metodi, piuttosto molti metodi possono essere applicati a oggetti di classi diverse e viceversa. Tuttavia, metodi diversi sono generalmente specializzati per essere applicati su specifiche classi di oggetti. Questa associazione metodi-classi è spesso molto intuitiva, ma esistono funzioni generiche per determinare di che classe sia un dato oggetto e quali siano i metodi su di esso applicabili.

Un altro aspetto importante dei linguaggi *object-oriented* è quello di *inheritance* (eredità). Questo aspetto può essere sintetizzato come la capacità degli oggetti generati durante l'esecuzione di un codice o di una procedura di calcolo di "portarsi appresso" le caratteristiche degli oggetti "genitori" o delle funzioni "genitrici", cioè quelli a partire dai quali sono stati creati. Un esempio intuitivo è il seguente: se definisco due quantità numeriche *a* e *b* e le sommo, quello che ottengo (*c*) erediterà le caratteristiche dei genitori (numerici entrambi) e potrà essere trattato come tale. Questo aspetto semplifica enormemente il lavoro poiché, a differenza di quanto avviene in linguaggi di base, non è necessario pre-definire l'oggetto *c* indicandone esplicitamente le caratteristiche e poi riempirlo.

Continuando l'elenco dei grandi vantaggi di *R*, dobbiamo sicuramente menzionare la sua enorme *Community*. Secondo la compagnia di analisi del software TIOBE, l'ambiente *R* è attualmente (febbraio 2025) al 15° posto tra i linguaggi più popolari, con una crescita in positivo.

```
TIOBE <- data.frame(Name =
c("Python", "C++", "Java", "C",
  "C#", "JavaScript",
  "SQL", "Go", "Delphi/Object Pascal", "Visual
  Basic",
  Fortran", "Scratch", "Rust",
  "PHP", "R", "MATLAB",
  "Assembly language", "COBOL",
  "Ruby", "Prolog"),
  Ratings = c(23.88, 11.37, 10.66, 9.84, 4.12,
    3.78, 2.87, 2.26, 2.18, 2.04, 1.75,
    1.54, 1.47, 1.14, 1.06, 0.98, 0.95,
    0.82, 0.82, 0.80),
  Change = c(8.72, 0.84, 1.79, -1.14, -3.41,
    0.61, 1.04,
    0.53, 0.78, 0.52, 0.35, 0.36, 0.42,
    -0.37, 0.07,
    -0.28, -0.24, -0.18, -0.17, 0.03))
par(las = 2, mar = c(5,10,1,1), cex = 0.7)
barplot(TIOBE$Ratings[order(TIOBE$Ratings, decreasing = F)],
  names.arg = TIOBE$Name[order(TIOBE$Ratings, decreasing = F)],
  xlab = "Ratings (%)", ylab = "", xlim = c(0,35),
  col = c(rep("grey",5), "blue", rep("grey",19)), horiz = T)
y_coords <- seq(0.7, by = 1.2, length.out = length(TIOBE$Change))
x_start <- TIOBE$Ratings[order(TIOBE$Ratings, decreasing = FALSE)]
x_end <- x_start + TIOBE$Change[order(TIOBE$Ratings, decreasing =
FALSE)]
arrows(x0 = x_start, y0 = y_coords, x1 = x_end, y1 = y_coords,
  col = ifelse(TIOBE$Change[order(TIOBE$Ratings, decreasing =
FALSE)] > 0,
    "black", "black"), length = 0.08, angle = 25,
  lwd = 1.5)
```

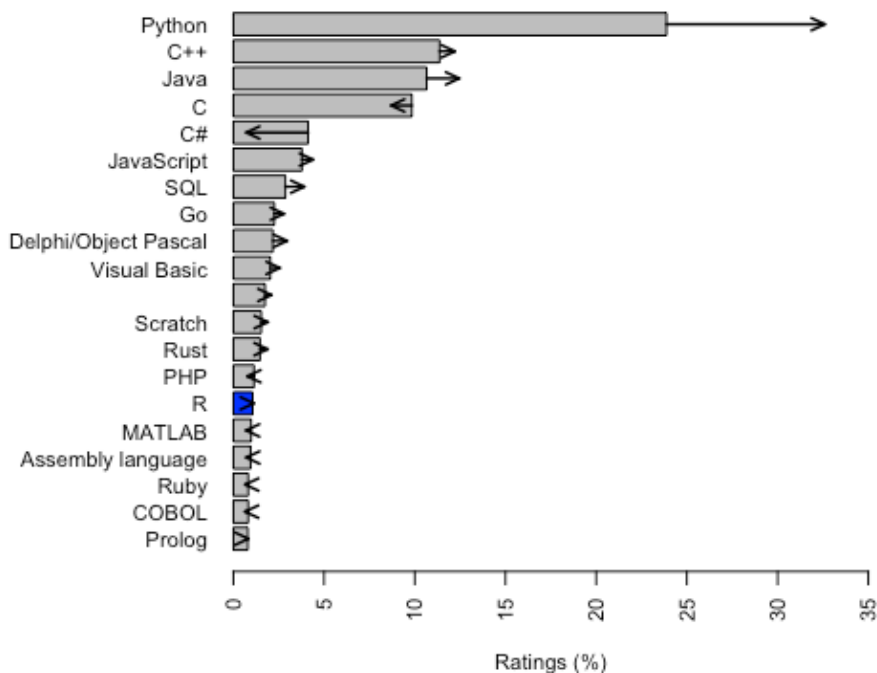


Figura 1 - Grafico a barre che rappresenta il Rating (% di diffusione nella comunità di utenti) dei principali ambienti di programmazione, aggiornato a febbraio 2025. La direzione delle frecce indica se la crescita è stata in positivo o in negativo nell'ultimo anno.

Questo dato è ancora più significativo se si considera che *R* non è un ambiente per produrre software compilato e quindi eseguibile (come avviene invece con i membri della famiglia **C**). Perché questo aspetto è così importante? Semplicemente perché, essendo *R* un ambiente “libero”, ogni utente può attingere a una sterminata banca dati di informazioni, algoritmi, pacchetti aggiuntivi e funzioni già sviluppate da altri utenti... e spesso può contare sull'aiuto diretto di una vasta *community*. In ambito scientifico, *R* ha avuto (e continua ad avere) una diffusione inarrestabile perché incarna perfettamente alcuni ideali della ricerca: la possibilità di sviluppare metodologie completamente replicabili, chiare e ulteriormente implementabili.

Come ha giustamente argomentato qui Shannon Ellis, una *community* ampia offre sviluppatori, esperti, potenziali collaboratori, estensori di documentazione, persone che mettono alla prova le nostre routine, “agitatori” (che sono sempre utili per tenere viva la *community*) o sem-

plicemente persone gentili che sono disposte ad aiutare i neofiti. Ma, soprattutto, qualsiasi sia la sfida con la quale siete alle prese, molto probabilmente qualcuno ci si è già cimentato e potrete ripartire dal suo lavoro. Non è forse, questo, un enorme vantaggio? Shannon Ellis ha compilato una lista di siti/blog che rappresentano i principali canali web della *community* di *R*, come ad esempio:

- **Rweekly** - un magazine settimanale di quello che succede nel mondo di *R*.
- **R-bloggers** - un archivio di post utili fatti tra utenti di *R*.
- **Stack Overflow** - dove cercare risposte alle nostre domande, spesso legate alle difficoltà dei neofiti (ma non solo).

Una bella e completa descrizione di cosa sia *R* la trovate *qui*. In sintesi:

- *R* è un ambiente integrato e coerente per l'analisi statistica.
- *R* è contemporaneamente un linguaggio ed un software.
- *R* è un potente strumento per la produzione di grafici (originali).
- *R* è distribuito gratuitamente sotto i vincoli della **GPL - General Public License** ed è disponibile per diverse architetture hardware e sistemi operativi: Unix, Linux, Windows, MacOS.
- *R* è disponibile presso il sito **The R Project for Statistical Computing** dove è possibile scaricare, oltre che il programma base, anche una serie di moduli aggiuntivi e un'ampia manualistica (in lingua inglese) sull'ambiente, che va dall'installazione del software al suo utilizzo nell'analisi dei dati.
- il sorgente di *R* è scritto in parte nel linguaggio C ed in parte nel linguaggio *Fortran*.

A cosa serve/non serve R

R non è una panacea per affrontare qualsiasi sfida scientifica, e anche gli appassionati di questo strumento devono essere a conoscenza dei suoi difetti quanto dei suoi pregi. Da una parte, R si comporta come gli smartphone che, dalla sua comparsa sul mercato, hanno completamente “assorbito” e messo fuori dal mercato altri *device* come i lettori mp3, le fotocamere, i navigatori satellitari. Infatti, chi usa R può facilmente rinunciare a fogli di calcolo, software proprietari per la statistica, gestori di database, piattaforme per la geostatistica, software per l'analisi e la manipolazione di immagini, e la lista potrebbe continuare. R può servire praticamente a tutto!

Un altro grande vantaggio di R è che, essendo un software *open source*, si può accedere al codice sorgente e modificarlo. R, inoltre, è del tutto gratuito, e possiede un'ampia biblioteca di manuali e dispense accessibili gratuitamente in rete, oltre a pubblicazioni a pagamento.

L'installazione base di R può essere integrata con librerie di funzioni aggiuntive (il vero valore aggiunto per il mondo della ricerca) che ne ampliano a dismisura le capacità. R offre un supporto tramite l'*R Development Core Team* ed infine, R sa “parlare” con altri linguaggi, da C a SQL a Fortran e Python, per cui si possono integrare metodi e funzioni sviluppate in altri contesti.

Per essere proprio completi, è importante ricordare che:

- R ha una quantità veramente incredibile di funzioni grafiche, che consentono di ottenere grafici estremamente efficaci dal punto di vista scientifico ma anche “belli”.
 - R permette di essere creativi ed efficienti, poiché consente di scrivere nuove funzioni o intere librerie in maniera molto semplice.
 - La sua versione base è molto snella, ed essendo sviluppato secondo una logica vettoriale lavora molto bene su vettori e matrici.
- E le brutte notizie? Beh...
- R è “brutto”: la sua interfaccia non è pensata per essere bella e accattivante come gli OS ai quali siamo abituati. L'interfaccia utente a carattere (CUI) di R è anche “ostica”, poiché ci costringe a cambiare modo di lavorare. È vero che è presente anche un'interfaccia GUI (*Graphical User Interface*), ma gli utenti esperti non la usano quasi mai.

- *R* è un “pachiderma”: può arrivare a occupare una grande quantità di memoria e la sua velocità di calcolo è di gran lunga inferiore a quelle di ambienti e linguaggi di livello più basso (come la famiglia C o *Visual Basic*). Questo, a volte, costringe gli utenti più esperti a compilare delle funzioni in C tutte le volte che le loro analisi diventano *computationally and time-demanding*.
- Non tutte le funzioni di *R* sono ottimizzate per avere elevate prestazioni (un buon esempio sono le funzioni di ottimizzazione numerica). A volte sarete costretti a metterci del vostro per sviluppare funzioni nuove o più performanti.
- Se sviluppate un’applicazione basata su *R*, è vostro onere tutto quello che riguarda il *debug* e il mantenimento: se salta fuori un problema, non potete prendervela con nessun altro!

“What You See Is What You Mean”: dallo *Script* al terminale (e viceversa)

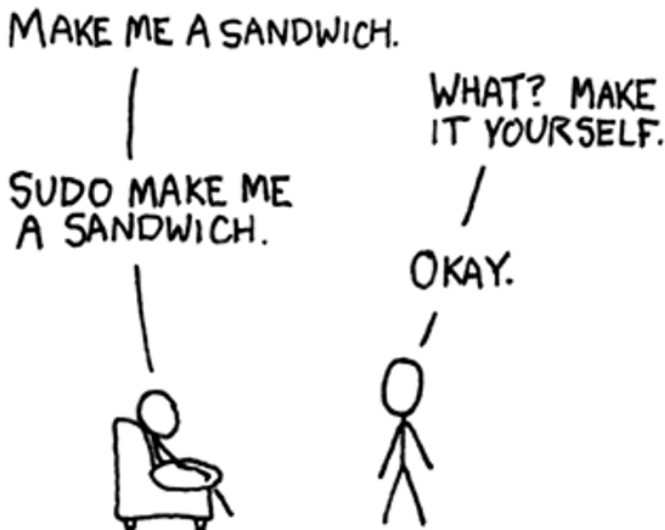


Figura 2 - Il terminale ci costringe a usare una precisa sintassi. Se lo facciamo, *R* eseguirà prontamente i nostri comandi, altrimenti si rifiuterà di farlo – Fonte: <https://xkcd.com/>.

Per affrontare la prima parte, dedicata ad una carrellata degli aspetti base (concettuali e di sintassi) dell’ambiente *R*, è necessario assimilare alcuni termini (e strumenti).

Il **terminale** (detto *console* in inglese, così come è denominata la sua finestra nell'interfaccia di *RStudio*), è l'**interfaccia** attraverso il quale l'utente interagisce con *R*. In informatica, un'interfaccia a riga di comando o console è un tipo di interfaccia utente caratterizzata da un'interazione testuale tra utente ed elaboratore (vedi *shell*): l'utente impartisce comandi testuali in input mediante tastiera alfanumerica e riceve risposte testuali in output dall'elaboratore mediante display o stampante alfanumerici. Nei sistemi operativi moderni usati nei personal computer, al posto della cli, in buona parte, c'è l'interfaccia grafica con caratteristiche tipicamente *user friendly*.

Il primo (e forse principale) trauma al quale il neofita di *R* si trova di fronte è la necessità di rinunciare (possibilmente da subito e definitivamente) all'interazione tramite il mouse, e quindi alle interfacce a finestre e menù a tendina che tanto ci sono familiari e rispetto alle quali abbiamo sviluppato una certa dipendenza! Quando si comincia ad usare *R*, bisogna accettare di inserire i comandi digitandoli direttamente nel terminale o predisponendoli in uno *script* (una sequenza di comandi annotata da qualche parte, generalmente in un editor di testo o in una apposita finestra di *RStudio*). Ci sono due motivi principali per questo:

- Sarebbe impossibile codificare tutti i possibili comandi di *R* in un menù grafico o a tendina, senza considerare che questo limiterebbe comunque l'originalità degli utenti e renderebbe assai complesso combinare insieme varie funzioni.
- Imparare a usare il terminale rende il lavoro estremamente più veloce (provare per credere). Questo secondo aspetto accomuna *R* a tanti altri ambienti di alto livello e persino a sistemi operativi come Ubuntu e Debian (due delle principali distribuzioni di Linux).

Per utilizzare il terminale, però, non basta conoscere la sintassi dei comandi: bisogna cambiare filosofia. Quando utilizziamo un elaboratore di testo, digitiamo i caratteri sapendo che esiste una precisa corrispondenza tra i tasti che premiamo ed il risultato che apparirà sullo schermo. Questo modo di operare viene denominato *What You See Is What You Get* («quello che vedi è quello che è» o «ottiene quanto vedi» – acronimo *WYSIWYG*).

Utilizzare il terminale di un ambiente come *R* significa invece applicare la filosofia *What You See Is What You Mean* (letteralmente: «quel che vedi è ciò che intendi», o sinteticamente, «vedi quanto intendi» –

acronimo *WYSIWYM*). In sostanza, le istruzioni che digitiamo non sono semplici parole sullo schermo ma implicano l'applicazione di funzioni anche complesse che, nelle intenzioni dell'utente, devono restituire un risultato di qualche tipo.

L'utilizzo di *R* si svolge tipicamente attraverso un continuo passaggio dall'editor di testo, dove si modifica lo *script*, al terminale dove lo si esegue. Se poi si usano interfacce avanzate come *RStudio*, che vedremo più avanti, si può disporre di utili finestre accessorie (come quella dei grafici). Ecco una tipica schermata di *RStudio*.

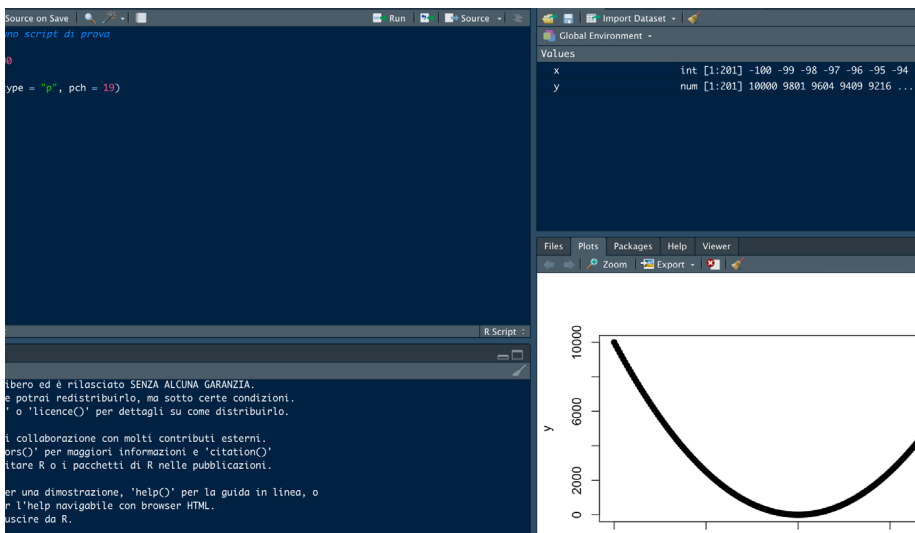


Figura 3 - Aspetto tipico dell'interfaccia di *RStudio*, il più diffuso strumento per lavorare con *R*.

L'interfaccia di *Rstudio* contiene 4 finestre che sono spesso organizzate in questo modo (ma l'utente ha ampie possibilità di personalizzarle):

- In alto a sinistra c'è lo **Script** (o gli *Script* se si vogliono gestire più procedure separate). Questa finestra è semplicemente un blocco note multi-pagina, che consente di inviare i comandi direttamente al **terminale**.
- In basso a sinistra ecco proprio lui, il terminale, ovvero il canale di collegamento in ingresso da noi verso *R*. In *Rstudio*, questa finestra si chiama **Console**.

- In alto a destra c'è una finestra “composita” che racchiude **Environment**, all'interno del quale troveremo un elenco (con alcune caratteristiche essenziali) degli oggetti nella memoria della sessione, **History**, ovvero una cronologia dei comandi passati alla **Console**, ed infine le tre sotto-finestre **Connections**, **Build** e **Tutorial** sulle quali, per ora, possiamo sorvolare.
- In basso a destra c'è un'altra finestra “composita” che racchiude **Files**, cioè l'elenco dei file nella cartella di lavoro (la *directory* del pc) scelta per la sessione, **Plots**, che conterrà le anteprime dei grafici realizzati, **Packages**, che è un prontuario di tutti gli help dei pacchetti installati, **Help**, l'interfaccia minimale di aiuto che presto impareremo ad apprezzare, e **Viewer**, destinata a mostrare contenuto html, come le applicazioni Shiny.

Una tipica sessione di lavoro in *R* con *Rstudio* prevede un utilizzo semi-simultaneo di questi pannelli, e principalmente dello Script e del **Terminale** o **Console**. Dopo queste premesse siamo pronti per rompere gli indugi e fare conoscenza con *R*. Qualsiasi sia la vostra installazione, aprite il terminale e cominciamo.

Le Espressioni

R è, prima di tutto, un ambiente per calcolo. In questo senso, possiamo vedere il terminale come una calcolatrice. È possibile eseguire calcoli semplici o complessi direttamente mediante il terminale, digitando delle istruzioni analoghe alle formule algebriche, cioè utilizzando un insieme di quantità numeriche, simboli e operatori matematici, e poi premendo il tasto *INVIO*.

Una cosa che possiamo dire subito è che, in R , l'uso degli spazi è (quasi sempre) arbitrario. Ad esempio, queste due espressioni sono equivalenti:

```
2 + 5
## [1] 7
2+5
## [1] 7
```

Come avviene per altri ambienti e linguaggi di programmazione, gli operatori matematici di base sono:

- La **somma**, codificata col simbolo **+**

```
2 + 5
## [1] 7
```

- La **sottrazione**, codificata col simbolo **-**

```
2 - 5
## [1] -3
```

- La **moltiplicazione**, codificata col simbolo *****

```
2 * 5
## [1] 10
```

- La **divisione**, codificata col simbolo **/**

```
2 / 5
## [1] 0.4
```

- L'**elevamento a potenza**, codificato col simbolo $\wedge$ (ad esempio x^2 indica x elevato al quadrato)

```
2 ^ 5
## [1] 32
```

Esistono poi altre simbologie con un significato matematico definito, come ad esempio:

- L'operatore ":" che serve a generare una sequenza ordinata di interi tra un minimo ed un massimo definiti

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

- Il simbolo "%*%" permette di operare la moltiplicazione tra vettori e matrici

```
3:7 %*% 1:5
##      [,1]
## [1,]    85
```

All'interno di R, l'utilizzo delle varie tipologie di **parentesi** è legato a funzionalità diverse (vedi più avanti). Le parentesi tonde sono, di fatto, l'unica tipologia di parentesi che mantiene (anche) un significato matematico. Ad esempio:

```
2 * (2 + 3)
## [1] 10
```

Questo implica che, se esistono espressioni annidate, dovremo utilizzare sempre e solo le parentesi tonde (invece che passare alle quadre e poi alle graffe come ci hanno insegnato a scuola):

```
((2 + 1) * (2 + 3))/2
## [1] 7.5
```

È ovvio che l'utilizzo delle parentesi deve essere sempre "simmetrico" (una parentesi apre e una chiude) e coerente (i conti devono tornare e

quindi ci devono essere tante parentesi aperte quanto quelle chiuse). Quando le espressioni diventano molto complesse, quindi, diventa complicato capire, a colpo d'occhio, quali sono le coppie di parentesi. Ad esempio:

```
(( (2 + 1) * (2 + 3)) / (2 * (4 / 2) + 1)) ^ (2 + 3)
## [1] 243
```

Comunque i più comuni editor per gli *script*, e quindi anche software più complessi e integrati come *RStudio*, ci vengono in soccorso evidenziando le coppie di parentesi quando ci spostiamo lungo una espressione con le frecce destra e sinistra della tastiera (provare per credere!). Le frecce in alto e in basso, invece, possono essere usate per scorrere nella cronologia dei comandi che abbiamo inserito tramite il terminale. Questo ci aiuta a non dover ri-digitare daccapo un'istruzione usata poco prima.

Dunque, possiamo digitare le espressioni attraverso il terminale che, abbiamo già imparato, ci appare come il simbolo `>`. Tuttavia, quando digitiamo un'espressione matematicamente incompleta (nella quale, ad esempio, manca una parentesi o un argomento legato ad un operatore), *R* non eseguirà il calcolo ma ci risponderà con l'operatore `+` che serve a farci capire che "manca qualcosa". È possibile (ma è preferibile non farlo per non pasticciare il codice) inserire più espressioni, separate da `;` o `.` nella stessa linea di comando (prima di premere *INVIO*).

In ogni caso, le espressioni digitate direttamente nel terminale non vengono salvate in memoria e non possono essere usate per produrre risultati molto complessi. Tutte le volte che sono necessari più passaggi per arrivare dai dati di partenza ai risultati finali è necessario "prendere appunti", cioè generare delle quantità intermedie e associare ad esse dei nomi. Questo è esattamente quello in cui consiste l'**assegnazione**.

Le Assegnazioni



Figura 4 - Il concetto di assegnazione ricorda la formale “investitura” di un cavaliere da parte del suo Re.

Il risultato di una semplice espressione, come gli esempi del paragrafo precedente, viene stampato sullo schermo, cioè appare come risposta di *R* nel terminale, ma non viene memorizzato. Per generare una costante, una variabile o un qualsiasi oggetto che contenga uno o più valori (che magari sono il risultato di espressioni matematiche) è necessario **assegnare** all'espressione un nome. Questo è possibile farlo attraverso gli operatori `=` e `<-`. Ad esempio:

```
a = 4
```

```
a <- 4
```

Questi due operatori hanno lo stesso significato e producono il medesimo effetto: l'oggetto `a` sarà associato alla quantità 4. Perché questa ridondanza (due simboli per la stessa operazione)? In realtà, `=` e `<-` non si comportano esattamente allo stesso modo. Per cogliere la differenza, proviamo a digitare la seguente espressione, basata sull'utilizzo della funzione `sum`:

```
sum(a = 4, b = 5)
## [1] 9
```

Dopo che *R* ci avrà restituito il risultato 9, proviamo a “chiederagli” se ricorda quanto valgono *a* e *b*, ad esempio digitando *b* nel terminale e premendo *INVIO*.

```
b
## Error: object 'b' not found
```

Scopriremo che *R* non ricorda quanto valga *b*!
Se invece provassimo a digitare l'espressione:

```
sum(a <- 4, b <- 5)
## [1] 9
```

e poi chiedessimo a *R* quanto vale *b* (oppure *a*), scopriremmo che *R* ha memorizzato i loro valori:

```
b
## [1] 5
```

Il perché di questa apparente stranezza sta nel fatto che l'ambiente di calcolo di *R* non è unico, bensì ogni funzione opera in un suo “spazio” secondario e solo le quantità assegnate con *<-* trascendono dal sottospazio definito dalla funzione *sum* allo spazio generale attraverso il quale dialoghiamo.

Dunque, a seconda che vogliamo assegnare delle quantità “generalì” o “localì” sceglieremo la simbologia corrispondente, tenendo presente che esistono delle “buone pratiche” che vedremo più avanti. Ovviamente una differenza sostanziale tra *=* e *<-* sta anche nella complessità (e quindi nella velocità) delle due scritture: nel secondo caso, dovremo usare una combinazione due tasti (*< + -*) invece che uno, e quindi l'assegnazione “generale” è più “faticosa”.

Un'ultima precisazione: le assegnazioni possono essere scritte anche da sinistra verso destra. Ad esempio le scritture:

```
a <- 4  
4 -> a
```

sono funzionalmente equivalenti. Tuttavia si sconsiglia la seconda poiché, come chiunque può sperimentare, è molto più chiaro un algoritmo in cui gli oggetti generati nei diversi passaggi compaiono subito all'inizio di ogni riga.

Sintassi di base (I): l'uso delle parentesi

Le parentesi sono dei simboli tipografici ampiamente usate sia nella lingua scritta che nella matematica. Esistono 3 tipi di parentesi, e per ognuno di essi esiste una versione di apertura delle parentesi e una di chiusura: la prima è generalmente un'immagine dotata di convessità verso sinistra, mentre la seconda la possiede generalmente a destra. Nell'ambiente *R*, le parentesi sono delle strutture molto importanti usate per finalità ben precise. Per questo è fondamentale imparare ad usarle in modo appropriato ed imparare a riconoscerle all'interno di uno *script*, in modo da orientarsi in procedure complicate o istruzioni annidate.

All'interno di *R*, ognuna delle tre diverse tipologie di parentesi (tonde, quadre, graffe) è destinata a uno scopo preciso.

Le parentesi tonde



Figura 5 - Le parentesi tonde, spesso annidate, formano una virtuale catena di funzioni che si "mangiano" tra loro.

Come abbiamo già visto, le parentesi tonde servono per garantire l'ordine desiderato delle operazioni. Oltre a questo, servono per specificare gli argomenti (ed eventualmente le opzioni) di una **funzione**. In sostanza, le parentesi tonde delimitano un micro-ambiente, parzialmente indipendente da quello generale, all'interno del quale entrano alcuni oggetti o valori (gli **argomenti**) e dal quale fuoriescono altri oggetti o valori (i **risultati**).

Può essere utile, a questo punto, fare un piccolo antipasto di come si usano le funzioni in *R* (torneremo estesamente più avanti sull'argomento). L'esempio seguente ci mostra il risultato dell'utilizzo della funzione *c*, una delle più semplici tra quelle di base:

```
c(1, 2, 3, 5)
## [1] 1 2 3 5
```

Il nome *c* sta per *concatenate* (concatena), e difatti questa funzione mette insieme più *argomenti*, separati da virgole, in un singolo oggetto (in questo caso un vettore composto da 4 numeri). **Le parentesi tonde seguono sempre il nome della funzione e racchiudono gli argomenti (e le opzioni, se ci sono) separati da virgole.** In questo caso non abbiamo effettuato nessuna assegnazione, e il terminale di *R* esegue il comando restituendoci il vettore desiderato. Un altro semplice esempio è:

```
a <- c(1, 2, 3, 5)
sum(a)
## [1] 11
```

In questo caso, abbiamo generato un vettore denominato *a* e poi abbiamo applicato la funzione *sum* (somma) su di esso, ottenendo il risultato corrispondente alla somma degli elementi di *a*. Gli argomenti di una funzione, dunque, sono i dati in ingresso, la materia prima per permettere alla funzione di lavorare e restituirci un risultato.

Le parentesi quadre

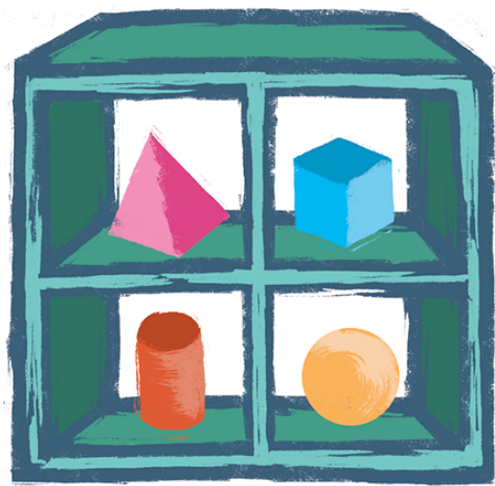


Figura 6 - Le parentesi quadre ci consentono di accedere alla struttura interna degli oggetti nel nostro spazio di lavoro.

Servono per **indicizzare** gli oggetti, cioè per identificare al loro interno delle sotto-parti precise. Come all'interno di una griglia spaziale (immaginate di giocare una partita di battaglia navale), le parentesi quadre ci permettono di fare riferimento a pezzi specifici di oggetti complessi, permettendoci di manipolarli (estrarne delle parti, sostituirle, modificarle, ecc.).

Anche in questo caso si possono fare degli esempi preliminari.

```
valori <- c(10, 15, 23, 5)
valori [1]
## [1] 10
```

La scrittura precedente definisce un nuovo vettore `a` e poi, mediante l'uso delle parentesi quadre, ne identifica e isola il primo valore (in questo caso il numero 10). Naturalmente potremmo isolare, con la stessa logica, un altro elemento di `a`, per esempio il quarto.

```
valori [4]
## [1] 5
```

Dovrebbe essere chiaro, quindi, che il numero che compare tra le parentesi quadre corrisponde alla posizione dell'elemento desiderato, e non al suo valore. Siccome un vettore di numeri è, per definizione, un oggetto unidimensionale, è sufficiente un singolo numero (la "posizione nella fila") per indicare un certo valore in quella posizione (come nella coda alle poste).

Ma cosa accade quando abbiamo oggetti con più di una dimensione? Per comprenderlo introduciamo le matrici, che sono oggetti a due dimensioni (righe e colonne). Vediamo un esempio:

```
m <- matrix(101:109, 3, 3)
m
##      [,1] [,2] [,3]
## [1,] 101 104 107
## [2,] 102 105 108
## [3,] 103 106 109
m[1,1]
## [1] 101
```

Sorvoliamo sull'analisi degli argomenti della funzione `matrix` (i lettori più scaltri avranno intuito che la funzione genera una matrice mettendo i valori da 101 a 109 in 3 righe e 3 colonne, come si vede dalla struttura che *R* stampa sullo schermo quando invochiamo `m`), e vediamo come, in questo caso, all'interno delle parentesi quadre abbiamo messo 2 numeri, 1 e 1, separati da una virgola. *R* ci dice che `m[1,1]` corrisponde al numero 101, quello collocato nella prima riga e prima colonna della matrice.

Allo stesso modo potremmo identificare il numero 104 come quello nella prima riga e seconda colonna.

```
m[1,2]
## [1] 104
```

Insomma i due valori separati da virgola tra le quadre sono niente altro che le coordinate dei valori desiderati. Nel caso di un oggetto bidimensionale come una matrice, per indicare un elemento si deve ragionare come in una partita di battaglia navale!

Naturalmente possiamo voler isolare più di un elemento di un oggetto complesso, ad esempio:

```
n <- m[1:2,1:2]
n
##      [,1] [,2]
## [1,] 101 104
## [2,] 102 105
```

In questo caso, abbiamo isolato un pezzo della matrice `m`, corrispondente alle prime 4 celle (quelle nelle prime due righe e prime due colonne) e lo abbiamo assegnato all'oggetto `n`.

Un ultimo esempio, la selezione di elementi non contigui:

```
n <- m[c(1,3),c(1,3)]
n
##      [,1] [,2]
## [1,] 101 107
## [2,] 103 109
```

Analizzando il codice dell'esempio, dovremmo poter capire che, all'interno delle quadre, abbiamo indicato due vettori generati con la funzione `c`. Ciascuno di essi corrisponde al vettore composto da due elementi, i numeri 1 e 3, e quindi il risultato complessivo sarà la creazione di un oggetto `n`, anch'esso una matrice, composto dagli elementi nella prima riga e terza colonna di `m`. Torneremo sul concetto di indicizzazione, e sul suo utilizzo, man mano che prenderemo familiarità con oggetti aventi struttura più complessa (come, ad esempio, le *liste* o gli *array*).

Le parentesi graffe



Figura 7 - Le parentesi graffe delimitano un ciclo `for`, il cui seme guiderà le iterazioni.

Servono principalmente a delimitare strutture precise come i *loop* (cicli). Dunque le vedremo comparire all'interno di un codice per generare e gestire delle strutture di controllo complesse come le scelte (il ciclo *if, else*) o iterazioni di blocchi di istruzioni (il ciclo *for*). Un esempio è:

```
valore <- 0
iterazioni <- 5
incremento <- 2
for(i in 1: iterazioni){
  valore <- valore + incremento
}
valore
## [1] 10
```

In questo caso, abbiamo definito tre oggetti (*valore*, *iterazioni*, e *incremento*), e poi abbiamo usato la struttura di controllo *for* per iterare una semplice operazione: prendere *valore* e sommargli *incremento* per un numero di volte pari a *iterazioni*. Il risultato, intuitivo, è 10. Le graffe servono, quindi, per racchiudere il ciclo *for*. Vedremo che le parentesi graffe servono anche a definire le funzioni, ma su questo torneremo più avanti.

Tipologie di dati: gli attributi e le classi

Tutti i linguaggi di programmazione ad oggetti (ed *R* è uno di questi) sono basati sul fatto che blocchi di informazione detti **oggetti** (siano essi quantità, variabili, istruzioni, procedure complesse) siano utilizzati per accedere ad altrettanti blocchi di dati in memoria. Questa è una differenza sostanziale rispetto ad altri linguaggi come *C* che si basano sull'accesso *diretto* ai dati in memoria, cosa che *R* non consente. All'interno di *R* gli oggetti hanno un nome esclusivo e sono l'elemento di base su cui lavorare, qualsiasi sia la loro struttura. Il modo in cui *R* immagazzina in memoria un dato oggetto è detto tipo dell'oggetto stesso. Non è importante approfondire i diversi tipi di oggetti in *R*, per cui non ci dilungheremo sull'argomento.

Quello che, invece, è veramente importante sapere è che ogni tipo di oggetto possiede una serie di **attributi**. Tra questi, quello che spicca per importanza è la **classe**. Come abbiamo già detto in precedenza, le classi sono importanti poiché a seconda di esse variano i metodi che si possono applicare agli oggetti. A questo punto è utile chiarire che i metodi non sono altro che funzioni, e che quindi a ogni classe di oggetti possono essere applicate funzioni diverse, ovvero una stessa funzione può avere comportamenti diversi a seconda della classe dei suoi argomenti (cioè degli oggetti sui quali viene applicata).

All'interno di *R* è possibile operare con diverse tipologie di dati, per ognuna della quali esiste una **classe** predefinita. Sebbene ci siano molte classi già implementate ed altre possano essere definite dagli utenti esperti, esistono 5 classi principali di dati:

- la classe **numeric**, destinata a tutti i **numeri reali** (elementi dell'insieme $\mathbb{R}$);
- la classe **integer**, destinata a tutti i **numeri interi** (elementi dell'insieme $\mathbb{N}$);
- la classe **character**, destinata alle stringhe di testo o alle sequenze di caratteri;
- la classe **logical**, destinata alle espressioni nell'accezione dell'**Algebra di Boole**;
- la classe **factor**, destinata a variabili o quantità qualitative o semi-quantitative.

È facile immaginare il senso delle prime due classi, poiché si riferiscono a insiemi ben noti della matematica.

Anche la terza classe è intuitivamente familiare: tutti sappiamo cosa sono i caratteri di un testo. Per gli altri è sufficiente dire che il testo (formato da caratteri) è un tipo di informazione particolare per gli ambienti di calcolo (come R): si possono valutare le caratteristiche di un testo (quanti caratteri ci sono in una stringa? Quante parole ci sono in un testo?) ma non ha senso applicare funzioni matematiche su di esso.

La quarta classe si riferisce agli oggetti (le espressioni, ma in un senso diverso da quello di cui abbiamo parlato in una sezione precedente a proposito di R) propri dell'Algebra di Boole, il grande matematico inglese che ha gettato le basi per la computazione elettronica.

L'ultima classe è pensata per permetterci di gestire valori semi-quantitativi o qualitativi. Si possono fare moltissimi esempi di "fattori" (**variabili che possono assumere un numero finito di stati o valori**): il sesso di ciascun individuo di una popolazione, il colore degli occhi di diversi individui, la classe di età (coorte) a cui appartengono, l'intensità di un trattamento farmacologico. La cosa importante da dire, in proposito, è che possono esserci fattori non ordinabili e fattori ordinabili. Un esempio del primo tipo sono il sesso o il colore degli occhi: non ha senso dire che maschio viene prima di femmina o che ci sia una scala ordinata di colori (o forse sì?). Esempi di fattori ordinabili sono invece la classe di età (coorte) o l'intensità di un trattamento farmacologico: è possibile stabilire una sequenza continua di valori (es. classe di età 0, classe 1, classe 2 e così via).

Per ciascuna delle 5 classi esiste un comando (**funzione**) che serve a generare un oggetto appartenente alla classe voluta. I comandi in questione hanno lo stesso nome della classe, e quindi avremo le funzioni:

- `numeric();`
- `integer();`
- `character();`
- `logical();`
- `factor().`

L'utilizzo di queste funzioni è abbastanza intuitivo. Ad esempio, la funzione `numeric` può essere utilizzata con un solo argomento (`length`, ovvero lunghezza) per generare un vettore numerico pieno di zeri (0 è il valore di *default*) di dimensione desiderata.

```
a <- numeric(5)
a
## [1] 0 0 0 0 0
```

Lo stesso avviene per `integer`, `character`, e `logical`.

La funzione `integer()` si comporta in maniera simile a `numeric()`:

```
a <- integer(5)
a
## [1] 0 0 0 0 0
```

Con la funzione `character()` otteniamo una sequenza di blocchi di testo vuoti:

```
a <- character(5)
a
## [1] "" "" "" "" ""
```

Con la funzione `logical()` otteniamo una sequenza di espressioni booleane, per definizione *FALSE* (che è il corrispettivo di 0 nell'Algebra binaria di Boole, mentre 1 corrisponde a *TRUE*):

```
a <- logical(5)
a
## [1] FALSE FALSE FALSE FALSE FALSE
```

Una piccola trattazione dell'Algebra di Boole, così importante per la programmazione, ci aspetta più avanti.

Una cosa che merita di essere detta, inoltre, è che R è in grado di riconoscere automaticamente le caratteristiche di un oggetto e assegnarlo alla classe opportuna. Proviamo a riprendere uno degli esempi

precedenti e, dopo aver generato una variabile di nome `a` alla quale assegniamo valore 4, usiamo la funzione `class` per chiedere a *R* che cosa è `a`:

```
a <- 4
class(a)
## [1] "numeric"
```

In questo esempio, *R* ha correttamente dedotto che volessimo generare una quantità numerica. In effetti, avremmo potuto anche scegliere di assegnare `a` alla classe *integer*. Tuttavia *R* sceglierà sempre come *default* la classe *numeric* per tutti gli oggetti che contengono solo numeri. Per chi ha già familiarità con linguaggi di programmazione, *numeric* è un sinonimo di *double*, e in effetti *double* rappresenta il modo in cui gli oggetti *numeric* vengono archiviati in memoria.

È anche possibile definire un oggetto senza inserirne direttamente il valore, e l'esempio che segue ci permette anche di riprendere il concetto di **eredità** (*inheritance*):

```
a <- sqrt(9) + 2^2
a
## [1] 7
class(a)
## [1] "numeric"
```

In questo esempio abbiamo definito la quantità `a` come la somma della radice quadrata di 9 (la funzione `sqrt` è appunto l'acronimo di *square root*) e del numero 2 elevato al quadrato. Interrogato sul valore di `a` e sulla sua classe, *R* ci risponde che `a` vale 7 e che è un *numeric*. *R* ha automaticamente calcolato il risultato e lo ha assegnato all'oggetto `a`. Inoltre, ha dedotto che `a` è un oggetto della classe *numeric*, e questo è precisamente un esempio di *eredità* della classe.

Strutture di dati

A questo punto, dopo aver visto quali sono le principali tipologie di dati che si possono gestire in R, dobbiamo comprendere in quali modi è possibile organizzarli. L'oggetto più semplice all'interno di R è il vettore, ma strutture di base sono anche le **matrici**, gli **array**, le **liste** e, infine, i **data frame** (che possono essere considerati degli oggetti "speciali").

Vettori

Un vettore può essere pensato come una serie di celle concatenate, ognuna delle quali contiene un valore. A ciascuna cella o elemento del vettore corrisponde un *indice di posizione* che, come nel caso dei numeri civici su una via che consentono a un corriere postale di trovare un destinatario specifico, possono essere usati per manipolare il vettore, ad esempio estraendone i valori presenti in celle precise identificate tra quadre in una indicizzazione. La cosa importante da assimilare è che **un vettore può contenere solo elementi che appartengono tutti a una stessa classe**. Questo vuol dire che R gestisce vettori di tanti tipi, i principali dei quali sono proprio i *character*, *integer*, *logical*, *numeric*... insomma proprio le classi che abbiamo visto poco fa.

Generare un vettore è molto semplice, e indicheremo due modalità alternative:

- Mediante la sua "**funzione generatrice**". Ne esiste una per ogni tipo di vettore, e il suo nome è intuitivo. Ad esempio, per generare un vettore *numeric* c'è la funzione omonima **numeric()**. Il suo utilizzo, come quello delle sue "sorelle" **character()**, **logical()**, **integer()** e **factor()** (ma vale anche per gli altri tipi appena menzionati), è molto semplice:

```
a <- numeric(10)
a
## [1] 0 0 0 0 0 0 0 0 0 0
```

```
class(a)
## [1] "numeric"
```

L'argomento all'interno delle parentesi specifica la lunghezza del vettore che si vuole generare. Si noter  che, per *default*, *R* ha generato un vettore composto da 10 quantit  tutte uguali a 0.

Intuitivamente, la funzione `character()` genera una serie di stringhe di lunghezza 0 (vuote):

```
a <- character(10)
a
## [1] "" "" "" "" "" "" "" "" "" ""
class(a)
## [1] "character"
```

Se non lo avete gi  notato, le quantit  *character* sono sempre facilmente identificabili perch  *R* le racchiude tra le virgolette. In entrambi gli esempi, l'applicazione della funzione `class` sugli oggetti generati restituisce una risposta analoga a quelle viste nel paragrafo precedente, e naturalmente univoca perch  tutti gli elementi del vettore sono della stessa classe.

- Mediante la funzione `c`, che abbiamo gi  incontrato:

```
a <- c(1, 2, 3)
a
## [1] 1 2 3
class(a)
## [1] "numeric"
a <- c("A", "B", "C")
a
## [1] "A" "B" "C"
class(a)
## [1] "character"
```

Noterete che nel primo caso abbiamo messo insieme (concatenato) 3 numeri per generare il vettore `a`, immediatamente classificato da *R* come *numeric*. Nel secondo caso abbiamo messo insieme le prime tre lettere

dell'alfabeto, ottenendo un vettore che *R* classifica come *character*. Per ciascuna delle funzioni generatrici esiste una sorella “coercitrice”. Per esempio a `numeric()` corrisponde `as.numeric()`, così come a `character()` corrisponde `as.character()`. Vediamo un esempio illuminante:

```
a <- c(1, 2, 3)
a
## [1] 1 2 3
class(a)
## [1] "numeric"
a <- as.character(a)
a
## [1] "1" "2" "3"
class(a)
## [1] "character"
```

In questo caso abbiamo prima generato un vettore *numeric* e poi lo abbiamo “costretto” a diventare *character*. A volte i neofiti di *R* sono confusi da alcuni aspetti di queste trasformazioni, per cui vediamo cosa succede caso per caso.

```
# Da numeric a character
a <- c(1, 2, 3)
a <- as.character(a)
a
## [1] "1" "2" "3"
# Da character a numeric
a <- c("1", "2", "3")
a <- as.numeric(a)
a
## [1] 1 2 3
a <- c("A", "C", "C")
a <- as.numeric(a)
## Warning: NAs introduced by coercion
a
## [1] NA NA NA
```

Nei primi due casi è andato tutto bene. Infatti, quando *R* ha **coercito** il vettore "1", "2", "3" a diventare *numeric* ha ottenuto dei numeri... cioè delle quantità che per lui hanno senso e possono essere gestite come tali. Ma, quando abbiamo coercito il vettore "A", "B", "C" a diventare *numeric*, *R* non ha ottenuto dei valori gestibili come *numeric* e ha quindi generato dei valori detti **NA (not assigned)**. Questa è la prima volta che ci imbattiamo in loro, e dobbiamo subito sapere che la loro comparsa può essere indice del fatto che qualcosa nei nostri calcoli è andato storto. Magari c'è un "errore" da qualche parte nel nostro codice. Invece, guardiamo per bene questi esempi, anche perché ne approfitteremo per illustrare la funzione generatrice **factor**:

```
#Da character a factor
a <- c("A", "C", "C")
a <- as.factor(a)
a
## [1] A C C
## Levels: A C

#Da numeric a factor
a <- c(1, 2, 3)
a <- as.factor(a)
a
## [1] 1 2 3
## Levels: 1 2 3

#Da factor a character
a <- factor(c("A", "C", "C"), levels = c("A", "B", "C"))
a
## [1] A C C
## Levels: A B C
a <- as.character(a)
a
## [1] "A" "C" "C"

#Da factor a numeric
a <- factor(c("A", "C", "C"), levels = c("A", "B", "C"))
a <- as.numeric(a)
a
## [1] 1 3 3
```

Nel primo esempio, abbiamo definito il vettore concatenando tre valori, due dei quali sono uguali tra loro, in modo da generare un vettore *character* che poi abbiamo coercito a diventare *factor*. Poichè gli *univoci* nel vettore di partenza erano solo due ("A" e "C"), R ha automaticamente stabilito che il *factor* contiene valori che possono assumere solo uno tra questi due possibili stati. Gli stati o valori che possono essere assunti da un *factor* si chiamano **livelli**. I livelli possono essere esplicitamente definiti mediante la funzione generatrice `factor()` o possono essere "desunti" dall'analisi dei valori assunti dagli elementi di un oggetto già esistente, come avviene anche nel secondo esempio (in cui abbiamo generato un *numeric* e poi lo abbiamo trasformato in un *factor*). Il terzo esempio, invece, ci mostra come possiamo generare un vettore *factor* definendo i suoi livelli in modo che comprendano stati o valori che non sono osservati nel vettore stesso, ma sono teoricamente ammessi. Nello specifico, a comprende solo tre elementi che assumono uno tra due valori ("A" e "C"), ma il livello "B" è ammesso anche se non osservato. Sorvoleremo sulle altre combinazioni di coercizione, ma il neofita è invitato a sperimentare e apprendere in maniera euristica...

Invece, è importante richiamare l'attenzione su un aspetto già menzionato di R: la sua logica ad oggetti è concepita per essere vettoriale. Questo significa che vettori e matrici sono le sue strutture di base, e di fatto ogni oggetto può essere visto come un qualche tipo di vettore. Addirittura gli scalari possono essere visti come speciali vettori di dimensione 1 (un unico elemento). Le conseguenze di questa cosa saranno evidenti nel prossimo paragrafo.

Matrici (e Array)

Ed eccoci qui a parlare di matrici. La prima cosa da dire a proposito di esse è che sono uno degli oggetti elettivi per il calcolo all'interno di R. Moltissime funzioni di base, infatti, sono già predisposte per essere applicate direttamente sulle matrici e sono "performanti" in termini di tempo di esecuzione.

Ma cosa è una matrice? Una matrice può essere definita come un

reticolo (in inglese *lattice*) di valori omologhi (nel senso che appartengono tutti alla stessa tipologia, per esempio sono tutti *numeric* o tutti *character*) disposti in maniera regolare in righe e colonne. Ogni valore sarà pertanto individuato in maniera univoca da una coppia di numeri interi che corrisponde ai suoi indici di posizione all'interno della matrice (riga e colonna). Questa logica topografica è la stessa che utilizzavamo quando giocavamo alla battaglia navale sui banchi di scuola.

La sintassi del comando che si deve usare per generare una matrice è molto semplice. Quello che occorre è specificare il contenuto della matrice e la sua geometria (quante righe, quante colonne).

```
# Genero la matrice M avente 3 righe, 2 colonne e contenente i
# numeri da 1 a 6
M <- matrix(data = 1:6, nrow = 3, ncol = 2)
M
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Questo primo, banale esempio ci aiuta a capire 2 cose importanti:

- Quando visualizziamo una matrice, *R* utilizza l'*indicizzazione* per mostrarci la sua struttura reticolare. Infatti all'inizio di ogni riga e sopra ogni colonna compaiono due parentesi quadre con all'interno i rispettivi indici di riga e di colonna.
- Nel comando precedente abbiamo specificato che l'argomento *data* (cioè il contenuto della matrice) deve corrispondere ai numeri da 1 a 6. Ebbene, *R* dispone questi numeri in 2 colonne e partendo dall'angolo sinistro e procedendo a riempire la matrice "per colonne".

In effetti, il comando `matrix` ha un'opzione che ci permette, se lo vogliamo, di riempire la matrice "per riga":

```
# Genero la matrice M avente 3 righe, 2 colonne e contenente i numeri
# da 1 a 6
# Riempiendola per riga
M <- matrix(data = 1:6, nrow = 3,
```

```

      ncol = 2, byrow = T)

M
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6

```

In questo esempio, l'*attivazione* dell'opzione `byrow` ci ha restituito un oggetto con lo stesso contenuto della matrice precedente ma con una diversa disposizione dei numeri al suo interno.

In entrambi i casi, comunque, la matrice può essere vista come una serie di vettori (righe o colonne) appiccicati insieme. Una dimostrazione di questo ci viene dall'utilizzo dei comandi `cbind` e `rbind` (acronimi di "columns bind" e "rows bind"):

```

# Genero la matrice M avente 3 righe, 2 colonne e contenente i numeri
# da 1 a 6
# Unendo due vettori per colonna
c1 <- 1:3
c2 <- 4:6
M <- cbind(c1, c2)
M
##      c1 c2
## [1,]  1  4
## [2,]  2  5
## [3,]  3  6
class(M)
## [1] "matrix" "array"
# Unendo due vettori per riga
M <- rbind(c1, c2)
M
##      [,1] [,2] [,3]
## c1     1    2    3
## c2     4    5    6
class(M)
## [1] "matrix" "array"

```

In entrambi i casi otteniamo un oggetto di classe `matrix`. Da questi semplici esempi si deduce che una matrice è un oggetto sempre e solo bidimensionale. Ebbene, per comprendere gli `array` basta sapere che essi sono semplicemente matrici sono più di 2 dimensioni. Non ci dilunghiamo oltre sulle matrici, ma torneremo estesamente su di esse quando (tra poco) parleremo di funzioni.

Liste

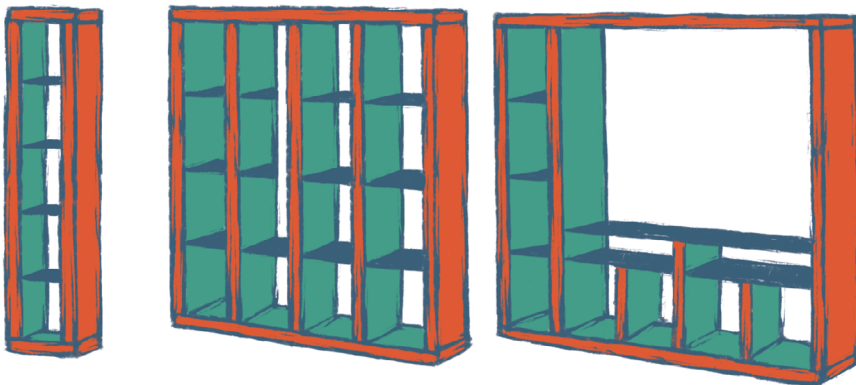


Figura 8 - Vettori, Matrici e Liste hanno una diversa struttura: i primi due tipi di oggetti sono “regolari”, mentre le liste possono avere una struttura irregolare per contenere oggetti disomogenei.

Le *liste* sono delle vere e proprie “collezioni” di oggetti in R . Ogni lista è di per sé un oggetto, ma in pratica ogni lista contiene dei sotto-oggetti (matrici, vettori, scalari, `array`, data frame, altre liste). L'utilità delle liste sta proprio nella loro versatilità strutturale: mentre vettori, matrici ed `array` sono oggetti che contengono elementi di un solo tipo, le liste possono contenere una molteplicità di sotto-oggetti di tipo diverso. Inoltre, tutti gli altri oggetti di R sono “regolari”: la loro struttura è organizzata in dimensioni (righe o colonne) rigide: ci si aspetta che ogni riga di una matrice contenga un elemento per ciascuna colonna, e viceversa.

Per comprendere la differenza tra le *liste* e gli altri oggetti, può essere utile pensare alla struttura di diverse librerie che possiamo trovare negli

uffici o in casa nostra. La figura parla da sé: non potreste mai mettere la TV in una matrice o in un vettore!

È possibile generare una lista mediante il comando `list`, di cui vediamo una prima semplicissima applicazione:

```
# Genero il vettore V avente 5 elementi numerici
V <- c(4, 9, 1, 0, 7)
V
## [1] 4 9 1 0 7
# Genero la matrice M avente 3 righe, 2 colonne e contenente i
# numeri da 1 a 6
M <- matrix(data = 1:6, nrow = 3, ncol = 2)
M
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
# Genero la lista L contenente i sotto-oggetti V e M
L <- list(V, M)
L
## [[1]]
## [1] 4 9 1 0 7
##
## [[2]]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Si vede subito che all'interno di `L` ci sono sia `V` che `M`.

Ora, prima di passare alle funzioni (che abbiamo subdolamente introdotto senza parlarne in maniera esplicita...), vediamo l'ultima tipologia di oggetto R di base: i `data.frame`.

Data Frame

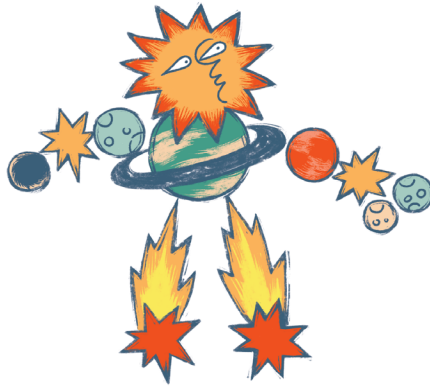


Figura 9 - Un data frame può contenere tante colonne appartenenti a classe diverse.

È abbastanza facile immaginare un data frame come una matrice, quindi un oggetto regolare che contenga diversi tipi di oggetti, tipicamente un tipo per ogni colonna. Ad esempio:

```
# Genero il vettore V avente 5 elementi numerici
V <- c(4, 9, 1, 0, 7)

# Genero il vettore C avente 5 elementi character
C <- c("A", "A", "B", "B", "C")

# Genero il vettore H avente 5 elementi factor
H <- factor(c("L1", "L2", "L2", "L2", "L1"), levels = c("L1", "L2"))

# Genero il data frame df contente gli oggetti V, C e H
df <- data.frame(V, C, H)
df
##   V C  H
## 1 4 A L1
## 2 9 A L2
## 3 1 B L2
## 4 0 B L2
## 5 7 C L1
```

Comodo no? Immaginate di essere un ecologo intento nel monitoraggio di una popolazione animale “sul campo”. Per ogni individuo analizzato (la riga del data frame) raccoglierete una serie di misure. Alcune continue (il peso, la lunghezza), altre *character* (il nome o il codice dell'esemplare), altre *factor* (il sesso), e così via. Vi sarà possibile organizzare queste molteplici informazioni nelle colonne di un data frame in modo da avere il corrispondente informatico di una tabella. È chiaro che tutto questo è molto intuitivo e versatile, ed è esattamente il motivo per cui, nella maggior parte dei corsi su *R*, i data frame sono un argomento centrale e sono considerati gli oggetti più importanti all'interno di *R*.

Tuttavia, da un punto di vista computazionale, i data frame sono oggetti da prendere con le pinze. Questo anche perché per usare i data frame è necessario avere una certa familiarità con tutte le tipologie di dati (*numeric*, *character*, *factor*, ecc.) Torneremo in maniera approfondita sui data frame quando affronteremo le diverse “filosofie” per produrre grafici, ma per ora li mettiamo da parte.

Sintassi di base (II): le funzioni

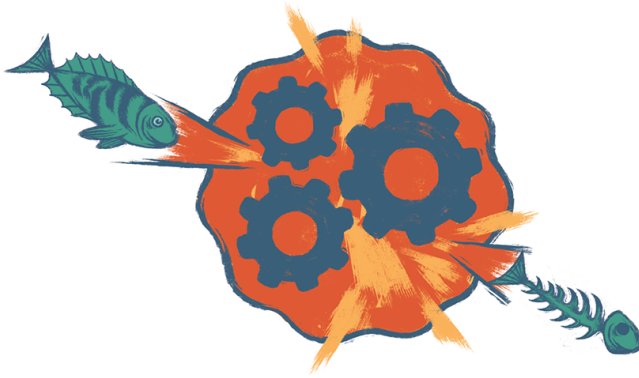


Figura 10 - Gli oggetti entrano in una funzione e ne vengono trasformati.

Abbiamo già spiegato che *R* è un ambiente *object-oriented* e, quindi, la manipolazione e trasformazione dei dati contenuti negli oggetti rappresenta in estrema sintesi il metodo di lavoro che si utilizza in questo ambiente. Le **funzioni** non sono altro che sequenze di istruzioni per manipolare dati in ingresso ed ottenere un certo risultato in uscita. *R*, anche nella sua versione base, contiene centinaia di funzioni preimpostate e disponibili, ma diciamo da subito che l'utente, anche se è un principiante, può definirne di nuove e anzi, il successo di *R* si deve soprattutto alla sua capacità di stimolare la fantasia e la creatività.

Per impadronirci di questo strumento essenziale, tuttavia, dobbiamo comprenderne ed assimilarne gli aspetti di base:

- Ogni funzione ha un suo **nome** preciso, non ambiguo ed unico. In generale, non possiamo definire oggetti che hanno lo stesso nome delle funzioni e viceversa (o meglio possiamo farlo, ma a nostro rischio e pericolo).
- Ogni funzione si utilizza mediante il suo nome seguito da una coppia di parentesi tonde al cui interno sono inseriti i riferimenti agli **argomenti** della funzione e alle sue **opzioni**.
- **Argomenti** ed **opzioni** sono specifici di ogni funzione. Questo significa che, col tempo, è necessario (e automatico) imparare a memoria tanti nomi di funzioni e dei loro argomenti...

Vediamo alcuni esempi cominciando da una funzione che abbiamo già incontrato: la funzione `matrix()`.

```
# Genero una matrice M avente 3 righe, 2 colonne e contenente i  
numeri da 1 a 6  
M <- matrix(data = 1:6, nrow = 3, ncol = 2)
```

In questo esempio di applicazione notiamo che:

- definiamo un oggetto di nome **M** mediante assegnazione alla funzione `matrix`. Questo significa che quello che esce dalla parte destra dell'assegnazione diventerà **M**;
- usiamo la funzione mediante il suo nome seguito da una serie di *input* forniti all'interno di una coppia di parentesi tonde;
- quello che c'è tra le parentesi tonde è una serie di assegnazioni con tanto di nomi e di valori. Questa è la modalità di utilizzo più “*verbose*”, cioè esplicita e prolissa. In realtà, potremmo omettere i nomi degli argomenti purché li inseriamo nell'ordine in cui la funzione se li aspetta. Ad esempio:

```
# Genero una matrice M avente 3 righe, 2 colonne e contenente i  
numeri da 1 a 6  
# Sintassi “verbose”  
M <- matrix(data = 1:6, nrow = 3, ncol = 2)  
  
# Sintassi equivalente omettendo gli argomenti  
M <- matrix(1:6, 3, 2)
```

Dunque la funzione `matrix()`, per funzionare, ha bisogno di 3 argomenti: `data`, `nrow` e `ncol`. Il significato dei loro nomi dovrebbe essere intuitivo: `data` corrisponde ai dati da inserire (i valori nelle celle della matrice), `nrow` specifica il numero di righe della matrice, `ncol` specifica il numero di colonne.

Non tutti gli argomenti di una funzione hanno la stessa importanza. Nel caso di `matrix`, la geometria della matrice da creare (cioè il suo numero di righe e di colonne) è più importante del contenuto. Ce ne accorgiamo provando a fare, ad esempio:

```
# Genero la matrice M "vuota" avente 3 righe, 2 colonne
M <- matrix(NA, 3, 2)
M
##      [,1] [,2]
## [1,]  NA  NA
## [2,]  NA  NA
## [3,]  NA  NA
```

L'oggetto **M** contiene una serie di valori tutti uguali chiamati **NA**. Scopriremo tra poco di cosa si tratta, ma intanto notiamo che la geometria è quella desiderata. Allo stesso modo, possiamo generare una matrice contenente valori tutti uguali (ma definiti):

```
M <- matrix(1, 3, 2)
M
##      [,1] [,2]
## [1,]    1    1
## [2,]    1    1
## [3,]    1    1
```

In questo caso *R* ha popolato la matrice con tanti valori pari a 1.

È interessante commentare cosa succede quando l'*input* non è "coerente". Ad esempio:

```
M <- matrix(1:7, 3, 2)
## Warning in matrix(1:7, 3, 2): data length [7] is not a sub-
## multiple or multiple
## of the number of rows [3]
M
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

In questo caso abbiamo indicato a *R* che la matrice deve contenere i valori da 1 a 7, pur avendo 3 righe e 2 colonne (e quindi 6 celle). *R*,

obbediente, produce un oggetto **M** identico ai precedenti omettendo l'ultimo valore da noi indicato per l'argomento *data* (il 7), ma ci avvisa che “i conti non tornano” mediante un “*warning*”. Come dice il suo nome, un **warning** non è un errore, ma un avvertimento, a cui però è importante fare attenzione perché l'incoerenza degli argomenti potrebbe essere frutto di un banale errore di digitazione. Vediamo un altro esempio:

```
M <- matrix(1:5, 3, 2)
## Warning in matrix(1:5, 3, 2): data length [5] is not a sub-
## multiple or multiple
## of the number of rows [3]
M
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    1
```

Se guardiamo bene il contenuto di **M**, ci accorgiamo che sono presenti i numeri da 1 a 5, inseriti per colonne, ma nell'ultima cella della matrice (quella in basso a destra) compare una seconda volta il numero 1 (il primo della serie da noi indicata). Il motivo è semplice: abbiamo detto a *R* di generare una matrice di 6 celle ma gli abbiamo fornito solo 5 valori per riempirla. Risultato: *R* ha “riutilizzato” il primo valore della serie.

Altra cosa da dire importante sulle funzioni è che, oltre agli **argomenti** (indispensabili), può essere importante conoscere e padroneggiare le opzioni. Riprendiamo l'esempio di `matrix`:

```
M <- matrix(1:6, 3, 2, byrow = TRUE)
M
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
M <- matrix(1:6, 3, 2, byrow = FALSE)
M
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Che differenza notiamo tra questi due comandi? Nel primo caso *R* ha usato la sequenza da 1 a 6 per riempire la matrice *M* “per colonna”, nel secondo caso lo ha fatto “per riga”. Questo ci permette di chiarire in senso generale l’uso delle opzioni: **esse rappresentano dei comandi di input delle funzioni utilizzate per alterarne il comportamento. Tuttavia, essendo appunto opzionali, ogni funzione ha sempre una modalità di esecuzione “di base” (o di default come dicono gli inglesi) che le opzioni hanno il potere di alterare.**

Per disattivare un’opzione dobbiamo porla uguale a *FALSE*, mentre se la vogliamo attivare dobbiamo porla uguale a *TRUE*. Questa è un’altra peculiarità delle opzioni: sono delle variabili booleane e possono assumere solo due valori, *TRUE* e *FALSE*, appunto. Se comprendiamo questo, possiamo facilmente intuire perché le opzioni sono sempre per definizione disattivate e siamo noi a doverle “attivare”. Ad esempio:

```
M <- matrix(1:6, 3, 2)
M
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
M <- matrix(1:6, 3, 2, byrow = FALSE)
M
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Questi due comandi producono risultati identici, perché *sono* identici: nel secondo caso l’opzione *byrow* è espressamente ed esplicitamente

annullata dall'utente, nel primo lo è perché quello è lo stato predefinito di ogni opzione.

Questo punto ci fornisce l'appiglio per introdurre una nuova, semplice funzione: `sum`. Come dice il suo nome, la funzione `sum` calcola la somma degli oggetti che le vengono forniti come *input*, indipendentemente dalla loro geometria (possono essere vettori o matrici o array) ma non dal loro tipo (la somma è definita solo per le quantità *integer* e *numeric*, non per quelle *factor* o *character* o per le altre). Quindi:

```
# Genero la matrice M avente 3 righe, 2 colonne
M <- matrix(1:6, 3, 2)
# Sommo gli elementi della matrice
SM <- sum(M)
SM
## [1] 21
# Genero il vettore V avente 5 elementi numerici
V <- c(4, 9, 1, 0, 7)
# Sommo gli elementi del vettore
SV <- sum(V)
SV
## [1] 21
```

Ok? Allora adesso introduciamo volutamente un esempio “rognoso”:

```
V <- c(1, 3, 5, -2, 3, 7)
V
## [1] 1 3 5 -2 3 7
V <- sqrt(V)
## Warning in sqrt(V): NaNs produced
V
## [1] 1.000000 1.732051 2.236068      NaN 1.732051 2.645751
M <- matrix(V, 3, 2)
SM <- sum(M)
SM
## [1] NaN
```

Analizziamo il pasticcio che abbiamo combinato:

- Abbiamo definito il vettore **V** come composto da 6 elementi numerici, di cui uno negativo.
- Abbiamo usato la semplice funzione **sqrt** (acronimo di *square root*) per calcolare le radici quadrate dei valori di **V**.
- Siccome non esiste un numero reale che corrisponda alla radice quadrata di un negativo, il nuovo vettore **V** contiene un valore “strano” che non abbiamo mai incontrato: **NaN** (acronimo di *not-a-number*).
- Siamo andati avanti generando la matrice **M** come se nulla fosse...
- Quando abbiamo calcolato il valore della somma **SM** di **M**, abbiamo ottenuto una brutta risposta: ancora un NaN!!!!

Perché ci siamo andati a impelagare in questa brutta storia? Per mostrare che la funzione **sum** ha un'utilissima (ma pericolosa!) opzione denominata **na.rm**. Proviamo ad attivarla:

```
M <- matrix(V, 3, 2)
SM <- sum(M, na.rm = TRUE)
SM
## [1] 9.345921
```

Questa volta abbiamo ottenuto un valore “buono” della somma di **M**. Se facciamo due calcoli a mente possiamo facilmente dedurre che esso corrisponde alla somma di tutti i valori di **M** tranne quello strano e brutto NaN, con il quale faremo conoscenza tra poco. Adesso però ci interessa un'altra cosa: l'opzione **na.rm** sta per “*remove NA or NaN*”, e quindi significa autorizzare la funzione a far finta che non ci siano valori strani. Quindi, attivandola, abbiamo detto a **R** di calcolare la somma solo sugli altri valori. A questo punto qualcuno di voi lettori si starà chiedendo: e perché mai questa fantastica opzione è disattivata di *default* e dobbiamo attivarla noi? Non sarebbe meglio se fosse sempre accesa e ci proteggesse da risultati brutti? La risposta è NO, non sarebbe affatto meglio, perché il fatto che i nostri calcoli si inceppino ogni volta che da qualche parte compare un valore strano è un ottimo modo per costringerci a ricontrollare le nostre procedure e fare in modo che siano le più generali e versatili possibili. Per esempio, vogliamo risolvere un problema ma non prendiamo in considerazione che i valori di alcune

variabili possono assumere valori negativi. Se così è, e da qualche parte nel nostro codice usiamo funzioni non definite sull'insieme R dei numeri reali (come la radice quadrata o il logaritmo) ci imbatteremo in un valore anomalo e saremmo naturalmente portati a rimediare.

Ci soffermiamo ora su un altro aspetto essenziale delle funzioni. Mi riferisco al fatto che ciascuna di esse vive e opera in un sotto-ambiente di quello principale, cioè la sessione di lavoro all'interno della quale stiamo operando. Vediamo cosa significa con un esempio:

```
x <- 5
y <- 3
sum(x = 5, y = 0)
## [1] 5
x
## [1] 5
y
## [1] 3
```

Sebbene avessimo definito x e y con valori pari a 5 e 3, la somma (in cui abbiamo ri-definito i valori di x e di y), ci ha restituito come somma 0. Inoltre, nell'ambiente principale i valori di x e di y sono rimasti quelli "originali". Questa ultima cosa dipende dal fatto che, all'interno della funzione `sum`, abbiamo usato l'uguale (=) invece dell'assegnazione (<-). Infatti, se facciamo il contrario:

```
x <- 5
y <- 3
sum(x <- 5, y <- 0)
## [1] 5
x
## [1] 5
y
## [1] 0
```

Scopriamo che ora il valore di y è cambiato anche nell'ambiente principale. Questo è il motivo per cui è generalmente sconsigliato fare

assegnazioni nello sviluppo di codici e pacchetti: si rischia di fare un pasticcio tremendo quando li si utilizza perché si rischia di sovrascrivere quantità e oggetti dell'ambiente principale.

Il mondo “a matryoske”



Figura 11 - Le funzioni si annidano le une dentro le altre e ci consentono di scrivere un codice più compatto.

Il mondo delle funzioni di R è fatto come le famose bambole russe, nel senso che ogni funzione generalmente può accettare, come suoi argomenti, anche altre funzioni, a patto che restituiscano oggetti coerenti con gli *input* attesi dalla funzione di rango superiore.

Se, ad esempio, volessi generare una matrice che contiene come elementi i valori restituiti dalla funzione `somma`, scriverò:

```
x <- 1:9
m <- matrix(sum(x), 3, 3)
```

In questo caso, la matrice `m` è riempita con il solo valore 9, cioè l'*output* della funzione `sum` applicata al vettore `x`. Come avevamo già detto, quello che avviene dentro le funzioni di rango più basso non si riflette (se non tramite i risultati) sulle funzioni di rango più elevato. Ancora un esempio:

```
x <- 1:9
m <- matrix(sum(x), 3, 3)
s <- sum(matrix(sum(x), 3, 3))
s
## [1] 405
```

In questo semplice esercizio abbiamo “annidato” tre funzioni una dentro l'altra, ottenendo il valore finale di *s*. Il grande vantaggio di questa potenzialità di *R* è che ci consente di risparmiare sugli oggetti (e i relativi nomi) che sono necessari da generare. Il prezzo è che, con strutture molto complesse e annidate, può essere difficile modificare il codice o correggerlo.

La frontiera del calcolo: NA e NaN

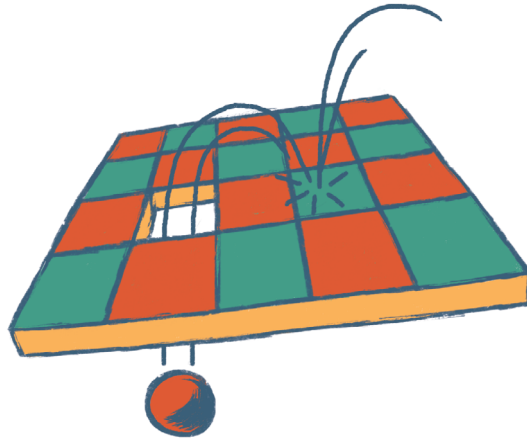


Figura 12 - Durante i vostri calcoli non vi siete mai imbattuti in un “buco” e non siete mai caduti in un NA o un NaN?

Abbiamo recentemente fatto la sgradevole conoscenza del **NaN**. Questa quantità, “*not-a-number*”, viene fuori quanto facciamo calcoli che non sono definiti nell’insieme dei numeri Reali. Quindi:

```
x <- 0
log(x)
## [1] -Inf
x <- -5
sqrt(x)
## Warning in sqrt(x): NaNs produced
## [1] NaN
```

Ora completiamo l’opera con la conoscenza dei **NA**, che sono ancora più antipatici. **NA** sta per “*not_assigned*”, cioè quantità non definita. Ecco alcuni esempi di come può saltare fuori:

```
x <- "Testo"
as.numeric(x)
## Warning: NAs introduced by coercion
## [1] NA
```

In questo caso abbiamo generato un **NA** chiedendo ad *R* di coercire come numerico una quantità testo, il che letteralmente non ha alcun senso. Più in generale, i **NA** sono il classico caso di “dato mancante”. Se si trovano all’interno di un oggetto, i **NA** devono essere interpretati e trattati come dei “buchi”, a differenza dei **NaN** che sono “valori impossibili”. La presenza dei **NA** in un codice è generalmente un sintomo del fatto che la procedura non è abbastanza generale (poiché la presenza di buchi vuol dire spesso che ci sono condizioni che non si realizzano mai, per esempio selezioni che restituiscono un insieme vuoto) e che quindi non è adatta ai dati.

Come abbiamo visto, *R* non nasconde la polvere sotto il tappeto (a meno che non lo invitiamo a farlo), per cui la graduale eliminazione dei passaggi che generano dei **NA** o dei **NaN** rappresenta una normale fase di revisione e *debugging* del codice.

Operatori relazionali e algebra di Boole

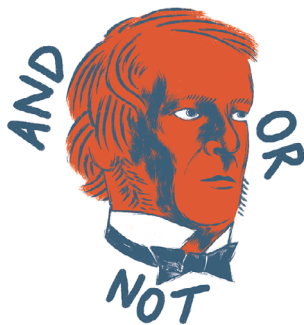


Figura 13 - Un tributo a uno dei più grandi matematici e precursori della storia: George Boole.

Sull’opera immortale di **George Boole** è stato scritto tanto, ma mai a sufficienza. Nessuno più di lui ha contribuito a gettare le basi della computazione elettronica. Tanto è vero che la sua **Algebra** è fondamentale anche per noi. Alla base di tutto c’è il fatto che le quantità che maneggiamo sono sempre tradotte, nel profondo dei nostri PC, in quantità binarie. Come nelle sinapsi del nostro cervello, i microchip dei nostri processori riducono la realtà a sequenze di stati **acceso-spento**.

La prova sta nel fatto che le quantità booleane le abbiamo già incontrate: sono le quantità di tipo **logical**. Adesso dobbiamo generalizzare a problemi più complessi, per esempio la **verifica di condizioni** all'interno di funzioni o algoritmi. Per farlo, prima dobbiamo prendere in rassegna la sintassi degli operatori booleani in R (Tabella 1):

Tabella 1 - Gli operatori booleani in R.

Operatore	Significato
==	uguale
>	maggiore
<	minore
>=	maggiore o uguale
<=	minore o uguale
!	negazione
&	and
	or

Vediamo ora alcune loro semplici applicazioni, in modo da spiegare in maniera intuitiva il loro utilizzo:

```
# Definisco le quantità x e y e faccio ad *R* una serie di domande
x <- 5
y <- 3
x > y # x è maggiore di y?
## [1] TRUE
x < y # x è minore di y?
## [1] FALSE
!(x == y) # negazione di (x è uguale a y)
## [1] TRUE
(x == y) & (x > y) # (x è uguale a y) and (x è maggiore di y)
## [1] FALSE
(x == y) | (x > y) # (x è uguale a y) or (x è maggiore di y)
## [1] TRUE
```


Prestiamo attenzione agli esempi precedenti perché abbiamo introdotto alcune cose importanti:

- Si possono scrivere semplici **espressioni** proprie dall’Algebra di Boole, ovvero quantità logiche che possono essere solo o vere (*TRUE*) o false (*FALSE*) senza ambiguità e complicazioni ulteriori.
- In risposta a queste semplici espressioni, *R* ci restituisce *TRUE* o *FALSE*.
- **Se vogliamo usare la negazione o combinare insieme più espressioni dobbiamo racchiuderle tra tonde perché, di fatto, ogni singola espressione è una piccola funzione.**

È chiaro che tutto questo può esserci utile soltanto se ricordiamo l’Algebra di Boole e, in particolare, le tabelle di verità degli indicatori che si usano per mettere insieme più espressioni:

- Tabella di verità per **AND (&)** (Tabella 2):

Tabella 2 - Tabella di verità dell’operatore AND.

Espressione1	Espressione2	Valore
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
FALSE	FALSE	FALSE

- Tabella di verità per **OR (|)** (Tabella 3):

Tabella 3 - Tabella di verità dell’operatore OR.

Espressione1	Espressione2	Valore
TRUE	TRUE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	FALSE	FALSE

Per capire l'utilità dell'Algebra di Boole in R possiamo usare alcuni semplici esempi:

```
x <- c(9, 4, 6, -7, 7, 3, 3, 1, -14, 0)
x[x > 0]
## [1] 9 4 6 7 3 3 1
x[x < 0]
## [1] -7 -14
x[x == 3]
## [1] 3 3
```

Dopo aver generato il vettore `x`, composto di 10 valori numerici, usiamo le parentesi **quadre** per racchiudere un'espressione che ci permetta di selezionare, usando l'algebra di Boole, alcuni degli elementi di `x`. In questi esempi, estraiamo gli elementi di `x` che sono positivi, o negativi, o uguali al valore 3. Si tratta ovviamente di esempi molto semplici. Ma la logica da utilizzare per combinazioni più complesse è esattamente la stessa... ma per applicarla dobbiamo fare conoscenza con la funzione `which`.

La funzione `which` serve per combinare più condizioni usando gli operatori di Boole. Immaginiamo di voler estrarre da `x`, in un colpo solo, gli elementi maggiori di 3 o minori di -3:

```
x <- c(9, 4, 6, -7, 7, 3, 3, 1, -14, 0)
x[which((x > 3)|(x < -3))]
## [1] 9 4 6 -7 7 -14
```

Naturalmente dobbiamo prestare grande attenzione alla struttura annidata delle parentesi e alla sintassi. Infatti, nel comando precedente, ci sono due condizioni che devono essere valutate, cioè `(x > 3)` e `(x < -3)`. Poiché esse sono collegate dall'operatore booleano **OR (|)**, è sufficiente che una sola di esse sia soddisfatta.

`which` è, in effetti, una funzione estremamente utile e frequente, che ci permette di assemblare condizioni anche molto complesse:

```
x <- c(9, 4, 6, -7, 7, 3, 3, 1, -14, 0)
```

```
x[which((x <= 3)&(x >= -3))&(x != 0))]  
## [1] 3 3 1
```

In questo esempio abbiamo “rovesciato” la selezione precedente, e quindi abbiamo scelto di estrarre i valori compresi tra -3 e 3 ma con l'esclusione dello zero. Si noti che le due condizioni $((x \leq 3) \& (x \geq -3))$ sono racchiuse da una coppia di parentesi tonde e quindi il risultato di questa espressione viene prima calcolato e poi combinato con quello dell'espressione $(x \neq 0)$. Le due espressioni $(x \leq 3)$ e $(x \geq -3)$ hanno quindi la precedenza e un livello più basso, mentre il loro risultato è di pari livello rispetto a $(x \neq 0)$.

La funzione `which` ha anche una opzione molto importante, che si chiama `arr.ind` (contrazione di `array indexes`). Per assimilarla, facciamo un esempio riferito a una matrice:

```
x <- matrix(c(9, 4, 6, -7, 7, 3, 3, 1, -14, 0), nrow = 5)  
  
x[which((x <= 3)&(x >= -3))&(x != 0)]  
## [1] 3 3 1  
x[which((x <= 3)&(x >= -3))&(x != 0), arr.ind = T)]  
## [1] 3 3 1  
which((x <= 3)&(x >= -3))&(x != 0)  
## [1] 6 7 8  
which((x <= 3)&(x >= -3))&(x != 0), arr.ind = T)  
##      row col  
## [1,]   1   2  
## [2,]   2   2  
## [3,]   3   2
```

L'utilizzo dell'opzione `arr.ind` non cambia il risultato dell'indicizzazione di `x`, ma cambia **la forma** in cui la funzione `which` ci restituisce gli **indici di posizione** degli elementi di `x` che soddisfano la condizione specificata. Se `arr.ind` è disattivata, `which` tratta tutti gli oggetti (anche le matrici) come dei vettori, e dunque il risultato che ci fornisce è vettoriale. Se `arr.ind` è attiva e stiamo lavorando su una matrice, `which` ci restituisce delle coordinate in due dimensioni: riga e colonna degli elementi di `x` che soddisfano la condizione desiderata.

L'uso dell'Help

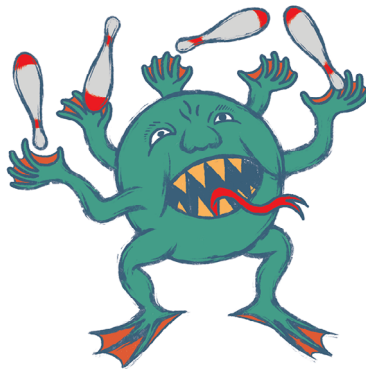


Figura 14 - L'help di R ha un aspetto poco accattivante ma sa dirci tutto quello che serve e risolvere quasi tutti i nostri dubbi.

Fino a questo punto, abbiamo *subdolamente* introdotto alcune funzioni di base di *R* (`matrix`, `which`, `sum`...) illustrandone le principali caratteristiche. Queste e altre funzioni fanno parte del vocabolario di base che un neofita di *R* deve assimilare, ma ce ne sono tante altre che si aggiungono via via ed è evidente che dobbiamo capire come si fa ad apprendere il loro utilizzo (in maniera possibilmente rapida). La risposta è: **usando l'help di R!** L'Help di *R* è un'interfaccia brutta e apparentemente antipatica. Però, a dispetto delle apparenze, è uno strumento estremamente versatile e completo, e quindi imparare a usarlo ci permette di procedere rapidamente con il nostro *self training*.

Per fare la sua conoscenza, proviamo a digitare da terminale il nome di una funzione preceduto dal punto interrogativo:

```
?sum
?which
?matrix
```

L'help di qualsiasi funzione di *R* è organizzato secondo le seguenti voci (per comodità ci riferiremo all'help della funzione `matrix`):

1. Il nome della funzione: `matrix {base}`
2. Il titolo dell'help: **Matrices**

3. Una breve e concisa (occhio ai termini usati) descrizione della funzione: **Description**
4. Un esempio generico ma completo del suo utilizzo: **Usage**
5. Una dettagliata descrizione degli argomenti necessari e delle opzioni: **Arguments**
6. Una articolata descrizione di cosa fa la funzione: **Details**
7. Alcune annotazioni: **Note**
8. I riferimenti alla letteratura (soprattutto se si tratta di un metodo scientifico): **References**
9. Un elenco delle funzioni che sono “apparentate” con la nostra funzione: **See Also**
1. Una serie di esempi pratici: **Examples**

Il nome della funzione (1) è sempre associato al nome del pacchetto, riportato tra parentesi graffe. Per adesso abbiamo incontrato solo funzione del pacchetto base, ma via via ne useremo molte altre disponibili mediante pacchetti aggiuntivi.

Il titolo della funzione (2), da non confondere con il **nome** della funzione, è semplicemente una denominazione immediata ed esplicita di quello che la funzione serve a fare.

La descrizione (3) è una breve frase, basata sull'utilizzo di termini precisi e con un chiaro significato nel contesto di R, di come lavora la funzione e di quello che ci consente di fare.

La voce **Usage** (4) è probabilmente la più diretta, utile ed efficace voce dell'*help*. Man mano che si diventa esperti di R si impara a usare nuove funzioni scorrendo rapidamente il loro *help* e intuendone l'utilizzo proprio dalla voce *Usage*. Essa ci indica immediatamente quali sono gli argomenti (gli *input*) necessari per far funzionare la funzione, ma ci dice anche si comporterà la funzione nel caso in cui non indichiamo esplicitamente alcuni argomenti. Ad esempio, nel caso della funzione `matrix`, ci dice che essa si aspetta 3 argomenti: `data`, `nrow` e `ncol`. Tuttavia, in assenza di indicazioni dell'utente, la funzione userà **NA** come valore di *default* dell'argomento `data`, e 1 come valore di `nrow` ed `ncol`. Vediamo due esempi:

```
matrix(, 3, 3)
##      [,1] [,2] [,3]
```

```
## [1,]  NA  NA  NA
## [2,]  NA  NA  NA
## [3,]  NA  NA  NA
matrix(1:9)
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    5
## [6,]    6
## [7,]    7
## [8,]    8
## [9,]    9
```

Nel primo caso otteniamo una matrice di 3 righe e 3 colonne piena di **NA**. Nel secondo caso abbiamo assegnato a `data` il vettore 1:9, quindi la funzione deve disporre 9 valori in qualche modo... ma, essendo prioritario l'argomento `nrow`, la funzione genera un vettore colonna.

Un aspetto importante è che ogni funzione di *R* si accontenta di avere i valori degli argomenti anche senza l'assegnazione al loro nome predefinito. Quindi se scriviamo:

```
matrix(1:12,4,3)
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

R capirà che 1:9 è riferito a `data`, mentre 4 e 3 sono riferiti rispettivamente a `nrow` ed `ncol`. Quindi la voce *Usage* ci dice anche in che ordine fornire gli argomenti alla funzione che ci interessa. Se vogliamo fornirli in ordine differente, però, (come può capitare con funzioni che hanno tanti argomenti) possiamo farlo usando nomi e assegnazioni. Ad esempio:

```
matrix(nrow = 4, data = 1:12, ncol = 3)
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

Ci fornisce lo stesso identico risultato di prima.

La voce **Arguments** (5), comunque, ci può tornare utile se non abbiamo subito chiara la natura degli argomenti necessari, ad esempio di quale classe (vettore? matrice? lista?) essi devono essere. Allo stesso tempo, questa voce dell'*help* ci spiega bene a cosa servono le opzioni.

La voce **Details** (6) ci illustra il rationale della funzione, cioè quali procedure ci sono alla sua base, ma ci indica anche cosa attenderci in casi "particolari". Nel caso della funzione `matrix`, la voce *Details* ci spiega che, quando la dimensione dell'argomento `data` non corrisponde al numero di celle della matrice, R prova ad "adattare i dati", ad esempio riciclando gli elementi di `data` se sono meno di quelli attesi. Vediamo un esempio:

```
matrix(data = 1:9, nrow = 4, ncol = 3)
## Warning in matrix(data = 1:9, nrow = 4, ncol = 3): data length
## [9] is not a
## sub-multiple or multiple of the number of rows [4]
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6    1
## [3,]    3    7    2
## [4,]    4    8    3
```

In questo caso abbiamo generato una matrice di 4 righe e 3 colonne, e quindi di 12 elementi, ma abbiamo fornito un vettore di soli 9 elementi come argomento `data`. Come si può intuire, R ha ricominciato da capo e dopo il 9 ha usato i valori 1, 2 e 3.

Sulle voci **Note**, **References** e **See Also** non ci dilunghiamo perché sono davvero molto intuitive.

Una cassetta degli attrezzi essenziali

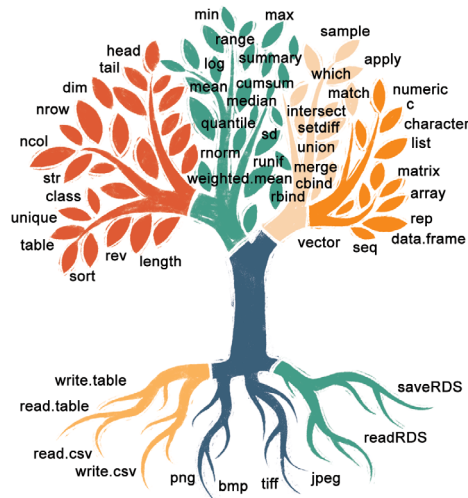


Figura 15 - Una sintesi di alcune delle funzioni di base di R, organizzate per “rami” in base al contesto in cui sono generalmente utilizzate.

In questa sezione faremo conoscenza, anche mediante alcuni esempi, con una serie di funzioni del pacchetto base di *R*. Conoscerle è utile per due motivi:

1. ci permette di avere gli strumenti di base per cominciare a usare sul serio *R*;
2. ci allena ad apprendere in modo autonomo l'utilizzo di qualsiasi altra funzione.

In figura sono rappresentate alcune delle funzioni del pacchetto base, arbitrariamente raggruppate sulla base di 7 diverse tipologie (tratteremo a parte le principali funzioni per fare grafici):

- Ramo **Funzioni per generare oggetti**
- Ramo **Funzioni per esplorare gli oggetti**
- Ramo **Funzioni per la combinazione di oggetti**
- Ramo **Funzioni statistiche di base**
- Ramo **Funzioni per salvare i grafici**
- Ramo **Funzioni per leggere/salvare *file* di testo**
- Ramo **Funzioni per leggere/salvare dati nel formato *R***

Prendiamole in considerazione, illustrandone brevemente le caratteristiche e l'utilizzo.

Funzioni per generare oggetti

`c()`

È la funzione più semplice ed infatti è una delle prime che abbiamo incontrato. Gli *input* di questa funzione sono gli oggetti da unire per creare il vettore, e sono messi insieme nella stessa sequenza con cui vengono menzionati come argomenti. Serve a generare dei vettori che avranno tutti gli oggetti appartenenti alla stessa classe. Questo vuol dire che, se forniamo a `c` argomenti di classi diverse, alcuni di loro saranno *coerciti* a diventare di un'altra classe. Questo avviene secondo una logica intuitiva: se mescolo numeri e caratteri, avrò un vettore di solo caratteri (perché non avrebbe senso trasformare i valori *character* in *numeric*):

```
vector_character = c("A","B")
vector_numeric = c(1:3)

c(vector_numeric, vector_character)
## [1] "1" "2" "3" "A" "B"
```

Invece ha senso convertire i livelli di un *factor* in una serie di valori ordinali:

```
vector_factor = factor(c("a","b"), levels = c("a","b"))
vector_numeric = c(4:6)

c(vector_numeric, vector_factor)
## [1] 4 5 6 1 2
```

Infine, un *factor* mescolato ad un *character* sarà prima convertito in *numeric* e poi in un carattere:

```
vector_factor = factor(c("a","b"), levels = c("a","b"))
vector_character = c("A","B")

c(vector_factor, vector_character)
## [1] "1" "2" "A" "B"
```

numeric()

La funzione per creare vettori di numeri. L'argomento della funzione è la lunghezza del vettore che vogliamo generare. Indipendentemente dalla lunghezza, il vettore sarà costituito solo da zeri. Ha due cugini di cui `as.numeric()` è spesso usato per coercire oggetti di classe diversa da *numeric* in vettori numerici. Se questa operazione non ha senso, la funzione `as.numeric()` restituirà una serie di **NA**:

```
numeric(10)
## [1] 0 0 0 0 0 0 0 0 0 0
numeric(5)
## [1] 0 0 0 0 0
vector_factor = factor(c("a","b"), levels = c("a","b"))
as.numeric(vector_factor)
## [1] 1 2
vector_character = c("A","B")
as.numeric(vector_character)
## Warning: NAs introduced by coercion
## [1] NA NA
```

character()

Questa funzione si comporta esattamente come la funzione `numeric()`: il suo unico argomento è la lunghezza del vettore di caratteri da generare, che sarà formato da stringhe vuote:

```
character(3)
## [1] "" "" ""
```

La funzione con essa imparentata, `as.character()`, trasforma tutto quello che le si dà come argomento in una sequenza di stringhe:

```
vector_factor = factor(c("a","b"), levels = c("a","b"))
vector_numeric = c(4:6)
```

```
as.character(vector_numeric)
## [1] "4" "5" "6"
as.character(vector_factor)
## [1] "a" "b"
```

list()

Si tratta della funzione per assemblare delle liste, ed è molto semplice da usare: è sufficiente fornirle in *input* gli oggetti che formeranno gli elementi della lista. Si può decidere di nominarli oppure di non farlo, a seconda delle esigenze:

```
vector_factor = factor(c("a", "b"), levels = c("a", "b"))
vector_numeric = c(4:6)
vector_character = c("A", "B")
list(vector_factor, vector_numeric, vector_character)
## [[1]]
## [1] a b
## Levels: a b
##
## [[2]]
## [1] 4 5 6
##
## [[3]]
## [1] "A" "B"
list(fattori = vector_factor, numeri = vector_numeric, caratteri =
vector_character)
## $fattori
## [1] a b
## Levels: a b
##
## $numeri
## [1] 4 5 6
##
## $caratteri
## [1] "A" "B"
```

matrix()

Abbiamo già incontrato questa funzione. Essa vuole come argomenti obbligati la geometria della matrice da generare, in termini di numero di righe e numero di colonne, ed il contenuto della matrice stessa (a meno che non ci accontentiamo che sia piena di **NA**). Come nel caso dei vettori, anche le matrici possono contenere solo oggetti che appartengono alla stessa classe:

```
matrix(1:9, 3, 3)
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
matrix(c("a","b","c","d","e","f","g"), 3, 3)
## Warning in matrix(c("a", "b", "c", "d", "e", "f", "g"), 3, 3): data
length [7]
## is not a sub-multiple or multiple of the number of rows [3]
##      [,1] [,2] [,3]
## [1,] "a"  "d"  "g"
## [2,] "b"  "e"  "a"
## [3,] "c"  "f"  "b"
matrix(c("a","b","c","d",1,2,3), 3, 3)
## Warning in matrix(c("a", "b", "c", "d", 1, 2, 3), 3, 3): data length
[7] is not
## a sub-multiple or multiple of the number of rows [3]
##      [,1] [,2] [,3]
## [1,] "a"  "d"  "3"
## [2,] "b"  "1"  "a"
## [3,] "c"  "2"  "b"
```

Naturalmente sappiamo che dobbiamo prestare attenzione al **modo** in cui *R* riempirà la nostra matrice. Per questo dobbiamo ricordare che esiste la funzione **byrow** che ci permette di indicare se la matrice deve essere riempita per riga invece che per colonna.

Un altro aspetto da considerare è la **coerenza** tra le dimensioni dell'argomento *data* e la geometria della matrice, definita dai valori

degli argomenti `nrow` e `ncol`. Se “i conti non tornano”, *R* cerca di farseli tornare ma ci avverte che qualcosa non va...

```
matrix(1:9, 3, 4)
## Warning in matrix(1:9, 3, 4): data length [9] is not a sub-
## multiple of number of columns [4]
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7    1
## [2,]    2    5    8    2
## [3,]    3    6    9    3
```

In questo esempio, abbiamo messo *R* in imbarazzo dando alla funzione `matrix` argomenti discordanti: `data` è definito come un vettore di 9 elementi, ma i valori di `nrow` e `ncol` definiscono una matrice avente 12 celle! Come si vede, *R* “ricicla” gli elementi di `data`, ripartendo dall’inizio del vettore, ma ci avvisa con un warning. Tuttavia...

```
matrix(1:2, 3, 4)
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    1    2
## [2,]    2    1    2    1
## [3,]    1    2    1    2
```

In questo caso *R* non ha protestato, perché replicando gli elementi di `data` è riuscito a ottenere un numero di elementi coerente con la geometria della matrice. Ovviamente questo è un caso particolare della regola generale.

```
matrix(100, 3, 4)
##      [,1] [,2] [,3] [,4]
## [1,]  100  100  100  100
## [2,]  100  100  100  100
## [3,]  100  100  100  100
```

In questo caso, avendo assegnato all’argomento `data` un singolo valore numerico, qualsiasi geometria della matrice andrà bene.

array()

Questa funzione è la corrispondente di `matrix` per matrici a n dimensioni, con n che appartiene a $\mathbb{N}^+$. A parte l'argomento `data`, che è identico a quello di `matrix`, la funzione `array` vuole un argomento `dim`, nella forma di un vettore numerico in cui ogni valore specifica il numero di elementi corrispondente alla n -esima dimensione. Dunque il numero di dimensioni dipende dalla lunghezza del vettore associato a `dim`. Ad esempio:

```
array(1:24, dim = c(3, 2, 4))
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
##
## , , 3
##
##      [,1] [,2]
## [1,]   13   16
## [2,]   14   17
## [3,]   15   18
##
## , , 4
##
##      [,1] [,2]
## [1,]   19   22
```

```
## [2,] 20 23
## [3,] 21 24
```

È interessante fare la conoscenza con il modo, rozzo ma efficace, che R usa per mostrarci oggetti complessi come questo `array` a 3 dimensioni. Per orientarsi si devono utilizzare gli indici, tra parentesi quadre, che ci indicano quale strato dell'`array` ci viene mostrato. Essendo alle prese con un `array` tridimensionale, ci servono 3 coordinate per identificare un suo elemento, e quindi tra le quadre (che servono per indicizzarlo) vediamo 2 virgole che separano 3 valori, siano essi presenti o meno. Quindi la scrittura `[,2,3]` vuol dire prendi TUTTI gli elementi dell'`array` rispetto alla sua prima dimensione, solo il secondo rispetto alla seconda dimensione e solo il terzo rispetto alla terza dimensione. Chiaro no?

```
a = array(1:24, dim = c(3, 2, 4))
a[,2,3]
## [1] 16 17 18
```

data.frame()

I data frame sono sicuramente gli oggetti più flessibili di R, e sono di gran lunga il tipo più usato per organizzare, gestire e analizzare dati biologici. I data frame combinano la “rigidità” strutturale delle matrici (ogni riga di un data frame ha le stesse colonne di ogni altra, e viceversa per le colonne) con la flessibilità del poter incorporare dati *factor*, *numeric*, *character*, ecc. Questo vuol dire che, generalmente, le colonne di un data frame contengono *record* (righe) omogenei e, quindi, ogni variabile (colonna) è formata da valori che appartengono alla stessa classe di R. Questo, però, li rende oggetti non ideali per fare calcoli, come vedremo parlando delle funzioni matematico-statistiche.

Per fare pratica coi data frame introduciamo uno dei *dataset* nativi del pacchetto base di R, il data frame ***iris*** (per ispezionarlo useremo la funzione `head` che descriveremo tra poco):

```
data(iris)
head(iris)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1          3.5          1.4          0.2  setosa
## 2          4.9          3.0          1.4          0.2  setosa
## 3          4.7          3.2          1.3          0.2  setosa
## 4          4.6          3.1          1.5          0.2  setosa
## 5          5.0          3.6          1.4          0.2  setosa
## 6          5.4          3.9          1.7          0.4  setosa
```

Iris è un famoso insieme di dati (Fisher's or Anderson's) che contiene biometrie di individui (fiori) di 3 specie del genere *Iris*. Come si può notare, ogni colonna ha un nome che corrisponde alla biometria misurata, mentre l'ultima colonna è un *factor* con 3 livelli corrispondenti alle 3 specie.

Nel caso dei data frame, oltre alla normale indicizzazione basata sulle parentesi quadre, possiamo disporre di un riferimento diretto alle colonne (variabili) mediante il simbolo **\$**. Ad esempio:

```
iris$Sepal.Length
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8
## 5.7 5.4 5.1
##  [19] 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2
## 5.5 4.9 5.0
##  [37] 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
## 6.4 6.9 5.5
##  [55] 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2
## 5.6 5.9 6.1
##  [73] 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4 6.0 6.7
## 6.3 5.6 5.5
##  [91] 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8 7.1 6.3 6.5
## 7.6 4.9 7.3
## [109] 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0 6.9 5.6 7.7
## 6.3 6.7 7.2
## [127] 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9 6.7
## 6.9 5.8 6.8
```



```
## [145] 6.7 6.7 6.3 6.5 6.2 5.9
iris$Species
## [1] setosa setosa setosa setosa setosa setosa
## [7] setosa setosa setosa setosa setosa setosa
## [13] setosa setosa setosa setosa setosa setosa
## [19] setosa setosa setosa setosa setosa setosa
## [25] setosa setosa setosa setosa setosa setosa
## [31] setosa setosa setosa setosa setosa setosa
## [37] setosa setosa setosa setosa setosa setosa
## [43] setosa setosa setosa setosa setosa setosa
## [49] setosa setosa versicolor versicolor versicolor versicolor
## [55] versicolor versicolor versicolor versicolor versicolor versicolor
## [61] versicolor versicolor versicolor versicolor versicolor versicolor
## [67] versicolor versicolor versicolor versicolor versicolor versicolor
## [73] versicolor versicolor versicolor versicolor versicolor versicolor
## [79] versicolor versicolor versicolor versicolor versicolor versicolor
## [85] versicolor versicolor versicolor versicolor versicolor versicolor
## [91] versicolor versicolor versicolor versicolor versicolor versicolor
## [97] versicolor versicolor versicolor versicolor virginica virginica
## [103] virginica virginica virginica virginica virginica virginica
## [109] virginica virginica virginica virginica virginica virginica
## [115] virginica virginica virginica virginica virginica virginica
## [121] virginica virginica virginica virginica virginica virginica
## [127] virginica virginica virginica virginica virginica virginica
## [133] virginica virginica virginica virginica virginica virginica
## [139] virginica virginica virginica virginica virginica virginica
## [145] virginica virginica virginica virginica virginica virginica
## Levels: setosa versicolor virginica
```

Quindi, usando il comando `class`, possiamo esaminare ogni variabile:

```
class(iris$Sepal.Length)
## [1] "numeric"
class(iris$Sepal.Width)
## [1] "numeric"
class(iris$Petal.Length)
```

```
## [1] "numeric"
class(iris$Petal.Width)
## [1] "numeric"
class(iris$Species)
## [1] "factor"
```

Esistono due modi in cui si può organizzare un data frame. Il primo, detto **wide**, contiene osservazioni ripetute di un soggetto in una singola riga, e ogni risposta è in una colonna separata. Il data frame *iris* è un esempio di data frame *wide*. Nel secondo tipo, detto **long**, ogni riga è combinazione di soggetto e risposta (variabile) su di esso osservata. Il data frame *iris* nel formato *long* ci appare così (occhio al fatto che abbiamo selezionato solo alcune delle tantissime righe che contiene):

```
library(reshape2)
iris$id = 1:150
iris_long = melt(iris, id.vars = "id")
## Warning: attributes are not identical across measure variables;
they will be
## dropped
iris_long[c(1:20, 700:720),]
##      id    variable    value
## 1     1 Sepal.Length    5.1
## 2     2 Sepal.Length    4.9
## 3     3 Sepal.Length    4.7
## 4     4 Sepal.Length    4.6
## 5     5 Sepal.Length     5
## 6     6 Sepal.Length    5.4
## 7     7 Sepal.Length    4.6
## 8     8 Sepal.Length     5
## 9     9 Sepal.Length    4.4
## 10    10 Sepal.Length    4.9
## 11    11 Sepal.Length    5.4
## 12    12 Sepal.Length    4.8
## 13    13 Sepal.Length    4.8
## 14    14 Sepal.Length    4.3
```

```
## 15    15 Sepal.Length      5.8
## 16    16 Sepal.Length      5.7
## 17    17 Sepal.Length      5.4
## 18    18 Sepal.Length      5.1
## 19    19 Sepal.Length      5.7
## 20    20 Sepal.Length      5.1
## 700 100      Species versicolor
## 701 101      Species virginica
## 702 102      Species virginica
## 703 103      Species virginica
## 704 104      Species virginica
## 705 105      Species virginica
## 706 106      Species virginica
## 707 107      Species virginica
## 708 108      Species virginica
## 709 109      Species virginica
## 710 110      Species virginica
## 711 111      Species virginica
## 712 112      Species virginica
## 713 113      Species virginica
## 714 114      Species virginica
## 715 115      Species virginica
## 716 116      Species virginica
## 717 117      Species virginica
## 718 118      Species virginica
## 719 119      Species virginica
## 720 120      Species virginica
```

Quindi ogni soggetto (identificato dal campo *id*, inserito apposta) avrà dati in più righe, e precisamente nel campo *value*, mentre la colonna *variable* specifica a quale variabile è riferito il valore di *value*.

Il lettore attento noterà che siamo passati dal formato **wide** al formato **long** usando funzioni che non sono nel pacchetto base di R ma del pacchetto *reshape2*, ma per ora sorvoleremo (e il lettore ci perdonerà) e torneremo su questo punto più avanti. Intanto ragioniamo sul fatto che il formato **wide** è più familiare (praticamente è una tabella) e sem-

plice da compilare, ma il formato **long** è molto più efficiente e sicuro. Per comprendere il perché di questa affermazione prendiamo il caso di un grande *dataset* che debba essere continuamente aggiornato, magari con osservazioni (nuove righe) non omogenee rispetto alle precedenti (ad esempio abbiamo misurato una nuova biometria rispetto a quelle precedenti). Si può capire facilmente che sarebbe molto più facile aggiungere questa nuova osservazione al data frame **long**, poiché dovremmo specificare i valori solo per il nuovo *record*. Se invece dovessimo aggiungere la nuova osservazione al data frame **wide**, e ci trovassimo ad aver misurato una nuova biometria, sarebbe necessario modificare **tutti** i record precedenti aggiungendo un NA nella nuova colonna per la nuova biometria.

La funzione `data.frame` permette di generare questo genere di oggetti da qualsiasi tipo di dato. Leggendo il suo *help*, noteremo che alla voce *Usage* compaiono tre puntini al posto degli argomenti, e per il resto ci sono solo opzioni.

vector()

Dopo aver descritto le funzioni specifiche per generare vettori di varie classi, vediamo la funzione generica **vector** che serve a generare vettori di qualunque tipo. In effetti, **vector** ha come argomenti *mode*, cioè la classe del vettore da generare, e *length* che ha lo stesso significato visto per *numeric* e *character*.

rep()

Anche questa funzione serve per generare vettori, ma questa volta usando la replicazione di elementi forniti dall'utente. **rep** vuole due argomenti, il primo dei quali (*x*) corrisponde agli elementi da replicare, mentre il secondo può essere *times*, cioè il numero di volte che *x* (in blocco) deve essere replicato, oppure *length.out*, cioè la lunghezza del vettore che si vuole ottenere. Vediamo degli esempi:

```
rep(1, times = 5)
## [1] 1 1 1 1 1
rep("a", times = 5)
## [1] "a" "a" "a" "a" "a"
rep(c(1,2), times = 3)
## [1] 1 2 1 2 1 2
```

Nel primo caso abbiamo replicato un numero, nel secondo una lettera, nel terzo un vettore.

In questo caso, invece, fissiamo la lunghezza del vettore:

```
rep(1, length.out = 7)
## [1] 1 1 1 1 1 1 1
```

seq()



Figura 16 - Come un trampoliere, la funzione `seq` si muove compiendo passi rigidi e fissi.

Questa funzione serve a generare sequenze regolari di valori numerici. I suoi argomenti sono `from` (cioè il punto di partenza della serie regolare - il suo minimo insomma), `to` (il suo massimo, che però può non coincidere con il punto finale della serie generata) e, a scelta, `by` (il passo della sequenza) o `length.out` (la lunghezza del vettore). Vediamo alcuni esempi:

```
seq(from = 1, to = 10, by = 1)
## [1] 1 2 3 4 5 6 7 8 9 10
seq(from = 1, to = 10, by = 2)
## [1] 1 3 5 7 9
seq(from = 1, to = 10, length.out = 11)
## [1] 1.0 1.9 2.8 3.7 4.6 5.5 6.4 7.3 8.2 9.1 10.0
```

Il primo esempio è molto semplice: generiamo una sequenza da 1 a 10 con passo 1.

Il secondo, in cui partiamo da 1 e “puntiamo” a 10 ma con passo 2, *R* si ferma a 9 poiché aggiungere un altro passo significherebbe ottenere 11 e quindi superare il massimo fissato (10).

Nel terzo esempio abbiamo ottenuto un vettore con estremi coincidenti con i valori di *from* e *to* ma con un intervallo decimale poiché abbiamo fissato il numero di elementi del vettore e, quindi, *R* ha calcolato il passo corrispondente per arrivare da 1 a 10 con 11 passi.

Funzioni per esplorare gli oggetti

head() e *tail()*

Queste due funzioni sono molto utili per avere una visione parziale di diversi oggetti, come per esempio i data frame. Nello specifico, *head* permette di visualizzare dal terminale la *testa*, cioè le prime righe del data frame, mentre *tail* permette di visualizzare la *coda*. Entrambi i comandi vogliono un argomento obbligato, *x*, cioè l'oggetto di cui si vuole visualizzare le prime o le ultime *n* righe, dove *n* è un argomento opzionale fissato come valore di *default* a 6. Modificando questo parametro, si possono visualizzare le prime (o le ultime) 10, 20, 30 righe.

```
head(iris)
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species id
## 1          5.1          3.5          1.4          0.2  setosa  1
## 2          4.9          3.0          1.4          0.2  setosa  2
```

```
## 3      4.7      3.2      1.3      0.2 setosa  3
## 4      4.6      3.1      1.5      0.2 setosa  4
## 5      5.0      3.6      1.4      0.2 setosa  5
## 6      5.4      3.9      1.7      0.4 setosa  6
```

`head(iris, 10)`

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	id
## 1	5.1	3.5	1.4	0.2	setosa	1
## 2	4.9	3.0	1.4	0.2	setosa	2
## 3	4.7	3.2	1.3	0.2	setosa	3
## 4	4.6	3.1	1.5	0.2	setosa	4
## 5	5.0	3.6	1.4	0.2	setosa	5
## 6	5.4	3.9	1.7	0.4	setosa	6
## 7	4.6	3.4	1.4	0.3	setosa	7
## 8	5.0	3.4	1.5	0.2	setosa	8
## 9	4.4	2.9	1.4	0.2	setosa	9
## 10	4.9	3.1	1.5	0.1	setosa	10

`tail(iris)`

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	id
## 145	6.7	3.3	5.7	2.5	virginica	145
## 146	6.7	3.0	5.2	2.3	virginica	146
## 147	6.3	2.5	5.0	1.9	virginica	147
## 148	6.5	3.0	5.2	2.0	virginica	148
## 149	6.2	3.4	5.4	2.3	virginica	149
## 150	5.9	3.0	5.1	1.8	virginica	150

`tail(iris, 10)`

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	id
## 141	6.7	3.1	5.6	2.4	virginica	141
## 142	6.9	3.1	5.1	2.3	virginica	142
## 143	5.8	2.7	5.1	1.9	virginica	143
## 144	6.8	3.2	5.9	2.3	virginica	144
## 145	6.7	3.3	5.7	2.5	virginica	145
## 146	6.7	3.0	5.2	2.3	virginica	146
## 147	6.3	2.5	5.0	1.9	virginica	147
## 148	6.5	3.0	5.2	2.0	virginica	148
## 149	6.2	3.4	5.4	2.3	virginica	149
## 150	5.9	3.0	5.1	1.8	virginica	150

nrow()*, *ncol()*, *dim()* e *length()

Facciamo finta di essere alle prese con un data frame che non conosciamo. Probabilmente una delle prime cose da fare è indagare la geometria del data frame, cioè capire quante righe e quante osservazioni ci sono. La funzione `nrow` ci fornisce il numero di righe, `ncol` ci fornisce il numero di colonne, `dim` ce li fornisce entrambi. Nel caso degli `array`, `dim` ci restituisce tutte le dimensioni dell'array stesso.

```
nrow(iris)
## [1] 150
ncol(iris)
## [1] 6
dim(iris)
## [1] 150 6
a = array(1:18, dim = c(3, 3, 2))
dim(a)
## [1] 3 3 2
```

`length` ci fornisce il numero di elementi che caratterizzano un vettore, ma si applica a qualsiasi oggetto di *R*, che come sappiamo è sempre visto da *R* come una struttura vettoriale.

```
length(iris$Sepal.Length)
## [1] 150
length(iris)
## [1] 6
```

str()

A parte le dimensioni, è importante conoscere la *struttura* dei dati, cioè le classi delle colonne del data frame (o di qualsiasi altro oggetto). Vediamo un esempio:


```
str(iris)
## 'data.frame':    150 obs. of  6 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1
1 1 1 1 1 1 1 1 ...
##  $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
```

Come possiamo vedere, *R* ci dice le dimensioni, ma anche la classe di ogni singola colonna/variabile.

```
m = matrix(1:9, 3, 3)
str(m)
##  int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
```

class()

Questa funzione, che abbiamo già incontrato, ci dice a quale classe specifica appartiene un dato oggetto.

```
class(iris$Sepal.Length)
## [1] "numeric"
class(iris$Species)
## [1] "factor"
class(iris)
## [1] "data.frame"
```

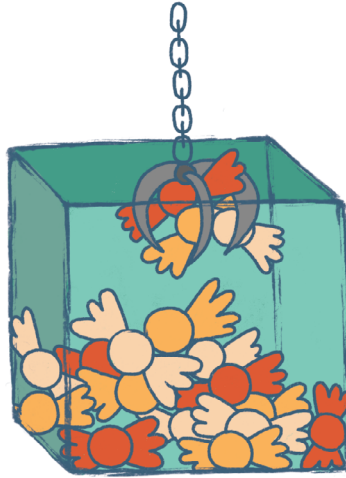
unique()

Figura 17 - La funzione `unique` estrae una singola copia di ognuno degli elementi ridondanti presenti in un oggetto.

Questa funzione ci permette di estrarre i valori **univoci** di qualsiasi oggetto, cioè tutti i valori che ricorrono almeno una volta. Vediamo degli esempi:

```
unique(iris$Sepal.Length)
## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.4 4.8 4.3 5.8 5.7 5.2 5.5 4.5 5.3 7.0
## [20] 6.4 6.9 6.5
unique(iris$Species)
## [1] setosa versicolor virginica
## Levels: setosa versicolor virginica
head(unique(iris))
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species id
## 1 5.1 3.5 1.4 0.2 setosa 1
## 2 4.9 3.0 1.4 0.2 setosa 2
## 3 4.7 3.2 1.3 0.2 setosa 3
## 4 4.6 3.1 1.5 0.2 setosa 4
## 5 5.0 3.6 1.4 0.2 setosa 5
## 6 5.4 3.9 1.7 0.4 setosa 6
```

Nell'ultimo esempio abbiamo usato la funzione `unique` sull'intero data frame. R non ha fatto altro che interpretare le righe (cioè le osservazioni) come unità costitutive del data frame e quindi ha restituito un oggetto con struttura analoga al data frame di partenza, ma avente un numero di righe inferiore nel caso ci fossero individui (righe) perfettamente uguali. Questa cosa per *iris* non succede, ma proviamo a truccare i dati e usiamo `dim` per vedere cosa succede.

```
dim(iris)
## [1] 150   6
dim(unique(iris))
## [1] 150   6
iris[2,] = iris[5,]
dim(unique(iris))
## [1] 149   6
```

Abbiamo sovrascritto la seconda riga di *iris* ponendola uguale alla quinta, ed infatti adesso il numero di righe univoche è 149.

table()

Continuando ad approfondire, potremmo voler capire quale è il numero di osservazioni per ciascuna combinazione di variabili. Nel caso di *iris*, abbiamo visto come usare `unique` per capire, ad esempio, quali specie del genere *iris* sono contenute nel data frame.

```
unique(iris$Species)
## [1] setosa      versicolor virginica
## Levels: setosa versicolor virginica
```

Adesso, vediamo come `table` ci fornisce il numero di occorrenze per ogni specie.

```
table(iris$Species)
##
```

```
##      setosa versicolor virginica
##           50           50           50
```

Gli oggetti che restituisce questa funzione sono delle **tabelle** che corrispondono a vettori nel caso in cui siano stati generati da un vettore (una singola variabile).

Quella dell'esempio è una **tabella di contingenza** per la variabile *specie*, ma possiamo ottenere tabelle anche combinando più variabili, dandole come argomento a `table` e separandole, come sempre, con le virgole. Vediamo un esempio intuitivo.

```
iris$color = factor(rep(c("red","blue"), 75), levels =
c("red","blue"))
table(iris$color, iris$Species)
##
##      setosa versicolor virginica
## red      25          25          25
## blue     25          25          25
```

In questo caso abbiamo generato una nuova colonna di iris, chiamata **color**, e contenente valori casuali di rosso e di blu.

rev()*, *sort()* e *order()

`rev` è una funzione di semplicissimo utilizzo ci permette di *rovesciare* dei vettori. Ad esempio:

```
x = c(1:10)
rev(x)
## [1] 10 9 8 7 6 5 4 3 2 1
```

`sort` ci permette, invece, di *ordinare* un vettore rispetto ai valori dei suoi elementi. Ha un'opzione, `decreasing`, che ci consente di scegliere tra ordine *crescente*, impostato come *default*, e *decrecente*.

```
x = c(sample(1:100, 10))
sort(x)
## [1] 1 4 6 48 50 54 58 59 60 62
sort(x, decreasing = T)
## [1] 62 60 59 58 54 50 48 6 4 1
```

`order`, infine, ci permette di estrarre gli indici di posizione ordinati degli elementi di un oggetto.

```
order(x)
## [1] 6 5 1 9 8 4 3 10 7 2
order(x, decreasing = T)
## [1] 2 7 10 3 4 8 9 1 5 6
```

Funzioni per la combinazione di oggetti

sample()

Più che per la combinazione, `sample` serve per la ri-combinazione: mediante questa funzione è possibile generare un nuovo vettore *campionando* (*sampling*) gli elementi di un altro vettore. Gli argomenti da fornire a questa funzione sono:

1. `x`, cioè l'insieme dei dati da cui estrarre i valori campionati. Generalmente è un vettore numerico ma può essere anche di altro tipo;
2. `size`, cioè il numero di osservazioni che vogliamo estrarre. Qui può nascere un problema. Di *default*, `sample` si comporta come il croupier nella tombola: ogni elemento estratto da `x` viene rimosso da `x`, per cui `x` non può contenere un numero di elementi minore del valore di `size`... a meno che...
3. `replace` (o `rep`) è l'opzione che, se impostata come *TRUE*, autorizza *R* a ricampionare gli stessi elementi di `x`, senza rimuovere i valori estratti;
4. Sempre per *default*, la funzione `sample` estrae "a caso" gli elementi di `x` assegnando a ciascuno di essi la stessa probabilità di estrazione. Mediante l'argomento `prob` è possibile però indicare a *R* un valore di probabilità per ogni elemento di `x`. Esaminiamo alcuni esempi:

```

x = 1:10
y = sample(x, 3)
y
## [1] 4 6 10
y = sample(x, 20, rep = TRUE)
y
## [1] 5 4 4 4 3 4 5 7 2 7 3 2 1 9 3 8 8 9 10 4
y = sample(x, 30, rep = TRUE, prob = c(1, rep(0.1, 9)))
y
## [1] 5 8 8 1 1 8 2 3 2 5 1 10 1 1 1 1 10 7 6 8 1
3 6 1 1
## [26] 1 1 1 10 1

```

Nel secondo esempio abbiamo visto come usare l'opzione `rep`, mentre nell'ultimo esempio possiamo notare come la probabilità di estrazione sia fortemente sbilanciata a favore del valore 1, riflettendosi sulla composizione del campione estratto.

cbind()* e *rbind()

Queste due funzioni ci consentono di “incollare” insieme vettori, matrici o data frame. In particolare `cbind` (*column bind*) permette di incollare colonne, mentre `rbind` permette di incollare righe. Il vincolo principale di queste operazioni è legato alla coerenza della geometria, degli attributi e anche dei nomi. Ad esempio, posso incollare insieme righe di due data frame se esse sono omogenee rispetto alle colonne:

```

data(iris)
small_iris = rbind(iris[1,], iris[51,], iris[150,])
small_iris
##      Sepal.Length Sepal.Width Petal.Length Petal.Width  Species
## 1           5.1         3.5         1.4         0.2    setosa
## 51          7.0         3.2         4.7         1.4 versicolor
## 150          5.9         3.0         5.1         1.8  virginica

```

which()

Abbiamo già parlato di questa funzione in precedenza, nella sezione riguardante l'Algebra di Boole.

match()

Si tratta di una funzione molto utile per associare vettori diversi. Nello specifico, `match` individua tutti gli elementi del vettore `x` (primo argomento) che corrispondono a elementi del vettore `table` (secondo argomento). Vediamo due esempi:

```
v1 = c("A", "B", "B", "C", "A")
v2 = c("A", "B", "C", "D", "E", "F", "G")
match(x = v1, table = v2)
## [1] 1 2 2 3 1
match(x = v2, table = v1)
## [1] 1 2 4 NA NA NA NA
```

Abbiamo generato due vettori, `v1` e `v2`, il primo dei quali contiene meno valori univoci e alcune duplicazioni rispetto al secondo. Quando abbiamo confrontato il primo vettore usando come riferimento (`table`) il secondo, otteniamo dei valori numerici che altro non sono se non gli indici di posizione degli elementi di `v1` rispetto a `v2`. Effettuando il confronto opposto, abbiamo ottenuto 3 valori numerici e 4 NA poiché gli ultimi 4 elementi di `v2` non hanno riscontro in `v1`.

intersect(), setdiff(), e union()

Con questa triade di funzioni facciamo capolino nel salotto di George Boole. Come suggeriscono i loro nomi, `intersect` serve a fare intersezioni, `union` a fare unioni e `setdiff` calcolare la differenza simmetrica (insieme complementare). Ad esempio:

```

v1 = c("A","B","B","C","A","H")
v2 = c("A", "B", "C", "D", "E", "F", "G")

intersect(v1, v2)
## [1] "A" "B" "C"
union(v1, v2)
## [1] "A" "B" "C" "H" "D" "E" "F" "G"
setdiff(v1, v2)
## [1] "H"
setdiff(v2, v1)
## [1] "D" "E" "F" "G"

```

I primi due esempi sono immediati. Gli altri due meritano di essere guardati meglio: a seconda dell'ordine con cui forniamo gli argomenti otteniamo delle differenze simmetriche diverse, anche se in entrambi i casi otteniamo gli elementi del primo insieme che non sono comuni al secondo. Il limite di queste funzioni è che sono tutte capaci di operare solo su 2 argomenti alla volta.

merge()

Chiunque abbia “risistemato” dei dati con un foglio di calcolo, ad esempio in Excel, sa bene come va a finire quando dobbiamo appaiare degli insiemi di dati che “si parlano” tra loro mediante una o più colonne, ma differiscono per altre. Un gran pasticcio, che diventa ingestibile quando i *dataset* sono grandi ed articolati. La funzione `merge` serve proprio a fondere due data frame che abbiano in comune delle colonne (usate come riferimento). Ad esempio:

```

data(iris)
iris$id = 1:150
iris_c = data.frame(id = sample(1:150,150),
                    color = factor(rep(c("red", "blue"), 75),
                                   levels = c("red","blue")))
head(iris_c)

```



```
##      id color
## 1   68  red
## 2   24  blue
## 3   63  red
## 4   26  blue
## 5  126  red
## 6  146  blue
iris_full = merge(iris, iris_c)
head(iris_full, 20)
##      id Sepal.Length Sepal.Width Petal.Length Petal.Width Species
color
## 1    1          5.1          3.5          1.4          0.2  setosa  red
## 2    2          4.9          3.0          1.4          0.2  setosa  blue
## 3    3          4.7          3.2          1.3          0.2  setosa  red
## 4    4          4.6          3.1          1.5          0.2  setosa  red
## 5    5          5.0          3.6          1.4          0.2  setosa  blue
## 6    6          5.4          3.9          1.7          0.4  setosa  blue
## 7    7          4.6          3.4          1.4          0.3  setosa  blue
## 8    8          5.0          3.4          1.5          0.2  setosa  red
## 9    9          4.4          2.9          1.4          0.2  setosa  blue
## 10 10          4.9          3.1          1.5          0.1  setosa  red
## 11 11          5.4          3.7          1.5          0.2  setosa  blue
## 12 12          4.8          3.4          1.6          0.2  setosa  blue
## 13 13          4.8          3.0          1.4          0.1  setosa  red
## 14 14          4.3          3.0          1.1          0.1  setosa  blue
## 15 15          5.8          4.0          1.2          0.2  setosa  red
## 16 16          5.7          4.4          1.5          0.4  setosa  blue
## 17 17          5.4          3.9          1.3          0.4  setosa  red
## 18 18          5.1          3.5          1.4          0.3  setosa  red
## 19 19          5.7          3.8          1.7          0.3  setosa  blue
## 20 20          5.1          3.8          1.5          0.3  setosa  red
```

Noterete che:

- abbiamo aggiunto a iris il campo id come una sequenza numerica progressiva da 1 a 150;
- abbiamo creato un data frame (iris_c) contenente un campo id analogo a quello

di iris, ma con valori casuali (usando opportunamente la funzione `sample`);

- quindi abbiamo accoppiato i due data frame `iris` e `iris_c` usando `merge`.

`merge` è capace di lavorare in autonomia individuando le colonne in comune tra i due data frame e, se queste sono omogenee (ad esempio la variabile `id` non può essere numerica in un caso e *character* in un altro), le usa per fondere i due data frame. Tuttavia questa funzione possiede anche alcuni argomenti opzionali (`by`, `by.x`, `by.y`) che ci consentono di specificare quali colonne usare o quali righe usare.

apply()



Figura 18 - La funzione `apply` è come un chirurgo a nostra disposizione, dobbiamo solo stare attenti a quale strumento chirurgico gli passiamo e al modo in cui gli diciamo di operare.

Abbiamo già avuto modo di spiegare che i data frame non sono oggetti ottimizzati per fare i calcoli. Sono utilissimi per gestire i dati, ma non possiamo applicare ad essi funzioni estremamente utili come `apply`. Per capire a cosa serve questa funzione, prendiamo un esempio: abbiamo bisogno di calcolare (per motivi oscuri) il valore medio delle 4 biometrie contenute in `iris`. Come possiamo fare? È evidente che calcolare riga per riga le medie sarebbe un'operazione laboriosa... oltre al fatto che dovremmo limitare il calcolo alle prime 4 colonne del data frame (escludendo quindi l'ultima colonna `Species`, di tipo *factor*).

```
data(iris)
b1 = mean(as.numeric(iris[,1]))
b2 = mean(as.numeric(iris[,2]))
b3 = mean(as.numeric(iris[,3]))
b4 = mean(as.numeric(iris[,4]))

b1
## [1] 5.843333
b2
## [1] 3.057333
b3
## [1] 3.758
b4
## [1] 1.199333
```

Ok, prima cosa: abbiamo introdotto di soppiatto la funzione `mean`, della quale parleremo tra poco. Per ora basti sapere che questa funzione consente di calcolare la media degli elementi di un vettore numerico. Torniamo alla domanda originale (calcolare il valore medio delle 4 biometrie contenute in `iris`, possibilmente in un'unica operazione)... forse potremmo usare un ciclo iterativo?... ma non abbiamo ancora incontrato i *cicli* e spesso non sono il modo più veloce per fare operazioni di questo tipo.

`apply` ci viene in aiuto! Questa funzione ha 3 argomenti:

- `X`, cioè l'oggetto su cui applicare la funzione `FUN`. `X` può essere un qualsiasi oggetto assimilabile ad un array numerico (se `FUN` si applica ai numeri) o non numerico (se `FUN` si applica anche ad altre classi);
- `MARGIN`, ovvero la "direzione" rispetto alla quale applicare `FUN`;
- `FUN`, cioè la funzione che ci interessa applicare.

Vediamo degli esempi:

```
data(iris)
apply(X = iris[,1:4], MARGIN = 1, FUN = "mean")
## [1] 2.550 2.375 2.350 2.350 2.550 2.850 2.425 2.525 2.225 2.400
2.700 2.500
## [13] 2.325 2.125 2.800 3.000 2.750 2.575 2.875 2.675 2.675 2.675
```

```

2.350 2.650
## [25] 2.575 2.450 2.600 2.600 2.550 2.425 2.425 2.675 2.725 2.825
2.425 2.400
## [37] 2.625 2.500 2.225 2.550 2.525 2.100 2.275 2.675 2.800 2.375
2.675 2.350
## [49] 2.675 2.475 4.075 3.900 4.100 3.275 3.850 3.575 3.975 2.900
3.850 3.300
## [61] 2.875 3.650 3.300 3.775 3.350 3.900 3.650 3.400 3.600 3.275
3.925 3.550
## [73] 3.800 3.700 3.725 3.850 3.950 4.100 3.725 3.200 3.200 3.150
3.400 3.850
## [85] 3.600 3.875 4.000 3.575 3.500 3.325 3.425 3.775 3.400 2.900
3.450 3.525
## [97] 3.525 3.675 2.925 3.475 4.525 3.875 4.525 4.150 4.375 4.825
3.400 4.575
## [109] 4.200 4.850 4.200 4.075 4.350 3.800 4.025 4.300 4.200 5.100
4.875 3.675
## [121] 4.525 3.825 4.800 3.925 4.450 4.550 3.900 3.950 4.225 4.400
4.550 5.025
## [133] 4.250 3.925 3.925 4.775 4.425 4.200 3.900 4.375 4.450 4.350
3.875 4.550
## [145] 4.550 4.300 3.925 4.175 4.325 3.950
apply(X = iris[,1:4], MARGIN = 2, FUN = "mean")
## Sepal.Length Sepal.Width Petal.Length Petal.Width
##      5.843333      3.057333      3.758000      1.199333

```

Nel primo esempio abbiamo applicato la funzione `mean` alle prime 4 colonne di `iris` (quelle con le biometrie), assegnando a `MARGIN` il valore 1 che, come spiega l'*help* di questa funzione, indica le righe. Di conseguenza, `apply` ci restituisce un vettore di 150 elementi (ovvero le righe di `iris`), ognuno corrispondente alle medie che, prima, ci eravamo messi a calcolare una per una.

Nel secondo esempio abbiamo applicato la funzione `mean` alle prime 4 colonne di `iris` (quelle con le biometrie), assegnando a `MARGIN` il valore 2, che indica le colonne. Questa volta `apply` ci restituisce un vettore di medie composto da soli 4 elementi (corrispondenti alle colonne di `iris`).

Vediamo un ultimo esempio che coinvolge la funzione `length`, che già conosciamo:

```
data(iris)
apply(X = iris, MARGIN = 1, FUN = "length")
##   [1] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
##   [38] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
##   [75] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
##  [112] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
##  [149] 5 5
apply(X = iris, MARGIN = 2, FUN = "length")
## Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           150           150           150           150           150
```

Questa volta non è stato necessario restringere il valore di `X` alle prime 4 colonne, poiché questa funzione può essere applicata anche ad oggetti di tipo diverso da *numeric*.

Funzioni statistiche di base

Il prossimo gruppo da esaminare è quello delle funzioni statistiche di base. Cominciamo col dire che tutte queste funzioni si applicano a dati quantitativi, e quindi essenzialmente ad oggetti appartenenti alla classe *numeric*.

min(), *max()*, *range()*

Queste tre funzioni generiche calcolano il minimo (`min`), il massimo (`max`) e l'intervallo (`range`) di un insieme di dati. Vediamo alcuni esempi intuitivi.

```
data(iris)
min(iris$Sepal.Length)
## [1] 4.3
max(iris$Sepal.Length)
## [1] 7.9
range(iris$Sepal.Length)
## [1] 4.3 7.9
```

In tutti questi casi abbiamo usato queste funzioni su vettori, vediamo cosa succede quando siamo alle prese con matrici o data frame.

```
m = matrix(sample(1:100, 25), 5, 5)
range(m)
## [1] 6 100
data(iris)
max(iris[, 1:4])
## [1] 7.9
```

Come si può vedere, queste funzioni “mangiano” di tutto. In più, se lo chiediamo, sono in grado di ignorare i valori “strani” (gli NA e NaN). Per farlo, come abbiamo già visto, è necessario usare l’opzione `na.rm`.

```
m = matrix(sample(1:100, 25), 5, 5)
m[sample(1:5, 2), sample(1:5, 2)] = NA
range(m)
## [1] NA NA
range(m, na.rm = T)
## [1] 2 98
```

summary()

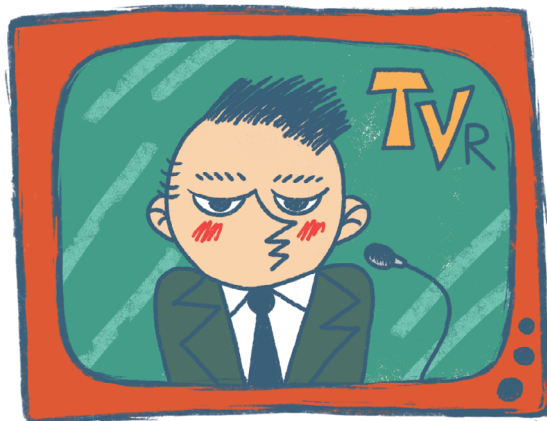


Figura 19 - La funzione `summary` è il nostro personale “anchorman” che ci riassume al volo la situazione.

Questa funzione, come suggerisce il suo nome, serve ad ottenere una visione compatta ma completa delle caratteristiche di un insieme di dati. Vediamo come si comporta su vari oggetti.

```
v = sample(1:100, 10)
summary(v)
```

##	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
##	1.0	16.0	32.5	37.7	56.0	91.0

```
m = matrix(sample(1:100, 25), 5, 5)
summary(m)
```

##	V1	V2	V3	V4	V5
## Min.	:28.0	Min. : 1.0	Min. :14.0	Min. : 7.0	Min. :34.0
## 1st Qu.	:33.0	1st Qu.: 8.0	1st Qu.:24.0	1st Qu.:12.0	1st Qu.:39.0
## Median	:70.0	Median :44.0	Median :29.0	Median :52.0	Median :45.0
## Mean	:55.6	Mean :39.4	Mean :34.2	Mean :49.6	Mean :53.2
## 3rd Qu.	:72.0	3rd Qu.:50.0	3rd Qu.:31.0	3rd Qu.:85.0	3rd Qu.:57.0

```
## Max. :75.0 Max. :94.0 Max. :73.0 Max. :92.0 Max.
:91.0
data(iris)
summary(iris)
## Sepal.Length Sepal.Width Petal.Length Petal.Width
## Min. :4.300 Min. :2.000 Min. :1.000 Min. :0.100
## 1st Qu.:5.100 1st Qu.:2.800 1st Qu.:1.600 1st Qu.:0.300
## Median :5.800 Median :3.000 Median :4.350 Median :1.300
## Mean :5.843 Mean :3.057 Mean :3.758 Mean :1.199
## 3rd Qu.:6.400 3rd Qu.:3.300 3rd Qu.:5.100 3rd Qu.:1.800
## Max. :7.900 Max. :4.400 Max. :6.900 Max. :2.500
## Species
## setosa :50
## versicolor:50
## virginica :50
```

Nel primo caso, relativo al vettore *v*, *R* ci ha restituito la seguente serie di valori:

- **Min.** il valore minimo;
- **1st Qu.** il primo quartile della distribuzione dei dati;
- **Median** la mediana o secondo quartile;
- **Mean** il valore medio;
- **3rd Qu.** il terzo quartile della distribuzione dei dati;
- **Max.** il valore massimo.

Nel caso della matrice *m* o del data frame *iris*, *R* ha restituito gli stesso valori, ma per ogni colonna. Nel caso della colonna *Species* di *iris*, che è un *factor*, ci ha fornito il numero delle occorrenze per ciascuna valore.

La funzione `summary`, comunque, è molto utile anche per capire alla svelta se nell'oggetto che ci interessa ci solo valori mancanti. Ad esempio:

```
m = matrix(log(sample(-100:100, 25)), 5, 5)
## Warning in log(sample(-100:100, 25)): NaNs produced
m
##      [,1] [,2] [,3] [,4] [,5]
## [1,] 4.143135 NaN 4.094345 NaN 4.430817
## [2,] 4.262680 NaN      NaN 3.496508      NaN
```



```
## [3,]      NaN 3.73767      NaN      NaN 2.302585
## [4,]      NaN      NaN 4.219508      NaN 4.110874
## [5,]      NaN      NaN      NaN      NaN      NaN
summary(m)
##          V1          V2          V3          V4
## Min.    :4.143  Min.    :3.738  Min.    :4.094  Min.    :3.497
## 1st Qu.:4.173  1st Qu.:3.738  1st Qu.:4.126  1st Qu.:3.497
## Median :4.203  Median :3.738  Median :4.157  Median :3.497
## Mean    :4.203  Mean    :3.738  Mean    :4.157  Mean    :3.497
## 3rd Qu.:4.233  3rd Qu.:3.738  3rd Qu.:4.188  3rd Qu.:3.497
## Max.    :4.263  Max.    :3.738  Max.    :4.220  Max.    :3.497
## NA's    :3      NA's    :4      NA's    :3      NA's    :4
##          V5
## Min.    :2.303
## 1st Qu.:3.207
## Median :4.111
## Mean    :3.615
## 3rd Qu.:4.271
## Max.    :4.431
## NA's    :2
```

Come si può vedere, in questo caso *R* ci fornisce anche il numero di NA (o di NaN, in questo caso non si fa differenza) per ciascuna colonna della matrice. Per generare i NaN abbiamo usato `sample` per estrarre 25 valori da un insieme che comprende numeri negativi, e poi abbiamo usato la funzione `log` (logaritmo), non definita per i numeri negativi, per ottenere una matrice con una serie di NaN disposti a caso.

`cumsum()`

Questa semplice funzione ci permette di calcolare la *somma cumulata* dei valori di un vettore.

```
v = sample(1:100, 10)
v
```

```
## [1] 71 93 44 37 94 48 73 24 65 74
cumsun(v)
## [1] 71 164 208 245 339 387 460 484 549 623
```

Da notare che questa funzione non ha un'opzione `na.rm`, per cui un singolo NA o NaN nei dati di partenza può causare una “reazione a catena”.

median() e mean()

Queste due funzioni, molto simili, ci permettono di calcolare *media* e *mediana*, i principali *indici di centralità*, su un qualsiasi insieme di dati. Entrambe le funzioni hanno l'opzione `na.rm` che chiede a R di ignorare i valori anomali.

```
m = matrix(log(sample(-100:100, 25)), 5, 5)
## Warning in log(sample(-100:100, 25)): NaNs produced
m
##           [,1]      [,2] [,3]      [,4]      [,5]
## [1,] 4.077537 3.850148  NaN 4.543295      NaN
## [2,]      NaN      NaN  NaN      NaN 4.189655
## [3,]      NaN      NaN  NaN 2.833213 4.219508
## [4,] 3.526361      NaN  NaN      NaN 0.000000
## [5,] 3.891820 3.332205  NaN      NaN      NaN
median(m)
## [1] NA
mean(m)
## [1] NaN
median(m, na.rm = T)
## [1] 3.870984
mean(m, na.rm = T)
## [1] 3.446374
mean(iris)
## Warning in mean.default(iris): argument is not numeric or
logical: returning NA
## [1] NA
```

```
mean(iris[,1:4])  
## Warning in mean.default(iris[, 1:4]): argument is not numeric or  
logical:  
## returning NA  
## [1] NA
```

Negli ultimi due esempi abbiamo avuto problemi nell'applicare la funzione `mean` a `iris`, anche quando ci siamo limitati alle prime 4 colonne che sappiamo essere *numeric*. Questo accade spesso quando si usano funzioni “matematiche” ai data frame, e fin dal primo incontro con questi oggetti avevamo precisato che non sono ottimali per il calcolo. Qui aggiungeremo che, se proprio volessimo calcolare la media di tutti i valori nelle prime 4 colonne di `iris`, saremmo costretti a trattarle come una matrice.

```
mean(as.matrix(iris[,1:4]))  
## [1] 3.4645
```

weighted.mean()

La funzione di R per il calcolo della **media pesata**. Ha bisogno di due *input*: la *x*, cioè il vettore dei valori su cui calcolare la media, e *w*, il vettore dei loro pesi. Se la somma di *w* è diversa da 1, i pesi sono **standardizzati** nell'intervallo 0-1 in modo che la loro somma sia 1.

```
x = sample(1:100, 20)  
w = sample(1:100, 20)  
weighted.mean(x, w)  
## [1] 42.52719  
weighted.mean(x, w/sum(w))  
## [1] 42.52719
```

quantile()

Abbiamo visto che `summary` ci fornisce i **quantili**. Nella statistica descrittiva, i quartili sono quei valori/modalità che ripartiscono la popolazione in quattro parti di uguale numerosità. I quartili sono un gruppo particolare di **quantili**. Il quantile di ordine α o α -quantili (con α un numero reale nell'intervallo $[0,1]$) è un valore q_α che divide la popolazione in due parti, proporzionali ad α e $(1-\alpha)$ e caratterizzate da valori rispettivamente minori e maggiori di q_α . Per poter calcolare un quantile di ordine α è necessario che il carattere sia almeno ordinato, cioè sia possibile definire un ordinamento sulle modalità.

La funzione `quantile` di *R* serve per calcolare i quantili di ordine α per un qualsiasi vettore numerico o di classi ordinali. I suoi argomenti sono `x`, il vettore, e `probs`, cioè le probabilità (α).

```
v = sample(1:100, 20)
quantile(x = v, probs = seq(0, 1, by=0.33))
##      0%      33%      66%      99%
##    2.00 40.54 74.86 96.81
```

In questo esempio, abbiamo diviso i dati in 3 blocchi omogenei. Ovviamente è possibile fare anche “tagli” più eterogenei.

```
v = sample(1:100, 20)
quantile(x = v, probs = c(0, 0.5, 0.7, 0.9))
##      0%      50%      70%      90%
##     4.0 53.0 74.3 87.5
```

sd()

Un'altra funzione statistica di base, che serve a calcolare la *standard deviation* (deviazione standard) di una serie di dati. Si applica esattamente come `mean` o `median`, e come loro ha un'opzione `na.rm`.

```
v = sample(1:100, 20)
sd(x)
## [1] 29.78073
```

runif()* e *rnorm()



Figura 20 - Come chi estrae i numeri dalla ruota del lotto, le funzioni `runif` e `rnorm` pescano per noi valori casuali da distribuzioni note.

R è prima di tutto un ambiente per la statistica applicata. In questo ambito, la possibilità di estrarre valori da una distribuzione nota è molto utile. `runif` e `rnorm` sono solo due tra le tante funzioni che iniziano per `r` e continuano con il nome della distribuzione da cui estrarre valori. Insieme a queste funzioni, e per tutte le principali distribuzioni di dati, ci sono poi anche delle funzioni che iniziano per `d` (calcolo della densità), `p` (calcolo della funzione di ripartizione) e `q` (calcolo della funzione quantilica).

Per brevità esamineremo solo le funzioni normale e uniforme, ma ci sono anche la gamma, la beta, la weibull, ecc.

Gli argomenti necessari per `rnorm` sono:

- `n`, il numero di elementi da estrarre;
- `mean`, il valore medio della normale;
- `sd`, la deviazione standard della normale.

```
vnorm = rnorm(n = 100, mean = 0, sd = 1)
summary(vnorm)
##      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
## -3.51374 -0.62886  0.05198 -0.06996  0.57423  2.60778
```

Gli argomenti necessari per `runif` sono:

- `n`, il numero di elementi da estrarre;
- `min`, il valore minimo;
- `max`, il valore massimo;

```
vunif = runif(n = 100, min = -1, max = 1)
summary(vunif)
##      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
## -0.99482 -0.57387 -0.01916 -0.04102  0.45457  0.99359
```

Permettiamoci una piccolissima anticipazione (incontreremo la funzione per realizzare gli istogrammi tra pochissimo) e creiamo un grafico per confrontare i risultati:

```
vnorm = rnorm(n = 100, mean = 0, sd = 1)
vunif = runif(n = 100, min = -1, max = 1)

par(mfrow = c(1,2))
hist(vnorm, col = "grey", main = "dist normale", cex.axis = 0.7)
hist(vunif, col = "grey", main = "dist uniforme", cex.axis = 0.7)
```

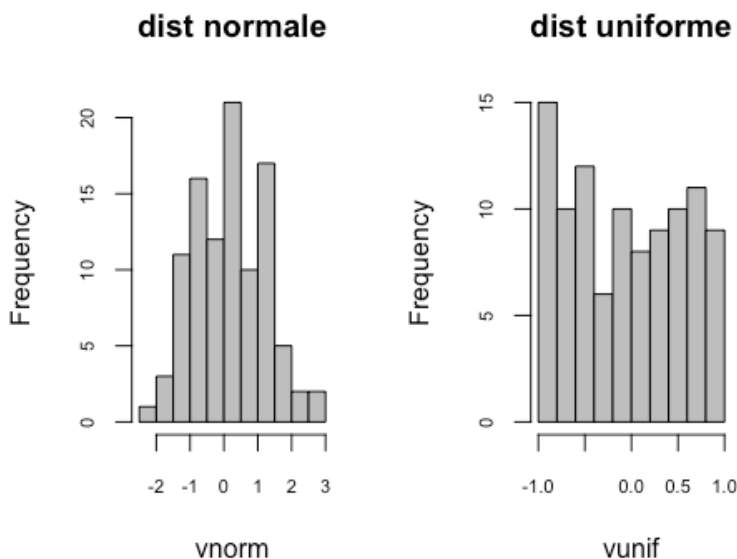


Figura 21 - Un confronto empirico tra una distribuzione normale e una uniforme.

Funzioni per leggere e scrivere (csv, file di testo, ecc.)

Ormai abbiamo imparato a fare tante cose in R con i dati, ma non abbiamo ancora detto come salvarli o importarli. R sa leggere (e salvare) praticamente qualsiasi formato, inclusi quelli *proprietary* come **xlsx** o **docx** o **pdf**. Tuttavia, se R diventerà il vostro strumento preferito, vi potreste accorgere che i formati più pratici ed efficaci sono il vecchio ma intramontabile **csv** comma separated values o un altro formato di testo generico.

In questa introduzione alle funzioni di base ci limiteremo alle funzioni per leggere/scrivere **csv** (`read.csv` e `write.csv`) e **file di testo** (`read.table` e `write.table`). Entrambe le funzioni `read` vogliono:

- l'argomento `file` che corrisponde al nome del *file* da caricare in R, compreso il suo percorso (se non è nella cartella di lavoro della sessione);
- l'opzione `header` che ci permette di dire a R se la prima riga contiene i nomi delle colonne oppure no (*default*);
- `sep`, il separatore delle colonne, che è definito per *default* come spazio bianco (" ") in `read.table` e virgola (",") in `read.csv`;
- `dec`, cioè il carattere che separa la parte intera da quella decimale dei valori numerici reali;

- nel caso delle funzioni `write`, serve anche l'argomento `x` che identifica l'oggetto *R* che deve essere salvato nel `file`.

R ha, comunque, un suo formato molto comodo e potente, perché permette di salvare non solo i dati, ma anche tutte le loro caratteristiche (come la *classe*). I due comandi `readRDS` e `saveRDS`, molto spartani ma efficienti, vogliono al massimo uno o due argomenti:

- `object` cioè l'oggetto di *R* da salvare (nel caso di `saveRDS`);
- `file` il nome del `file`.

Un po' di grafici di base

Per completare la nostra parziale panoramica dei metodi base di *R*, è necessario esaminare le funzioni di base per produrre grafici. Queste funzioni sono estremamente flessibili e, sebbene il loro uso di base sia rapido ed efficace, possono essere utilizzate anche per produrre figure complesse.

Esistono almeno 5 tipologie di grafici di base.

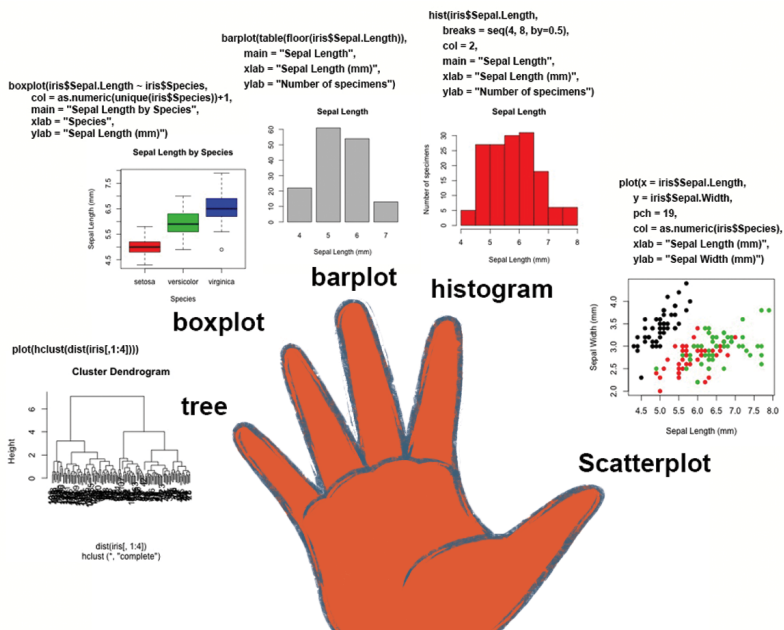


Figura 22 - Ci sono 5 modi fondamentali per produrre un grafico con *R* base.

- gli **scatterplot**, cioè la rappresentazione bidimensionale di un gruppo di osservazioni (es. altezza contro peso, oppure due delle 4 biometrie di *iris*). Questo tipo di grafico è la scelta più naturale quando si vogliono esaminare correlazioni.
- gli **istogrammi**, cioè rappresentazioni della distribuzione dei valori di una certa variabile, tipicamente continua oppure discreta, ma con tante classi ordinali. È la scelta più immediata quando si vuole studiare il comportamento di una certa quantità.
- i **grafici a barre**, che sono simili agli istogrammi ma generalmente sono la loro alternativa quando i dati non sono continui.
- i **box & whisker**, un vero e proprio classico per rappresentare simultaneamente varie distribuzioni di dati che, spesso, non sono normali (non seguono una distribuzione normale).
- i **dendrogrammi**, strumento classico nella rappresentazione gerarchica di gruppi di osservazioni.

plot()



Figura 23 - Ci sono poche funzioni (o metodi) più generici e flessibili di `plot`, un vero artista che sa curare ogni dettaglio.

`plot` è una delle funzioni più generiche di R, tanto che sarebbe più appropriato parlare di un *metodo* invece che di una semplice funzione. Spesso, infatti, pacchetti aggiuntivi di R hanno delle loro

specifiche funzioni `plot`. Questo è possibile – e avviene – senza creare confusione ed equivoci tra funzioni omonime, perché *R* è **capace di riconoscere la classe di ogni oggetto e di applicare su di esso la funzione plot più appropriata**. Vedremo tra poco come `plot`, se riferita ad un albero di classificazione, restituisce un dendrogramma invece che uno scatterplot.

Comunque, andando con ordine, cominciamo dalla funzione `plot` del pacchetto base, cioè appunto quella per rappresentare nuvole di punti in uno spazio bidimensionale. Gli argomenti di `plot` sono `x` (la coordinata sull'asse delle ascisse) e `y` (la coordinata sull'asse delle ordinate). Se le coordinate sono numeriche (interi o reali, non ha importanza) non c'è molto da dire, mentre se sono dei valori ordinati, *R* li tratterà come una sequenza di numeri naturali 1, 2, 3, ... n, dove n rappresenta il numero di livelli del *factor*. Se sono dei *factor* non ordinati... *R* produrrà qualcosa che assomiglia a una tabella di contingenza.

Vediamo degli esempi:

```
data(iris)
iris$color = factor(rep(c("red","blue"), 75), levels =
c("red","blue"))
# Lunghezza contro larghezza dei sepali
plot(iris$Sepal.Length, iris$Sepal.Width)
```

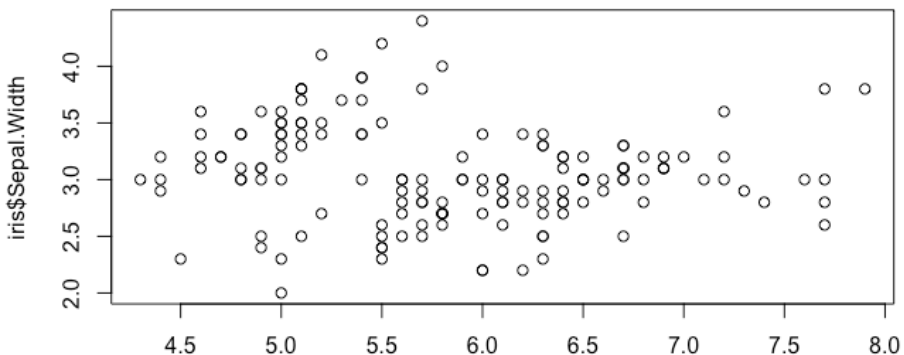


Figura 24 - Lunghezza contro larghezza dei sepali.

```
data(iris)
iris$color = factor(rep(c("red","blue"), 75), levels =
c("red","blue"))
# Lunghezza dei sepali contro specie
plot(iris$Sepal.Length, iris$Species)
```

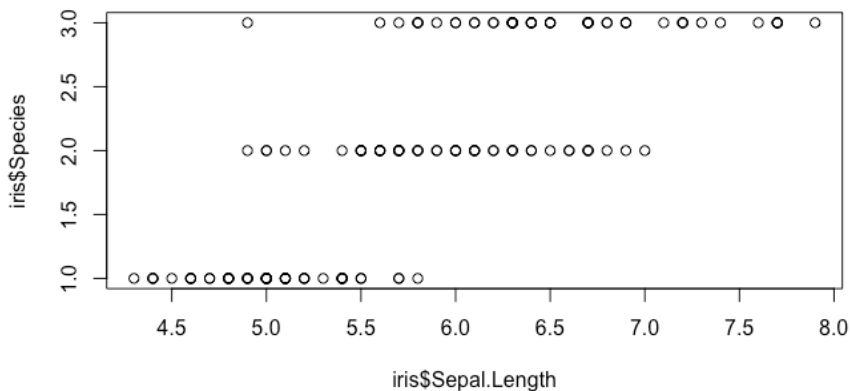


Figura 25 - Lunghezza dei sepali contro specie.

```
data(iris)
iris$color = factor(rep(c("red","blue"), 75), levels =
c("red","blue"))
# Specie contro colori
plot(iris$Species, iris$color)
```



Figura 26 - Specie contro colori.

Oltre a questo, occorre solo sapere che `plot` ha una lunga serie di argomenti opzionali che servono a specificare:

- il tipo di grafico (punti, linee, punti e linee, etc.);
- la forma dei punti;
- il colore dei punti e dello sfondo;
- i titoli degli assi ed il titolo del grafico;
- le dimensioni dei caratteri;
- tanti altri aspetti (margini, ecc.).

Tutti questi argomenti opzionali sono, più propriamente, gli argomenti di un'altra funzione, chiamata *par* (abbreviativo di *parameter*), che si usa per definire questi aspetti **prima** di usare `plot`. Vediamo degli esempi:

```
data(iris)
# pch = 19 permette di usare dei cerchi pieni come simboli nel
# grafico
# (default sono cerchi vuoti)
plot(iris$Sepal.Length, iris$Sepal.Width,
     pch = 19, main = "Esempio di utilizzo dell'argomento pch")
```

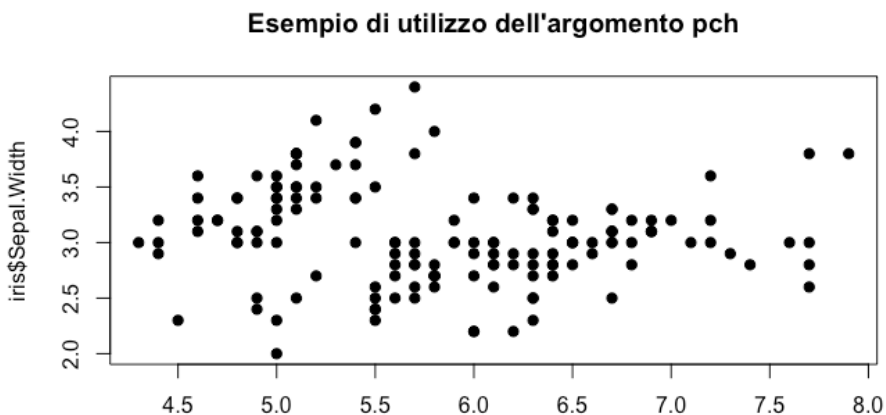


Figura 27 - L'utilizzo dell'argomento pch della funzione plot.

```
plot(iris$Sepal.Length, iris$Sepal.Width,
     pch = 19, col = "red", main = "Esempio di utilizzo dell'argomento col")
```

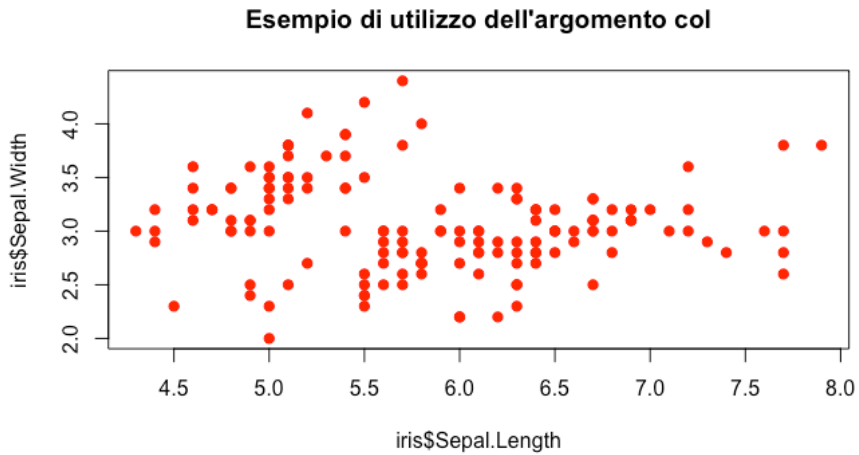


Figura 28 - L'utilizzo dell'argomento col della funzione plot.

```
# type = "b" crea un grafico che mostra sia i punti dei dati  
# che le linee che li collegano.  
plot(iris$Sepal.Length, iris$Sepal.Width,  
      pch = 19, col = "red", type = "b",  
      main = "Esempio di utilizzo dell'argomento type")
```

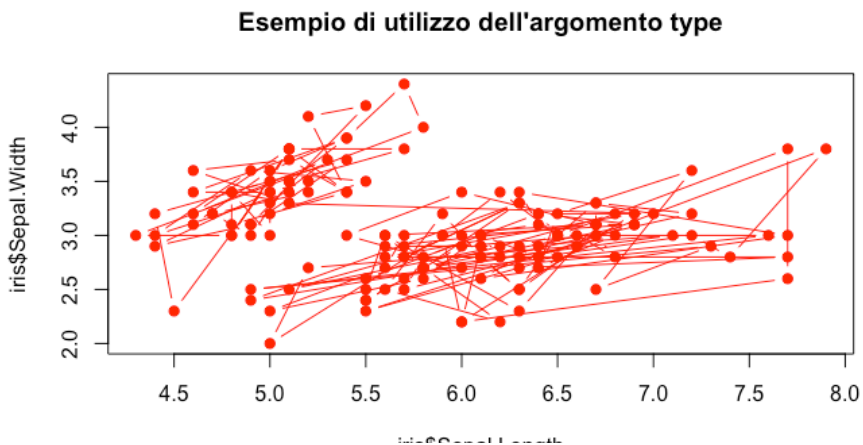


Figura 29 - L'utilizzo dell'argomento type della funzione plot.

*# cex agisce come un fattore di scala, che determina la dimensione
dei simboli rispetto alla dimensione predefinita.*

```
plot(iris$Sepal.Length, iris$Sepal.Width,
     pch = 19, col = "red", cex = 3,
     main = "Esempio di utilizzo dell'argomento cex")
```

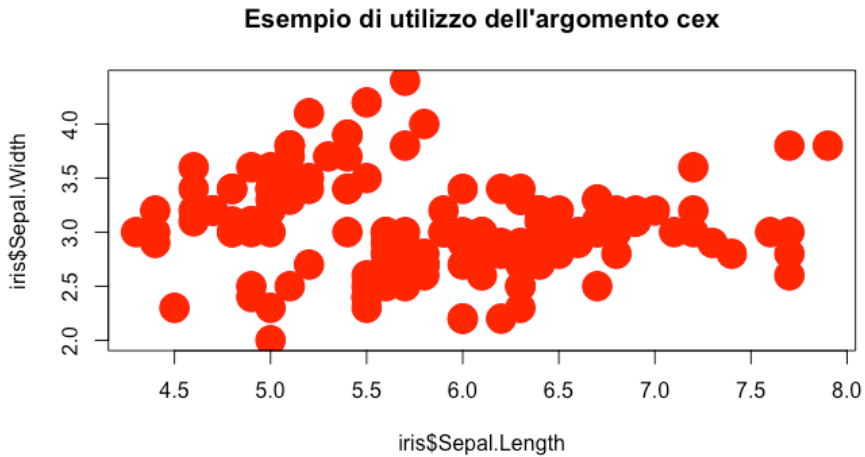


Figura 30 - L'utilizzo dell'argomento cex della funzione plot.

```
plot(iris$Sepal.Length, iris$Sepal.Width,
     pch = 19, col = "red", xlab = "lunghezza dei sepali",
     ylab = "larghezza dei sepali",
     main = "Esempio di utilizzo degli argomenti xlab e ylab")
```

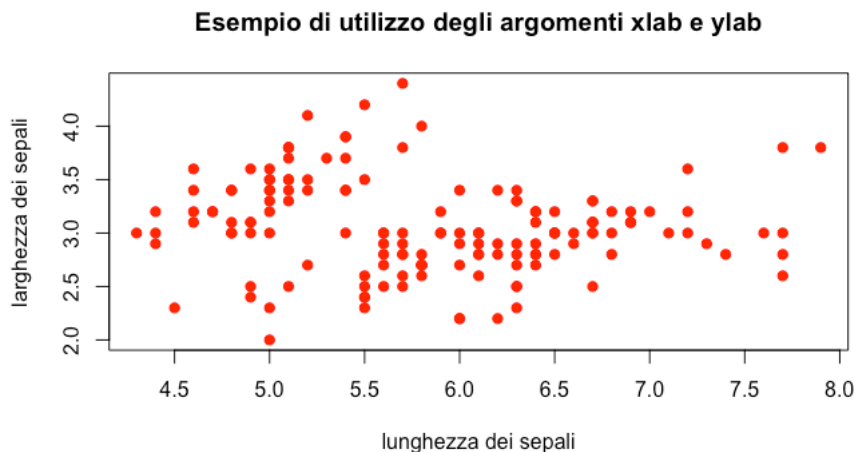


Figura 31 - L'utilizzo degli argomenti xlab e ylab della funzione plot.

```
par(mfrow = c(2,2))
plot(iris$Sepal.Length, iris$Sepal.Width, pch = 19)
plot(iris$Sepal.Length, iris$Petal.Length, pch = 19)
plot(iris$Sepal.Length, iris$Petal.Width, pch = 19)
plot(iris$Sepal.Width, iris$Petal.Width, pch = 19)
```

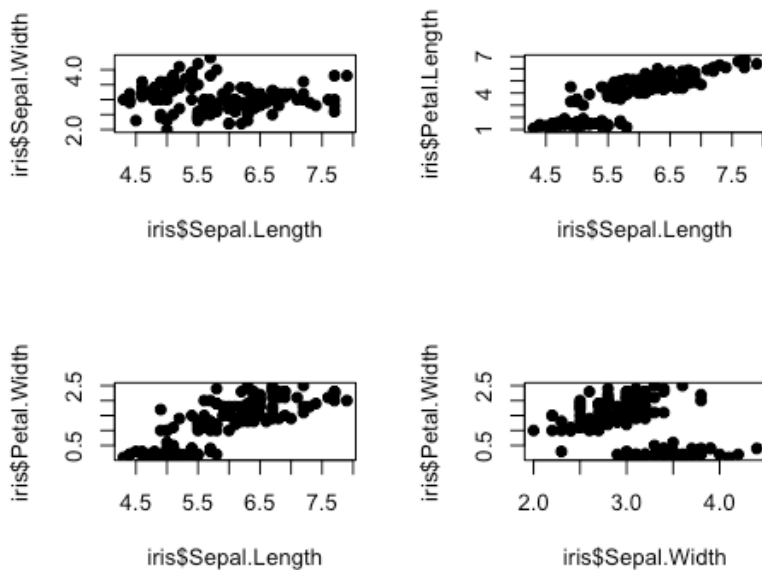


Figura 32 - L'utilizzo della funzione par.

Nell'ultimo esempio abbiamo usato `par` e `mfrow` (un suo argomento) per inizializzare uno "spazio grafico" avente 2 righe e 2 colonne, in modo da ottenere un mosaico di scatterplot perfettamente coerenti dal punto di vista grafico (stesse proporzioni, stessi caratteri, stesso allineamento, ecc.).

Il bravo neofita è invitato a giocare con le (quasi) infinite opzioni di `plot`, che sarebbe noioso esaminare una per una. Piuttosto, è utile insistere sull'"intuito" della funzione `plot`. Vediamo cosa succede se la applichiamo ad un data frame:

```
data(iris)
plot(iris)
```

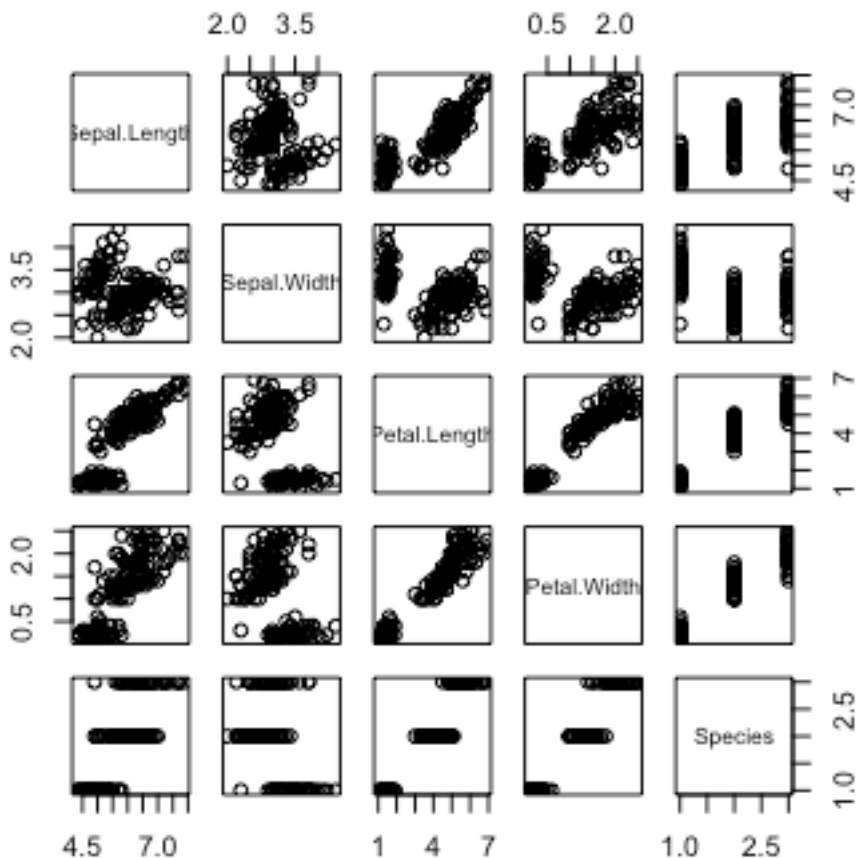


Figura 33 - Il risultato classico che si ottiene usando la funzione `plot` su un data frame.

Il risultato è sorprendente, ma non troppo: *R* è partito dall'assunto che il data frame sia una raccolta di osservazioni (righe) rispetto a 5 variabili... e quindi ci ha restituito uno scatterplot di tutte le possibili coppie di variabili. Un metodo rozzo ma efficace per avere una prima visione d'insieme dei dati (primo passaggio di un qualsiasi protocollo statistico di analisi dei dati). A questo punto viene spontaneo chiedersi come si comporterà plot con oggetti di altri tipo. Vediamo degli esempi:

```
m = matrix(sample(1:100, 30), 10, 3)
plot(m)
```

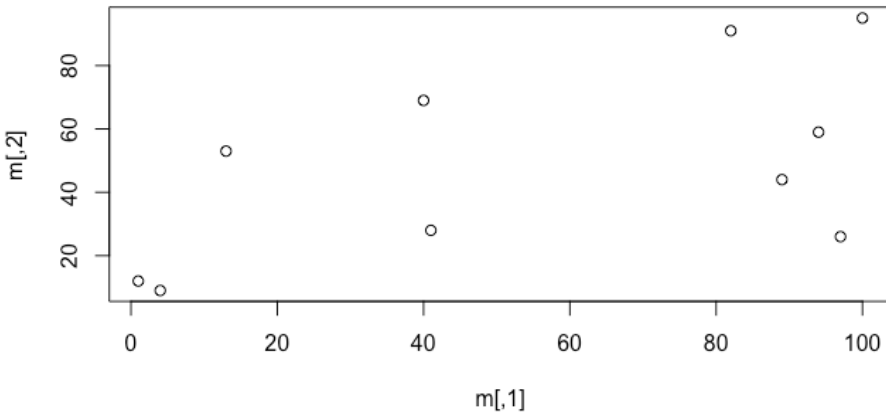


Figura 34 - Il plot di una matrice.

Nel caso di una matrice *m*, *R* ha usato le prime due colonne come *x* e *y*, rispettivamente, ignorando la terza colonna. Ce ne accorgiamo dai nomi dei due assi dello scatterplot.

```
v = sample(1:100, 10)
plot(v)
```

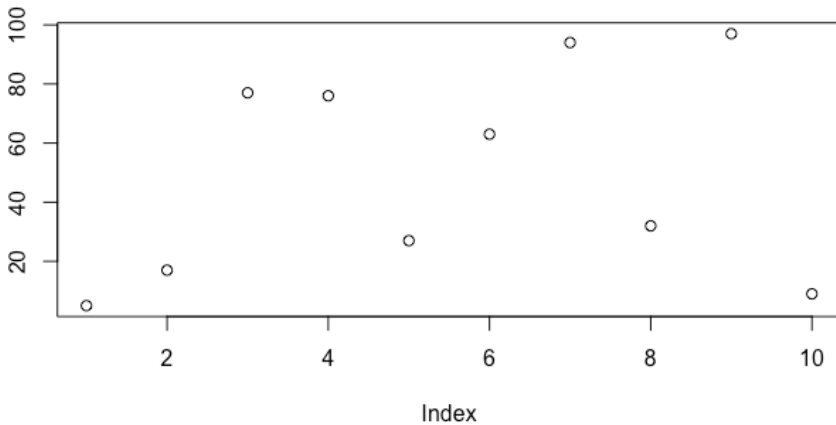


Figura 35 - Il plot di un vettore.

Nel caso di un vettore v , R ha trattato i dati come una serie e ha usato i valori del vettore come y , contro una x rappresentata dagli indici di posizione degli elementi che compongono v .

hist()

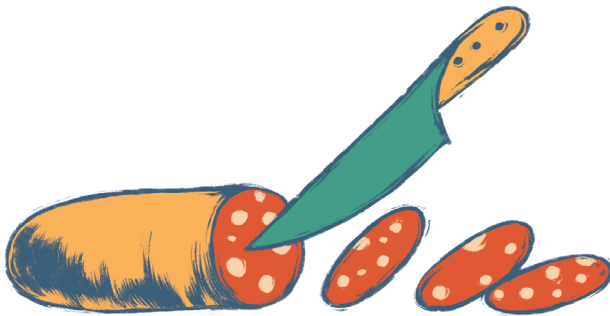


Figura 36 - Un istogramma è uno stimatore di densità che taglia a fette i nostri dati per produrne una rappresentazione grafica.

La funzione `hist` (da non confondere con la quasi omonima `histogram` del pacchetto `lattice`) ha bisogno di due argomenti:

1. `x`, cioè il vettore dei valori di cui si vuole rappresentare la distribuzione;
2. `breaks` (tagli), cioè la serie di intervalli o il criterio per generarli.

Ad esempio:

```
data(iris)
hist(iris$Sepal.Length,breaks = 3)
```

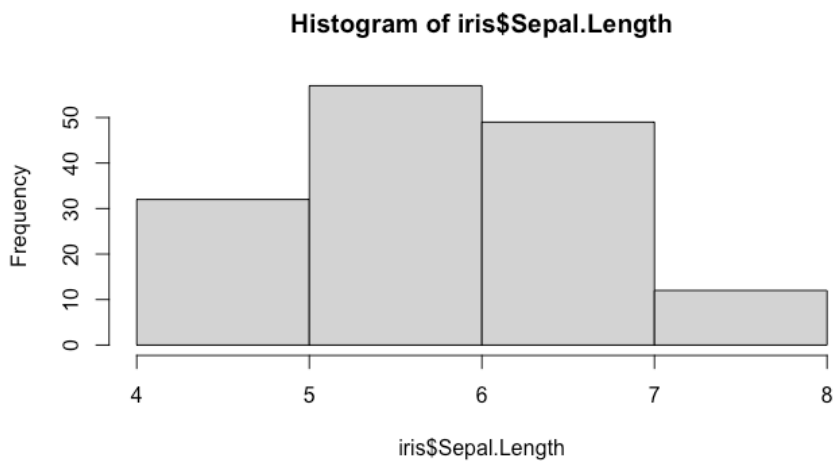


Figura 37 - Un istogramma della lunghezza dei sepali con pochi intervalli.

```
data(iris)
hist(iris$Sepal.Length,breaks = seq(1, 10, by = 1))
```

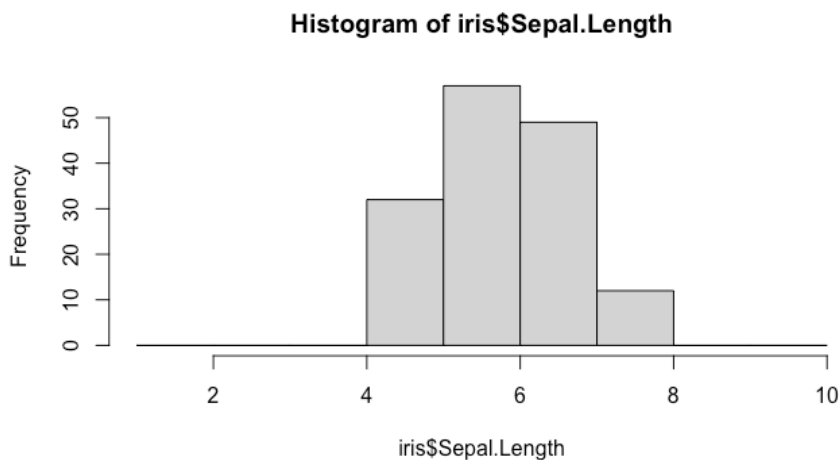


Figura 38 - Un istogramma della lunghezza dei sepali con più intervalli omogenei.

```
data(iris)
hist(iris$Sepal.Length, breaks = c(1,6,7,8))
```

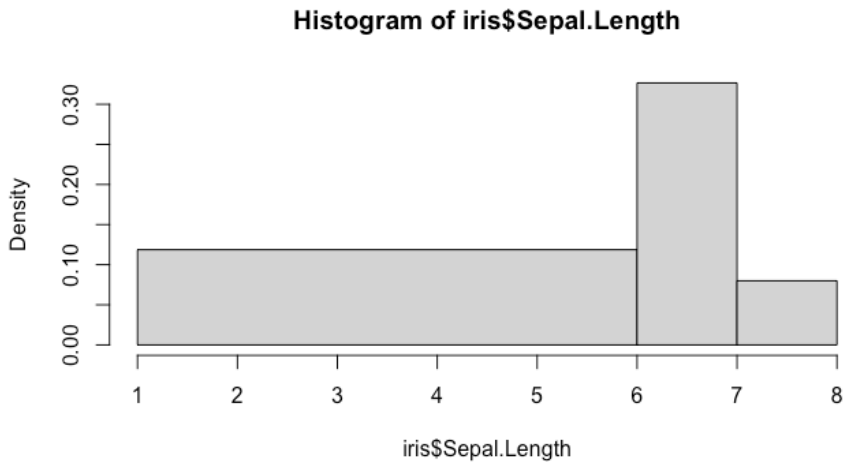


Figura 39 - Un istogramma della lunghezza dei sepali con intervalli disomogenei.

Nel primo caso abbiamo usato `breaks` per dire a *R* di fare 3 “tagli” e quindi generare 4 gruppi di dati. Nel secondo caso abbiamo specificato un vettore di intervalli omogenei. Nel terzo esempio abbiamo definito degli intervalli di ampiezza eterogenea.

Anche `hist` ha diverse opzioni per specificare titoli, colori, ecc.

barplot()

Il grafico a barre è un cugino degli istogrammi ma è usato principalmente per rappresentare il numero di occorrenze rispetto ad una serie di casi. Il suo unico argomento obbligatorio, `height`, è un vettore che rappresenta appunto il numero di osservazioni, qualcosa come l’*output* della funzione `table`. Volendo, si possono indicare i nomi delle classi. Ad esempio:

```
data(iris)
iris$color = factor(sample(c("red", "blue"), 150, rep = T),
                    levels = c("red", "blue"))
```

```
tab_iris = table(iris$color)

barplot(height = tab_iris, names.arg = names(tab_iris))
```

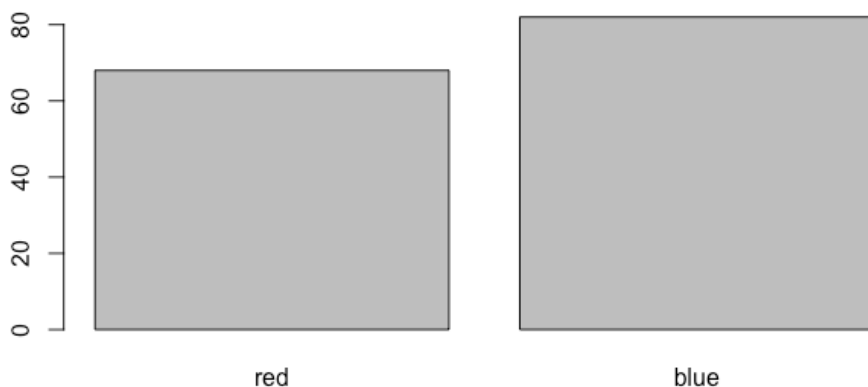


Figura 40 - Grafico a barre delle occorrenze dei colori.

Nell'esempio precedente ci siamo avvalsi della funzione `names` che serve a estrarre i nomi delle classi di una tabella.

boxplot()

I grafici del tipo *Box and Whisker* sono molto utili per confrontare in maniera non parametrica (cioè mediante mediana e quantili) diverse distribuzioni di dati. `boxplot` è una funzione molto generica, un po' come visto per `plot`. Vediamo alcuni esempi della sua applicazione:

```
data(iris)
boxplot(iris$Sepal.Length)
```

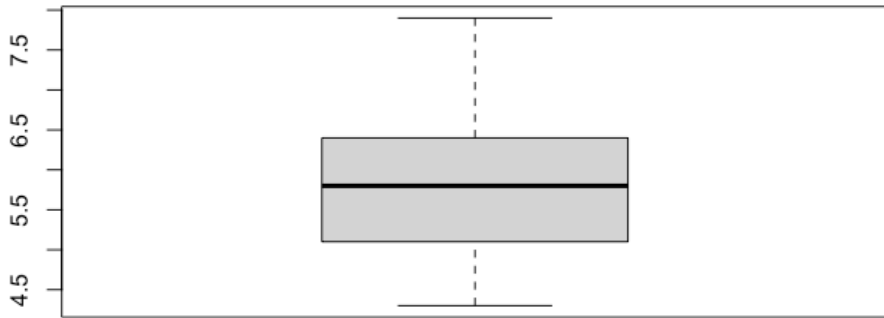


Figura 41 - Boxplot della lunghezza dei sepli.

```
data(iris)
boxplot(iris, cex.axis = 0.7)
```

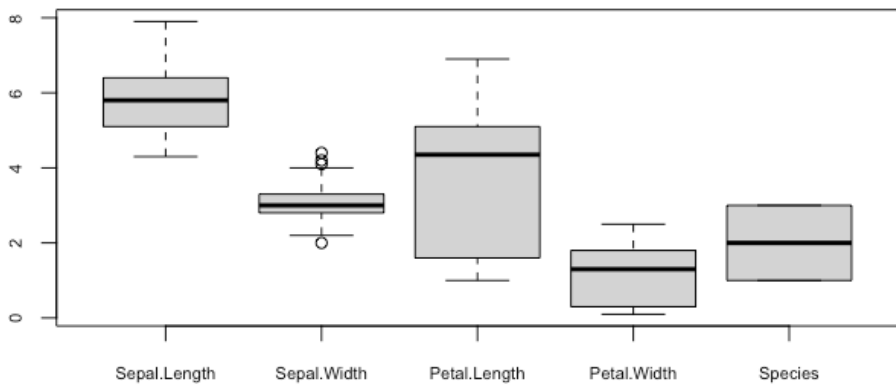


Figura 42 - Boxplot del data frame iris.

```
boxplot(matrix(sample(1:100, 40), 10, 4))
```

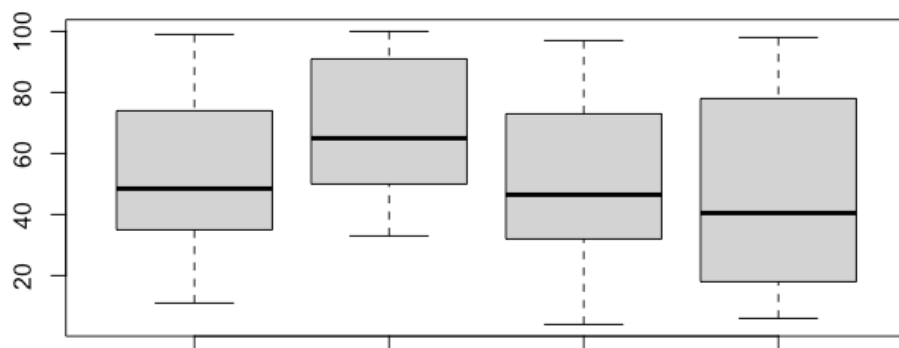


Figura 43- Boxplot di una matrice.

Nel primo caso abbiamo usato una singola colonna di `iris` e abbiamo ottenuto un singolo grafico, in cui gli estremi delle barre rappresentano il minimo e il massimo delle osservazioni, la linea nera spessa rappresenta il valore della mediana e gli estremi del rettangolo grigio rappresentano il secondo ed il terzo quartile, rispettivamente. Questo genere di grafici è molto utile per catturare al volo la *curtosi* e la *simmetria* della distribuzione di una serie di valori.

Nel secondo esempio abbiamo applicato `boxplot` all'intero *dataset* `iris`, ottenendo una figura per ciascuna variabile (colonna) di `iris`. Con la stessa logica si spiega il terzo esempio, in cui R ci restituisce le distribuzioni dei valori di ogni colonna della matrice.

Per completare il discorso su questi grafici, dobbiamo fare conoscenza con la tilde, cioè il simbolo `~`. Questo simbolo si usa, in R, per specificare una *formula*, cioè una relazione matematica di qualche tipo. Nel prosieguo di questo manuale la tilde diventerà di uso sempre più frequente.

Per capirne il significato, partiamo dal modello dell'equazione lineare di una retta: $y = mx + q$. La tilde sostituisce il simbolo `=` in questa formula, ed ha lo stesso senso: mettere in relazione la variabile dipendente con quella(e) indipendente(i).

Adesso vediamo un suo utilizzo:

```
data(iris)
boxplot(iris$Sepal.Length ~ iris$Species)
```

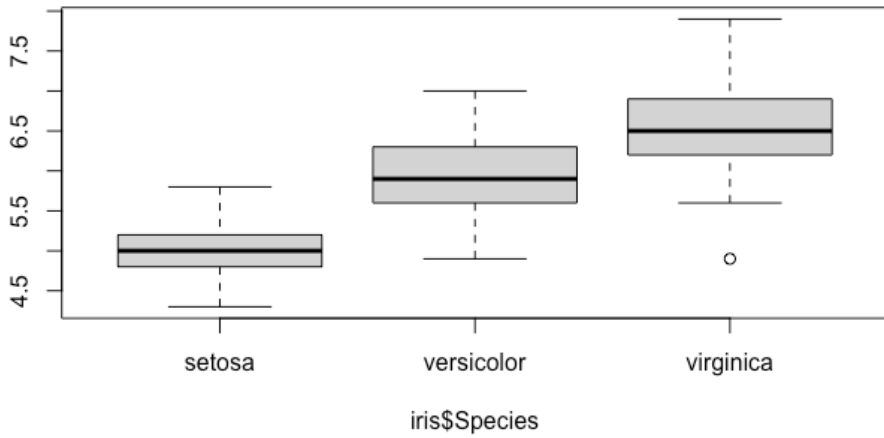


Figura 44 - Boxplot della lunghezza dei sepali per specie.

L'esempio parla da solo: *R* ha rappresentato le distribuzioni della variabile *Sepal Length* **rispetto** alle specie, producendo un grafico molto chiaro che ci suggerisce l'esistenza di una morfologia diversa tra le diverse specie, almeno relativamente a questa singola biometria.

plot()(2)

Non abbiamo ancora introdotto le funzioni necessarie per fare analisi dei dati complesse come la **cluster analysis**:

```
plot(hclust(dist(iris[,1:4])))
```

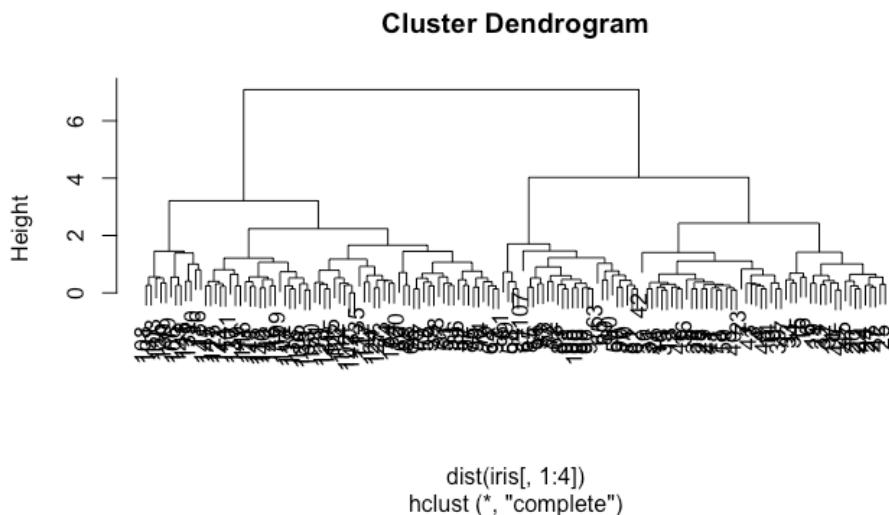


Figura 45 - Plot di un dendrogramma.

In questo esempio abbiamo usato due funzioni nuove, `dist` (che serve a calcolare una matrice di distanze) e `hclust` (che serve a fare le *cluster analysis*) per ottenere una classificazione degli individui nel data frame `iris` in base alle loro biometrie. Poi abbiamo applicato la funzione `plot` sul risultato e abbiamo ottenuto un dendrogramma. Torneremo in un altro capitolo su questo tipo di analisi statistica...

jpeg(), bmp(), png(), tiff() e pdf()

Non ci resta che dire come fare a esportare i grafici (via via più belli) che impareremo a fare con *R*. Tra i tanti vantaggi di *R* c'è la sua capacità di produrre grafici in tanti formati e delle dimensioni e risoluzione desiderati. Tra i più comuni formati ci sono sicuramente:

- il *jpeg* (**J**oint **P**hotographic **E**xperts **G**roup), un comitato di esperti ISO/CCITT che ha definito il primo *standard* internazionale di compressione dell'immagine digitale a tono continuo, sia a livelli di grigio sia a colori, un formato molto comodo perché leggero ma sicuramente non lo standard ottimale per immagini di elevata qualità;
- il *bmp* (**b**itmap), un formato dati utilizzato per la rappresentazione di immagini raster sui sistemi operativi Microsoft Windows, uno dei formati più usati per la grafica raster;
- il *png* (**P**ortable **N**etwork **G**raphics), un formato simile a quello *gif* ma con molte meno limitazioni;
- il *tiff* (**T**agged **I**mage **F**ile **F**ormat), un formato immagine di tipo raster sviluppato dalla Aldus Corporation (oggi Adobe) che è molto flessibile e spesso usato per immagini di elevata qualità;
- il *pdf* (**P**ortable **D**ocument **F**ormat), è un formato di *file* basato su un linguaggio di descrizione di pagina sviluppato da Adobe negli anni '90 per rappresentare documenti di testo e immagini in modo indipendente dall'hardware e dal software utilizzati per generarli o per visualizzarli.
- Tutte queste funzioni si usano nello stesso modo per cui, dopo aver scelto il formato che più ci aggrada, dobbiamo specificare i seguenti argomenti:
 - **filename**: il nome della figura che vogliamo generare. È bene darle un'estensione coerente, ad esempio `.jpeg`;
 - **width** e **height**: la larghezza e l'altezza dell'immagine nell'unità di misura indicata da **units**. Può essere *px* (pixel), *in* (pollici), *cm* o *mm*;
 - la sua risoluzione (**res**), ad esempio 120 dpi.

Per applicare la funzione dobbiamo inserirla nel codice in modo che **preceda** le funzioni che generano il grafico. Dopo la funzione che crea la figura, infine, è necessario specificare il comando `dev(off)`. Questo perché le funzioni come *jpeg* inizializzano un *device*, cioè un sottospazio in

cui viene creata la figura, e per salvarla e chiudere il *device* è necessario appunto specificare `dev(off)`.

Vediamo un esempio:

```
data(iris)
jpeg(filename = "hist.jpeg", width = 1000, height = 600, units =
"px")
  hist(iris$Sepal.Length, breaks = 3)
dev.off()
```

Se controlliamo, la procedura ha salvato nel nostro *hard disk* una figura chiamata `hist.jpeg`, contenente l'istogramma della variabile di `iris Sepal.Length`.

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

2



I modelli lineari

Una delle prime cose che abbiamo detto di R è che si tratta di un ambiente primariamente sviluppato per la statistica. Quindi, dopo aver messo da parte una prima serie di comandi di base, è logico imparare come R può aiutarci a condurre alcune analisi statistiche di base, come ad esempio la stima di modelli lineari.

In generale, la statistica si potrebbe descrivere come l'arte di *generalizzare da informazioni parziali*.

I **modelli statistici** si usano per:

- fare *previsioni*;
- trarre *inferenze* (decidere se una relazione statistica è “significativa”, qualunque cosa significhi).

Perché queste due operazioni sono connesse tra loro?

NOTA

Diciamo che una relazione è “significativa” quando decidiamo di affidarci ad essa per:

- fare delle previsioni;
- sviluppare ulteriori ragionamenti o indagini (per esempio, valutare se un'ipotesi scientifica predice l'osservazione di quella relazione).

Un esempio (molto) stupido



Figura 46 - Immaginiamo di avere di fronte due porte e di osservare attentamente le persone che escono da esse.

Per cominciare esaminiamo un esperimento ideale. Ci sono **due porte**, dalle quali entrano ed escono persone che, però, noi **non vediamo**. Qualcuno ci promette 10 euro se **indoviniamo esattamente quanto è alta la prossima persona che uscirà** da una delle due porte (questo qualcuno sa *esattamente* quanto sono alte tutte le persone che entrano ed escono – con una precisione al millemiliardesimo di centimetro! – non importa come fa a saperlo – ed è *onestissimo*). Naturalmente, indovinare *esattamente* è abbastanza difficile (al millemiliardesimo di centimetro possiamo tranquillamente dire che è impossibile). Per fortuna, il generoso scommettitore ci darà comunque qualcosa se ci avviciniamo *abbastanza*. Diciamo che ci toglie un euro ogni centimetro di errore.

Valore atteso

Tutti noi *abbiamo un'idea* di un valore su cui proveremmo a scommettere. Ma immaginiamo, invece, di non avere *la minima idea* di quanto sia alta una persona (se ci rifiutiamo di immaginare tanta ignoranza, dobbiamo diventare Bayesiani...) e che lo scommettitore ci offra di conoscere **un campione di n misure**.

```
library(car)
library(DescTools)
```

Eccole (la media e la deviazione standard usate per simulare questi dati sono tratte da *qui*):

```
n = 20
mu = 164.7
sigma = 7.07
set.seed(2)
s_1 = rnorm(n = n, mean = mu, sd = sigma)
s_1
## [1] 158.3588 166.0069 175.9261 156.7082 164.1326 165.6362
169.7052 163.0053
## [9] 178.7302 163.7188 167.6528 171.6410 161.9236 157.3495
177.3004 148.3607
## [17] 170.9117 164.9532 171.8607 167.7561
```

Abbiamo incontrato la funzione `rnorm` nel capitolo precedente, ora ne vediamo un utilizzo concreto (per quanto riferito ad un esperimento ideale). Possiamo *visualizzare* le nostre osservazioni come delle *tacche* lungo una scala orizzontale, o meglio, (anche) con un *istogramma*:

```
def_mar = par()$mar
layout(matrix(c(1,2), ncol = 1, byrow = T), heights = c(3,1))
par(mar = c(0,4.1,4.1,2.1))
hist(s_1, xlim = range(s_1) + c(-10,10), col = SetAlpha(4, 0.25),
     xaxt = "n", main = "Our sample")
```

```
par(mar = c(5.1,4.1,0,2.1))
plot(table(s_1), xlim = range(s_1) + c(-10,10),
     xaxt = "n", yaxt = "n", bty = "n", ylab = "", xlab = "height")
axis(side = 1, at = round(seq(min(s_1), max(s_1), by = 10)/10, 0)*10)
```

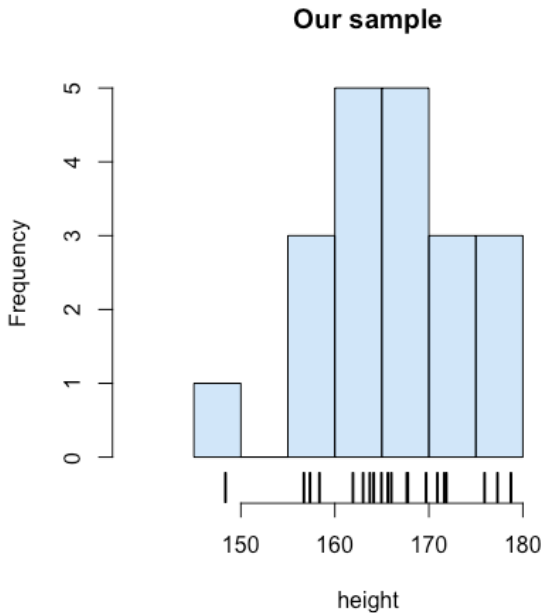


Figura 47 - Distribuzione delle misure di altezza nel campione. Le tacche nere sull'asse orizzontale rappresentano le singole misure. L'istogramma è rappresentato da barre colorate.

```
par(mar = def_mar)
```

Adesso abbiamo delle **informazioni**. E **speriamo** che queste informazioni ci siano utili per fare delle **previsioni**. La nostra “speranza” può essere espressa in termini di **assunzioni**:

la prima, indispensabile, *assunzione*, è che il campione che abbiamo osservato sia **rappresentativo** della **popolazione** delle persone che entrano ed escono (a noi interessano soprattutto quelle che usciranno d’ora in poi);

una seconda *assunzione* che possiamo provare a fare è che la distribuzione delle altezze nella popolazione che ci interessa sia *simmetrica*.

Se entrambe le assunzioni fossero fondate, allora sapremmo che,

scommettendo sul valore **medio** della popolazione che ci interessa, saremmo certi di fare la **migliore scommessa possibile**.

Dalla “**legge dei grandi numeri debole**”, sappiamo anche che **la media di un campione “rappresentativo” è uno “stimatore non distorto” (*unbiased estimator*) della media della popolazione da cui proviene**. Quindi, se crediamo nelle nostre due assunzioni (e nella matematica), possiamo decidere che la nostra **migliore scommessa** non è altro che **la media delle osservazioni** che abbiamo.

```
mu = mean(s_1)
mu
## [1] 166.0819
```

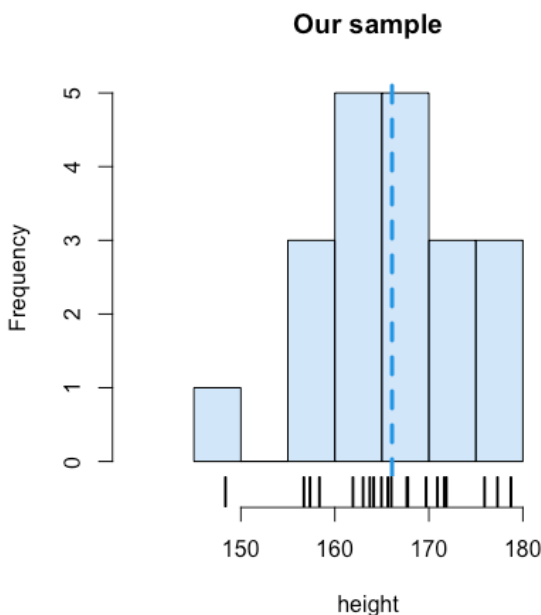


Figura 48 - La linea verticale tratteggiata indica la media delle misure di altezza nel campione.

La **media** rappresenta il **valore atteso** della variabile che ci interessa, **predetto** dal nostro **modello** del fenomeno che stiamo (in modo non del tutto disinteressato) cercando di studiare. Chiamiamo **modello una formula matematica che descrive il modo in cui sono generati i dati che osserviamo** (e ci aspettiamo di osservare).

Residui



Figura 49 - Qualsiasi modello, quando viene confrontato con i dati reali, si lascia dietro dei residui che ci raccontano molto del modello stesso.

Per adesso, il nostro *modello* dice che la migliore scommessa che possiamo fare è 166.1 cm. Ma noi abbiamo già osservato gente *ben più bassa* e *ben più alta*. C'è della **variazione** nei dati che “**resta**” dopo aver stabilito il valore più probabile. Questa è la variazione “residua” del nostro modello. Più precisamente, **le differenze tra ciascuna osservazione e il valore atteso** (nel nostro caso, la media), che possiamo calcolare così...

```
s_1 - mean(s_1)
## [1] -7.72309535 -0.07502577  9.84415698 -9.37366552
-1.94928943
## [6] -0.44569810  3.62333043 -3.07657454 12.64832123
-2.36313368
## [11]  1.57088130  5.55908263 -4.15826567 -8.73236917
11.21844924
## [16] -17.72116793  4.82982488 -1.12875601  5.77878935
1.67420514
```

...si chiamano i “**residui**” del modello.

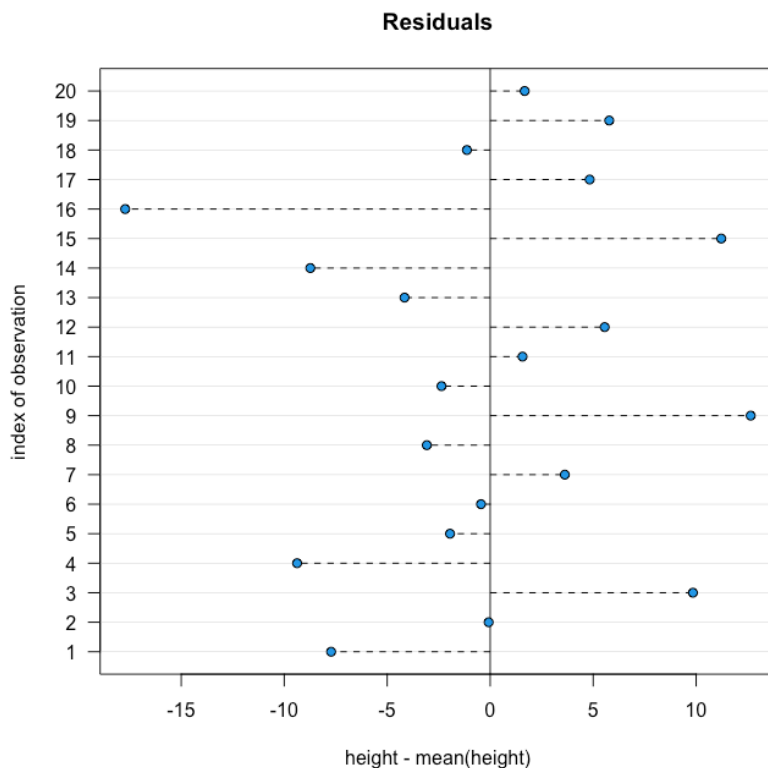


Figura 50 - Una rappresentazione dei residui.

Possiamo visualizzare la loro *distribuzione* con un istogramma:

```
hist(s_1 - mean(s_1), xlab = "height - mean(height)", col =
  SetAlpha(4, 0.25),
  main = "Histogram of residuals")
abline(v = mean(s_1 - mean(s_1)), col = 4, lwd = 3, lty = 2)
```

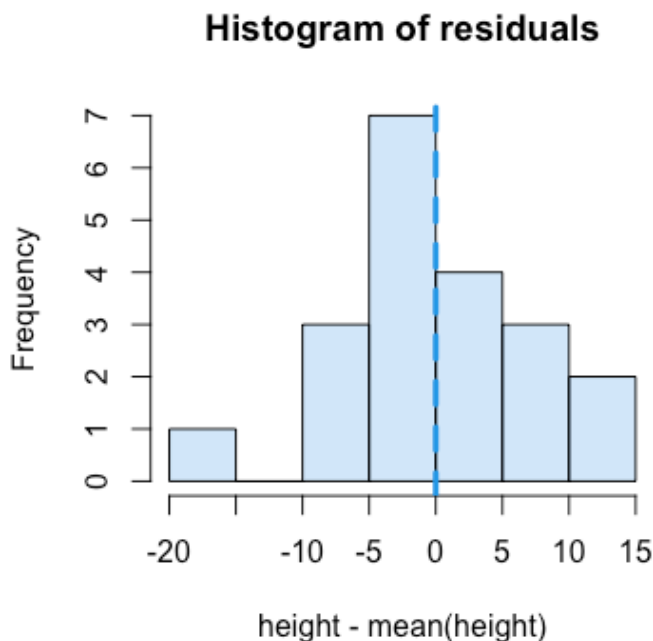


Figura 51 - La distribuzione dei residui. La linea verticale tratteggiata indica la media.

La **media dei residui è praticamente zero**, come è ovvio, visto che si tratta di deviazioni dalla *media* del campione. L'**ampiezza** della distribuzione dei *residui* è una misura dell'**errore** (proprio nel senso etimologico: quanto i dati "errano" attorno alla media). Maggiore l'*errore*, minore la nostra capacità di *predizione*. Un modello con un *errore* piccolo (con una *bassa deviazione standard dei residui*) è un modello che **spiega bene le osservazioni**). Possiamo formulare il nostro modello completo come:

$$H = \mu + \varepsilon$$

dove **H** è l'altezza che stiamo cercando di indovinare, **μ** è il valore atteso (la *media*) e **ε** l'errore (i *residui*). Per poter fare dei calcoli a partire da questa formula, però, abbiamo bisogno di avere un'**equazione** che descriva **come sono distribuiti** i *residui*. Possiamo provare ad **assumere** (ancora) che questi *residui* seguano una **distribuzione normale** di probabilità (se si chiama così, ci sarà un motivo, no?).

$$\frac{1}{\sigma\sqrt{2\pi}} e^{\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

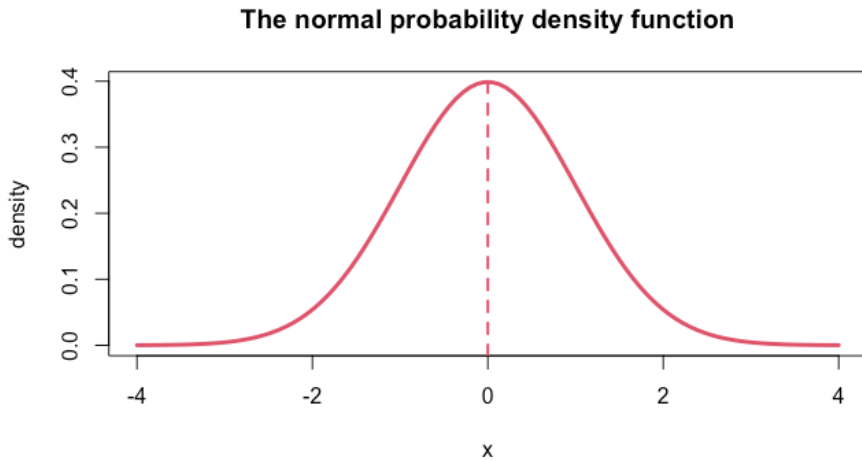


Figura 52 - Densità di probabilità per una distribuzione normale con media 0 e deviazione standard (σ) 1.

Il nostro modello, allora, diventa:

$$H = \mu + \text{Norm}(\theta, \sigma)$$

Sappiamo che una *distribuzione normale* è definita dai parametri **media** μ e **deviazione standard** σ , ma non sappiamo ancora **quanto vale** σ . A quanto pare, stimare correttamente la deviazione standard di una popolazione a partire da un campione non è sempre **banale**, ma, ai nostri fini, basterà la correzione usuale...

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

Questa correzione è applicata dalla funzione base `sd` in R, quindi...

```
sigma = sd(s_1)
sigma
## [1] 7.483983
```

Le nostre assunzioni hanno adesso costruito un **modello probabilistico** della popolazione di partenza. Questo modello ha due **parametri**: μ e σ .

Possiamo **sovrapporre visivamente** il modello ai **dati** che abbiamo osservato.

```
xs = seq(mu-4*sigma, mu+4*sigma, l = 100) # interval over which
our_normal_pdf = dnorm(x = xs, mean = mu, sd = sigma)
par(mfrow = c(1,1))
hist(s_1, xlim = range(s_1)+c(-10,10), col = SetAlpha(4, 0.25),
     main = "Our sample", freq = F, ylim = c(0, 0.06), xlab = "H")
abline(v = mu, col = 4, lwd = 3, lty = 2)
lines(xs, our_normal_pdf, col = 4, lwd = 3)
```

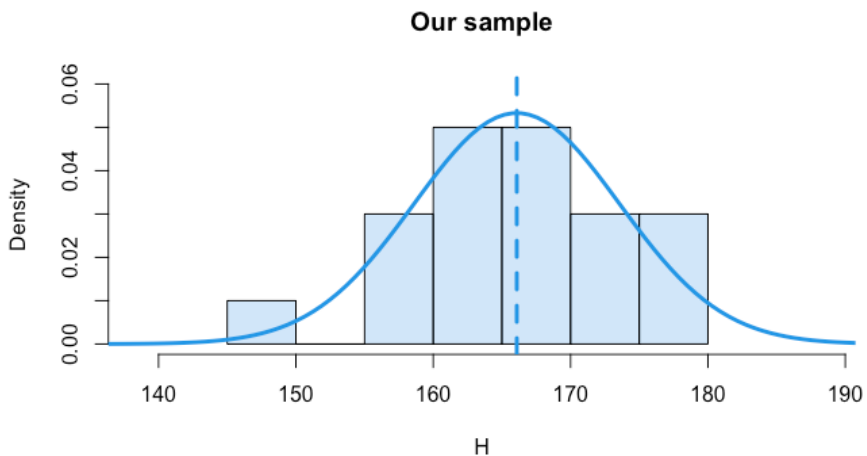


Figura 53 - Confronto tra la distribuzione dei dati (istogramma: barre colorate) e la densità di una distribuzione normale di probabilità con media e deviazione standard pari a quelle del campione (linea continua). La linea verticale tratteggiata indica la media.

Likelihood



Figura 54 - La verosimiglianza è una funzione di probabilità condizionale che ci permette di stimare i parametri di un modello in modo che si adattino al meglio ai dati.

Questo modello ci permette di assegnare una probabilità (**relativa!**) alle osservazioni che abbiamo fatto, **condizionale** al modello e ai valori dei suoi (due) parametri. Per esempio, il primo dato del nostro campione è.

```
s_1[1]
## [1] 158.3588
```

La densità di probabilità per l'osservazione di una persona con questa altezza in una popolazione con media $\mu = 166.1$ e deviazione standard $\sigma = 7.48$ si può calcolare come...

```
dnorm(x = s_1[1], mean = mu, sd = sigma)
## [1] 0.03129916
```

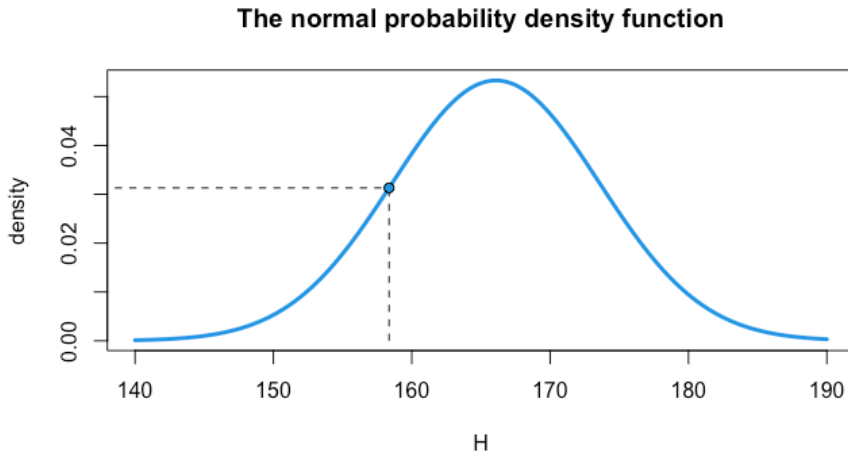


Figura 55 - Densità di una distribuzione normale di probabilità con media e deviazione standard pari a quelle del campione corrispondente al valore di 158.4 cm (linee tratteggiate).

Naturalmente, è lo stesso che...

```
dnorm(x = s_1[1] - mu, mean = 0, sd = sigma)
## [1] 0.03129916
```

La probabilità (relativa) di un'osservazione *condizionale ad un modello e ai suoi parametri* si chiama **likelihood** (in italiano si tradurrebbe *verosimiglianza*, ma usiamo la parola inglese quasi sempre).

Ora... **assumendo** (ancora!) che il nostro campione sia composto da osservazioni che sono tutte **indipendenti** tra loro, possiamo calcolare la *likelihood* dell'**intero campione** semplicemente **moltiplicando tra loro** tutte le *likelihood* delle singole osservazioni...

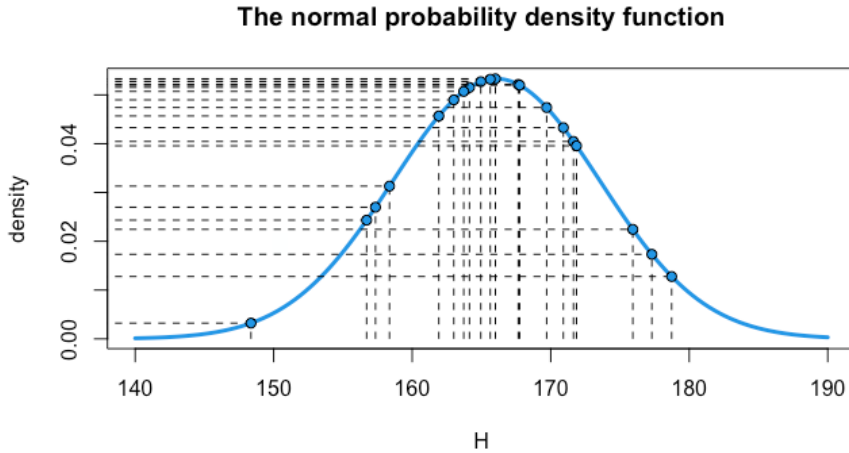


Figura 56 - Una rappresentazione del calcolo della verosimiglianza.

```
prod(dnorm(x = s_1 - mu, mean = 0, sd = sigma))
## [1] 2.568903e-30
```

Il numero è *molto* basso. Per renderlo più trattabile, possiamo usare il suo logaritmo...

```
log(prod(dnorm(x = s_1 - mu, mean = 0, sd = sigma)))
## [1] -68.13407
```

Oppure, in modo equivalente ma più pratico (la somma dei logaritmi è uguale al logaritmo del prodotti)...

```
sum(log(dnorm(x = s_1 - mu, mean = 0, sd = sigma)))
## [1] -68.13407
```


“Fittate” un modello

Ora... è abbastanza intuitivo che, se *invece della media del campione* usassimo *un altro valore qualunque*, la *likelihood* del nostro campione diminuirebbe.

Facciamo un confronto con un tentativo a caso, fissando la media a 160 cm, invece che a 166.0819095 cm)...

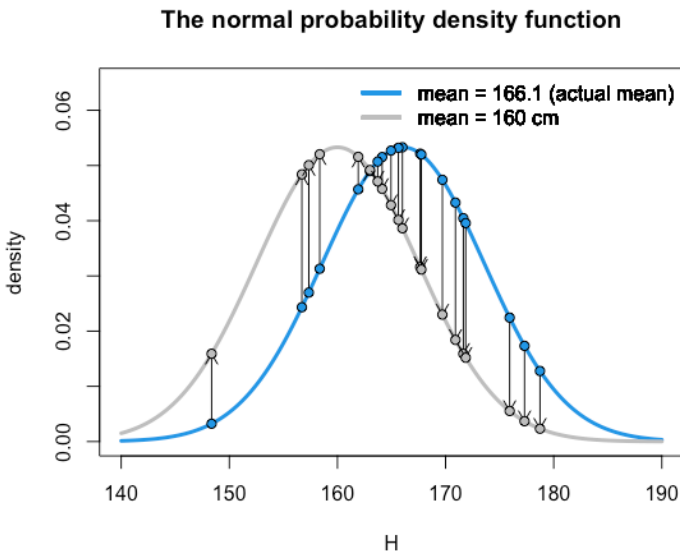


Figura 57 - Stime della verosimiglianza a confronto.

```
sum(log(dnorm(x = s_1 - 160, mean = 0, sd = sigma)))
## [1] -74.73818
```

... mentre prima era...

```
sum(log(dnorm(x = s_1 - mean(s_1), mean = 0, sd = sigma)))
## [1] -68.13407
```

È anche abbastanza intuitivo che il valore che abbiamo scelto come **migliore stima possibile**, cioè quello che **massimizza** la nostra **vincita**

attesa (ovvero – speriamo, con le nostre assunzioni – minimizza la differenza tra la nostra scommessa e le osservazioni che verranno), è anche quello che **massimizza la *likelihood* delle nostre osservazioni**. Per la precisione, la **combinazione** dei parametri μ e σ che abbiamo trovato con le banali funzioni `mean` e `sd` è quella che massimizza la ***likelihood*** delle osservazioni. Si può dire che quei due valori dei parametri corrispondono al miglior **fit** possibile del modello ai dati. In generale, “**fittare**” un modello a un *set* di dati vuol dire **trovare la combinazione di valori dei parametri che massimizza la *likelihood* dei dati a disposizione**.

Il nostro modello consiste semplicemente in una **distribuzione normale di probabilità**. In questo caso, la ***sample mean*** e la ***sample standard deviation*** corrispondono ai valori “ottimali”. Nella maggior parte dei casi, però, la ***maximum likelihood*** non si calcola facilmente, e ci sono molti algoritmi diversi per **approssimarla** (il più semplice, e inefficiente, è andare semplicemente *per tentativi*, esplorando una “griglia” di valori possibili).

Un approccio che possiamo benissimo usare anche nel nostro caso, ma che è (un po’) più generale rispetto al calcolo di `mean` e `sd` per trovare la ***maximum likelihood***, è **minimizzare la somma degli scarti quadratici** (il “metodo dei minimi quadrati”). Questa operazione viene compiuta dalla **funzione `lm`** usando una tecnica che si chiama *QR decomposition* (ma non chiedetemi di più...). Per esempio...

```
fit_1 = lm(formula = s_1 ~ 1)
```

Ecco un primo contatto con la **sintassi** dei ***linear models*** in *R*. Ci siamo finalmente imbattuti con la prima (e più semplice) delle tante funzioni che, in *R*, ci consentono di stimare dei modelli. La studieremo con un certo dettaglio perché **tutte** le funzioni analoghe si comportano (dal punto di vista informatico e della struttura dei risultati che restituiscono) in maniera molto simile:

- Il nome della funzione `lm` è l’acronimo di *linear model*;
- l’argomento principale della funzione `lm` è una *formula* che descrive il modello (in questo caso, è l’unico argomento);
- quando si usa la funzione `lm` non ci si deve preoccupare di descrivere la **distribuzione dei residui**, perché `lm` **assume sempre** una **distribuzione normale**;

- il simbolo $\sim$, e che si chiama **tilde**, nella formula indica che la variabile a sinistra del $\sim$ (nel nostro caso i valori di altezza contenuti nel vettore `s_1`) **dipende** dai termini indicati a destra del $\sim$;
- quando, a destra del $\sim$, compare solo un numero (diverso da zero), la formula indica un modello descritto completamente da un valore atteso (la media) e una deviazione standard dei residui, esattamente come quello che abbiamo considerato fino ad ora (mentre, se c'è uno zero, il modello è "forzato" ad avere una media uguale a zero);
- generalmente (ed è una buona abitudine), tutti i dati che vogliamo usare per stimare il modello possono essere organizzati in un `data.frame` (che qui chiameremo `df` per comodità), e possiamo specificare che ci riferiamo ad esso mediante l'argomento `data` della funzione `lm`, in modo poi da avere comodo accesso a tutte le variabili che ci occorrono;
- se, in modelli più complessi, vogliamo stimare un modello in cui una variabile y è funzione di un certo numero di predittori `x1`, `x2`, `x3`, (senza nominarli uno per uno) contenuti nello stesso data frame, potremmo usare una sintassi del tipo...
-

```
df = data.frame(y = rnorm(100), x1 = rnorm(100), x2 = rnorm(100), x3 = rnorm(100))
fit_x = lm(data = df, formula = y ~ .)
```

- dove il punto indica che y dipende da tutte le altre variabili contenute in `df`.

Quindi, tornando al nostro modello, il comando `lm(s_1 ~ 1)` "fitta" un modello identico al "nostro". In effetti, restituisce **gli stessi risultati** ottenuti in precedenza (più altre cose!).

Possiamo estrarre le stime di μ e σ usando l'**ubiqua funzione** `summary`. La funzione `summary`, che in realtà è un metodo (cioè una funzione che si può applicare in tanti contesti e su tanti oggetti diversi in R), quando applicata a un modello lineare restituisce una lista che contiene una serie di *slot*, corrispondenti a tutti gli elementi del modello, compresi la formula e i dati utilizzati per fittarlo ed una serie di valori che descrivono il *fit* del modello.

```
summary(fit_1)
##
## Call:
## lm(formula = s_1 ~ 1)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -17.7212  -3.3470  -0.2604   5.0121  12.6483
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  166.082      1.673   99.24  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.484 on 19 degrees of freedom
```

Una cosa utile da sapere è che possiamo accedere direttamente a ciascuno degli *slot* di questa lista usando il solito simbolo \$:

```
mu_1 = summary(fit_1)$coefficients[, "Estimate"]
sigma_1 = summary(fit_1)$sigma
```

...e confrontare con i valori di *mu* e *sigma* ottenuti usando le funzioni *mean* e *sd*:

```
mu_1
## [1] 166.0819
mu
## [1] 166.0819
sigma_1
## [1] 7.483983
sigma
## [1] 7.483983
```

Confronto tra modelli

Un modello lineare “fittato” a un campione di dati può essere usato per testare la “**significatività**” di una **relazione statistica** tra una variabile di interesse (**risposta**) e un'altra variabile (**predittore**). Immaginiamo di poter osservare **un altro campione** simile al precedente. Immaginiamo, per esempio, che, mentre il primo campione era composto da gente che usciva dalla porta **destra**, il secondo sia composto da gente che esce dalla porta **sinistra**.
(Per sicurezza, ricreiamo anche il primo campione da capo...)

```
n = 20
mu = 164.7
sigma = 7.07
set.seed(2)
s_1 = rnorm(n = n, mean = mu, sd = sigma)
s_2 = rnorm(n = n, mean = mu, sd = sigma)
```

Calcoliamo il *fit* del nostro modello “normale” al vecchio ed al nuovo campione...

```
fit_1 = lm(s_1 ~ 1)
fit_2 = lm(s_2 ~ 1)
```

(Notate che non c'è bisogno di scrivere `formula =`, la funzione `lm` assume che, se non si specifica altrimenti, il primo argomento è la formula).
... ed estraiamo i parametri...

```
mu_1 = summary(fit_1)$coefficients[, "Estimate"]
sigma_1 = summary(fit_1)$sigma
mu_2 = summary(fit_2)$coefficients[, "Estimate"]
sigma_2 = summary(fit_2)$sigma
```

Plottiamo tutto ...

```
xs = seq(mu_2-4*sigma_2, mu_2+4*sigma_2, l = 100) # interval..
normal_pd_1 = dnorm(x = xs, mean = mu_1, sd = sigma_1)
```

```

normal_pd_2 = dnorm(x = xs, mean = mu_2, sd = sigma_2)
par(mfrow = c(2,1))
hist(s_1, xlim = range(c(s_1, s_2))+c(-10,10), col = SetAlpha(4, 0.25),
     freq = F, ylim = c(0, 0.06), xlab = "H")
abline(v = mu_1, col = 4, lwd = 3, lty = 2)
lines(xs, normal_pd_1, col = 4, lwd = 3)
hist(s_2, xlim = range(c(s_1, s_2))+c(-10,10), col = SetAlpha(2, 0.25),
     freq = F, ylim = c(0, 0.06), xlab = "H")
abline(v = mu_2, col = 2, lwd = 3, lty = 2)
lines(xs, normal_pd_2, col = 2, lwd = 3)

```

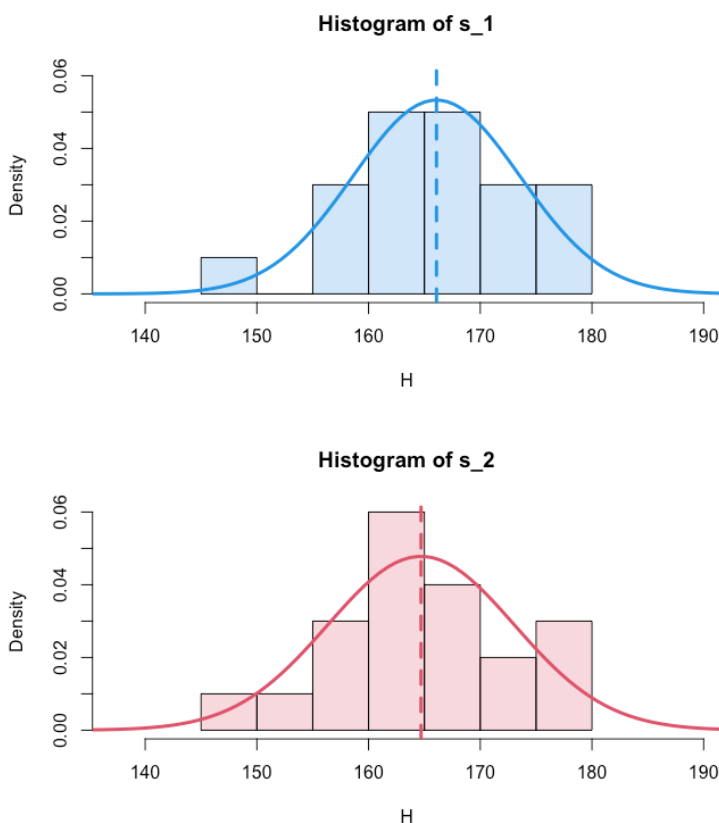


Figura 58 - Istogrammi di due campioni della stessa popolazione e densità di probabilità calcolata dai modelli 'fittati' a ciascun campione.

Meglio sovrapposti?

```

par(mfrow = c(1,1))
hist(s_1, xlim = range(c(s_1, s_2))+c(-10,10), col = SetAlpha(4, 0.25),
     freq = F, ylim = c(0, 0.06), xlab = "H", main = "Istogramma di
s_1 e s_2")
abline(v = mu_1, col = 4, lwd = 3, lty = 2)
lines(xs, normal_pd_1, col = 4, lwd = 3)
hist(s_2, xlim = range(c(s_1, s_2))+c(-10,10), col = SetAlpha(2, 0.25),
     freq = F, ylim = c(0, 0.06), xlab = "H", add = T)
abline(v = mu_2, col = 2, lwd = 3, lty = 2)
lines(xs, normal_pd_2, col = 2, lwd = 3)

```

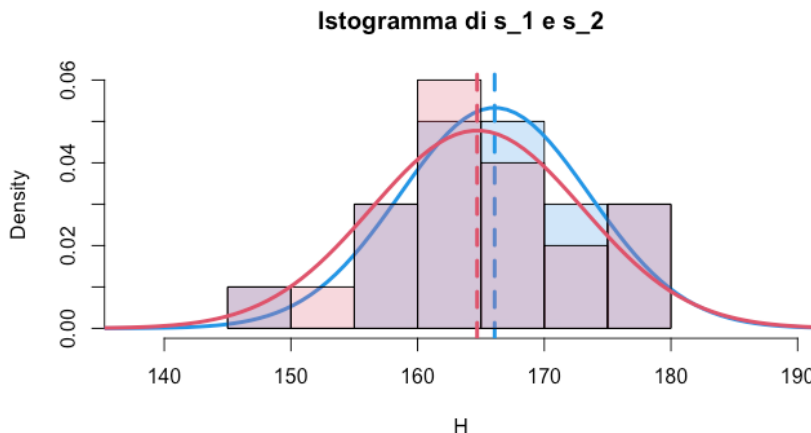


Figura 59 - Confronto tra due modelli 'fittati' a due campioni della stessa popolazione.

I due *fit* (così come i due campioni) sono molto simili, ma ci sono delle piccole differenze (ovviamente!)...

Calcoliamo la *likelihood* di **ciascun campione condizionale al proprio modello!**

```

logLik(fit_1)
## 'log Lik.' -68.12114 (df=2)
logLik(fit_2)
## 'log Lik.' -70.30128 (df=2)

```

Usando la funzione `logLik` siamo più veloci (il risultato non è *esattamente* identico a quello ottenuto "a mano" con `dnorm`, ma è solo questione

di approssimazioni). df è il numero di parametri del modello di cui si calcola la *likelihood* (naturalmente, è «la *likelihood* dei dati sui quali è stato “fittato” il modello condizionale al modello stesso»).

Calcoliamo la *likelihood* dei due campioni **insieme**, ma ammettendo che **ognuno provenga da una popolazione diversa** (cioè, **ciascuno secondo il proprio modello**!).

Possiamo farlo, semplicemente, moltiplicando tra loro le *likelihood* dei due campioni relative, ciascuna, al proprio modello (ma siccome questi sono i *logaritmi*, li sommiamo, invece che moltiplicarli).

```
logLik(fit_1) + logLik(fit_2)
## 'log Lik.' -138.4224 (df=2)
```

Adesso calcoliamo la *likelihood* dei due campioni **insieme**, ma **assumendo che entrambi provengano da una stessa popolazione** (cioè, **entrambi secondo lo stesso modello**)!...

“Fittiamo” un modello che *assume una sola media e una sola deviazione standard* all’*insieme dei due campioni* (che “concateniamo” tra loro usando la funzione `c`)...

```
fit_all = lm(c(s_1, s_2) ~ 1)
```

... e calcoliamone la *likelihood*...

```
logLik(fit_all)
## 'log Lik.' -138.7008 (df=2)
```

La *likelihood* è **più piccola**! Sarà un caso? Facciamolo **1000 volte**...

Lo *script* nel *chunk* che segue:

- parte da una popolazione descritta dalle solite media e deviazione standard e poi,
- per `n_rep = 1000` volte,
- ne estrae due campioni (`s_1` e `s_2`) usando la funzione `rnorm`;
- calcola il *fit* di due modelli “campione-specifici” (`fit_1` e `fit_2`);
- calcola il *fit* di un modello “globale” (`fit_all`);
- calcola la differenza tra le (*log*) *likelihood* dell’insieme dei dati condi-

zionale al modello “globale” (modello a *una popolazione*) e all'insieme dei due modelli (*modello a due popolazioni*);

- raccoglie le 1000 differenze di (*log*) *likelihood* in un vettore chiamato `logLik_diff`;
- genera un istogramma per visualizzare la distribuzione di `logLik_diff`.

```
n = 20
mu = 164.7
sigma = 7.07
set.seed(0)
n_rep = 1000
logLik_diff = rep(NA, n_rep)
for(i in 1:n_rep) {
  s_1 = rnorm(n = n, mean = mu, sd = sigma)
  s_2 = rnorm(n = n, mean = mu, sd = sigma)
  fit_1 = lm(s_1 ~ 1)
  fit_2 = lm(s_2 ~ 1)
  fit_all = lm(c(s_1, s_2) ~ 1)
  logLik_diff[i] = logLik(fit_all) - (logLik(fit_1) + logLik(fit_2))
}
hist(logLik_diff, col = 4, breaks = 25)
```

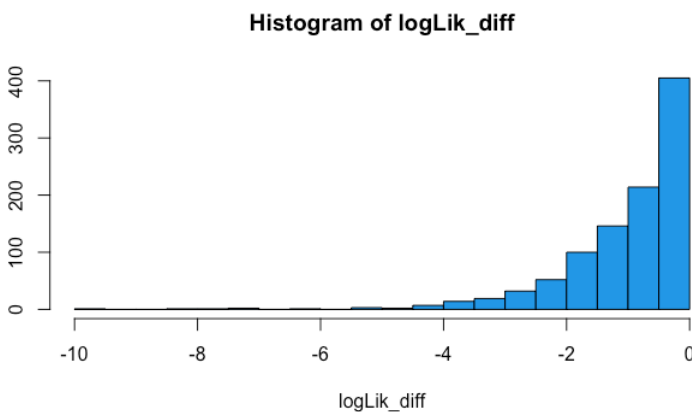


Figura 60 - La distribuzione delle differenze di LogLikelihood tra i due modelli.

La (*log*) *likelihood* del modello “globale” è *sempre* più piccola!

Modelli più o meno semplici

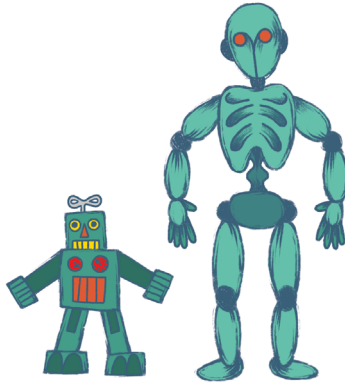


Figura 61 - Modelli più o meno complessi sono come robot diversi: possono contenere pochi o molti componenti (ma questo non implica necessariamente che il robot più complesso sia migliore di quello semplice nello svolgere un determinato compito).

Quando abbiamo calcolato la *likelihood* dell'**insieme dei dati** usando due modelli separati, abbiamo, di fatto, creato **un nuovo modello** nel quale i dati provengono da **due popolazioni**, ognuna caratterizzata dai *suoi* parametri μ e σ .

In questo modello, **quando sappiamo che la persona è uscita dalla porta *destra***, vale l'equazione...

$$H = \mu_1 + \text{Norm}(\theta, \sigma_1)$$

Mentre **quando sappiamo che la persona è uscita dalla porta *sinistra***, vale l'equazione...

$$H = \mu_2 + \text{Norm}(\theta, \sigma_2)$$

Potremmo creare una nuova variabile D (per *door*), e descrivere l'equazione del modello come...

```
if (D = right) → H =  $\mu_1$  + Norm ( $\theta$ ,  $\sigma_1$ )
if (D = left) → H =  $\mu_2$  + Norm ( $\theta$ ,  $\sigma_2$ )
```

Non credo che sia una notazione matematica molto corretta...

Comunque, **una soluzione più elegante** potrebbe essere dire che la variabile D ...

- vale 0 se la porta è "destra";
 - vale 1 se la porta è "sinistra";
- (ovviamente, è arbitrario quale porta vale 0 quale vale 1).
A quel punto, il nostro modello si potrebbe scrivere...

$$H = \mu_1 + DC_\mu + \text{Norm}(\theta, \sigma_1 + DC_\sigma)$$

in cui C_μ è la **differenza tra le medie dei due campioni**, e C_σ è la **differenza tra le deviazioni standard**, per cui...

$$\mu_1 + C_\mu = \mu_2$$

$$\sigma_1 + C_\sigma = \sigma_2$$

e, come sopra, quando sappiamo che la persona è uscita dalla porta *destra*, vale l'equazione...

$$H = \mu_1 + \text{Norm}(\theta, \sigma_1)$$

Mentre quando sappiamo che la persona è uscita dalla porta sinistra, vale l'equazione...

$$H = \mu_2 + \text{Norm}(\theta, \sigma_2)$$

Ah-ah...

Confrontiamo ora questo modello con un modello in cui, dalle due porte, **escono persone che appartengono alla stessa popolazione**.

Questo modello si potrebbe scrivere semplicemente...

$$H = \mu_1 + \text{Norm}(\theta, \sigma)$$

e **diventa del tutto equivalente** al modello:

$$H = \mu_1 + DC_\mu + \text{Norm}(\theta, \sigma_1 + DC_\sigma)$$

quando $C_\mu = 0$ e $C_\sigma = 0$ (scrivere μ_1 invece che μ e σ_1 invece che σ è pura convenzione).

NOTA BENE! Da quello che abbiamo appena detto, si capisce che l'**affermazione**

$$C_\mu = 0 \text{ e } C_\sigma = 0$$

e l'**affermazione**

Dalle due porte escono persone appartenenti alla stessa popolazione (almeno per quanto riguarda l'altezza)

sono **equivalenti!**

Quindi:

- i due modelli sono semplicemente uno (quello con *due popolazioni*) una "**complicazione**" dell'altro (quello con *una sola popolazione*);
- il modello *più complesso* ha **due parametri (non-zero) in più** del modello *più semplice*;
- dire che un parametro **è uguale a zero** è come dire che **non c'è**.

E, soprattutto:

- la *likelihood* di un modello **più complesso** è **sempre maggiore** della *likelihood* di un modello **più semplice!**

Se ci pensiamo, quest'ultima conclusione è abbastanza **intuitiva**: le medie (e anche le deviazioni standard) di due campioni, per quanto *rappresentativi* della **stessa popolazione**, non saranno mai **esattamente identiche!** Sarà *sempre* più probabile osservare **quel preciso campione** calcolando in base a un modello che è stato "cucito addosso" a quel particolare *set* di dati!

Il problema è che...

- Se basiamo le nostre *previsioni* su un modello **inutilmente complesso**, non guadagniamo in precisione, anzi, ne perdiamo!
- Se siamo interessati a *sapere se* le popolazioni che escono dalle due porte **sono effettivamente diverse** (*inferenza*), non possiamo fidarci della *likelihood*!

Likelihood ratio test

Adesso proviamo a prendere la differenza tra le (*log*) *likelihood* dei due modelli calcolate nelle mille simulazioni di prima (il vettore `logLik_diff`), e ci applichiamo **un piccolissimo trucco**: le moltiplichiamo *tutte* per -2... e plottiamo di nuovo la distribuzione in un istogramma.

```
hist(-2*logLik_diff, col = 4, breaks = 25, freq = F)
```

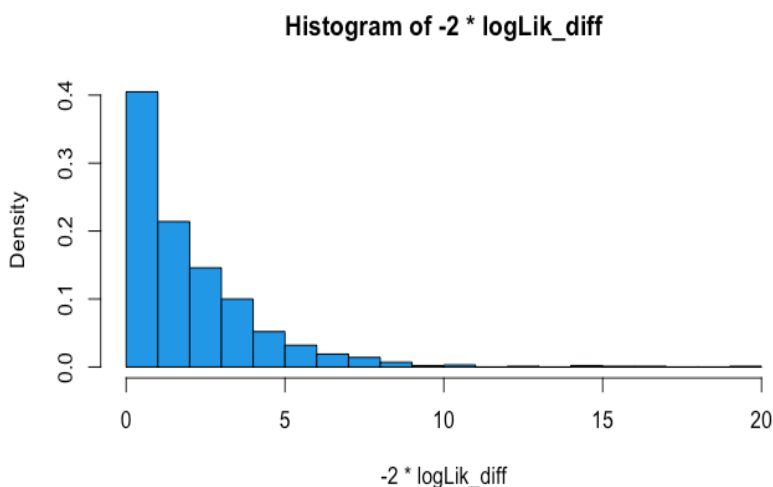


Figura 62 - Istogramma ottenuto dopo la trasformazione dei dati.

Bene... Poi, colti da un **improvviso istinto**, proviamo a **sovrapporre** alla distribuzione *empirica* di `-2*logLik_diff` ottenuta dalla simulazione una **ben nota distribuzione di probabilità** (χ^2)...

La distribuzione di probabilità χ^2 ha un solo parametro, che si chiama **k**, che è anche detto “**gradi di libertà**” (*degrees of freedom, df*)... Sempre *per istinto*, decidiamo di scegliere **df = 2**, proprio come **il numero di parametri di differenza tra il nostro modello “complesso” e quello “semplice”** (ricordiamo che il modello “a due popolazioni” ha due μ e due σ , quello “a una popolazione” ha un solo μ e un solo σ)...

```
hist(-2*logLik_diff, col = 4, breaks = 25, freq = F)
```

```
xs = seq(0, 20, l = 100)
lines(xs, dchisq(xs, df = 2), col = 7, lwd = 3)
```

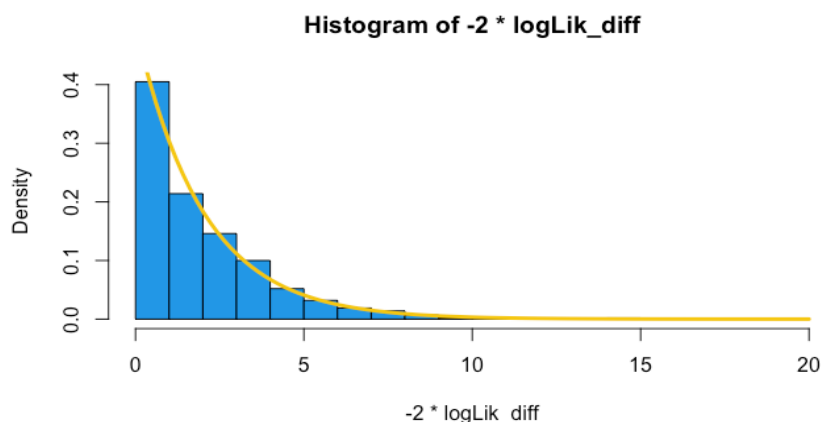


Figura 63 - Confronto tra la distribuzione simulata delle differenze di likelihood tra i due modelli e la densità di probabilità di una distribuzione χ^2 con $df = 2$.

Calza come un guanto!

Facciamo il riassunto della situazione...

- il modello che ha **generato effettivamente** i dati è un modello con **una sola popolazione** (lo sappiamo perché i dati li abbiamo simulati noi!);
- ci sono **due parametri in più** nel modello più complesso (un μ e un σ);
- la distribuzione di $-2 * \logLik_diff$ corrisponde (più o meno) ad una distribuzione di probabilità χ^2 con $df = 2$!!

Ecco... pare che questa ultima cosa **non sia un caso**! In effetti, il **teorema di Wilks** dice (circa) che...

Quando ci sono **due modelli** M_1 e M_2 che sono "nidificati" (**nested**: ovvero nei quali M_2 contiene *tutti i parametri* di M_1 e qualcuno *in più*), la distribuzione della quantità

$$\lambda_{LR} = -2 * \log \frac{P(D|M_1)}{P(D|M_2)}$$

tende a $\chi^2_{df=n}$, dove n è la **differenza tra il numero di parametri di M_2 e il numero di parametri di M_1** .

$P(D|M_1)$ e $P(D|M_2)$ sono le *probabilità condizionali* di uno **stesso set di dati D** rispetto ai modelli M_1 e M_2 , quindi sono le *likelihood* dei dati rispetto a quei modelli, che possiamo chiamare anche L_1 e L_2 .

Quindi L_1/L_2 è il rapporto tra le likelihood (**likelihood ratio, LR**) dei due modelli.

Naturalmente, per le proprietà dei logaritmi:

$$\log \left(\frac{L_1}{L_2} \right) = \log (L_1) - (L_2)$$

Che possiamo farci con questa scoperta?!!

In questo caso, possiamo calcolare la **probabilità di ottenere i dati che abbiamo ottenuto se, ad essere vera, è un'ipotesi più semplice (una sola popolazione), rispetto ad un'altra più complessa (due popolazioni diverse)**, sulla base del valore di λ_{LR} (detto anche $-2 \cdot \log \text{Lik_diff}$)!!

Per esempio, se **immaginiamo** di osservare un valore $\lambda_{LR} = 5.2$ con $df = 2$, la probabilità di osservare $\lambda_{LR} > 5.2$, se il modello più semplice è quello che ha generato i dati, è data da...

```
lambda = 5.2; df = 2
pchisq(lambda, df, lower.tail = F)
## [1] 0.07427358
```

The Chisquare probability density distribution

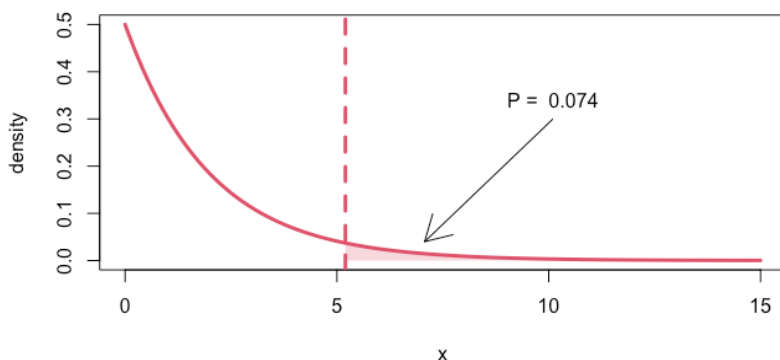


Figura 64 - Probabilità di ottenere un valore maggiore di un valore dato (in questo caso 5.2) assumendo la densità di probabilità con distribuzione χ^2 con $df = 2$. La probabilità $\sqrt{p}$ visualizzabile come l'area sottostante alla porzione di curva per $x > 5.2$ (area colorata).

In questo caso, abbiamo una **probabilità del 7% di osservare un λ_{LR} maggiore di quello osservato effettivamente, se il modello più semplice ha generato i dati**. Abbiamo appena effettuato un ***Likelihood Ratio Test* (LRT)**. A questo punto, possiamo **decidere se la probabilità è abbastanza bassa da scegliere di affidarci all'ipotesi più complessa o restare con l'ipotesi semplice**. Quanto è “abbastanza”? Dipende da noi... la cosa bella è che, se tutto va bene, **sappiamo cosa aspettarci**. Se scegliamo una soglia del 5% di probabilità, ci aspettiamo che un test “positivo” su 20 sia un falso positivo (*type I error*), e così via...

Notiamo che la distribuzione χ^2 è tanto più spostata a destra quanto più *df* è grande...

```
xs = seq(0, 15, l = 200)
plot(0, 0, xlim = range(xs), ylim = c(0, 0.5), type = "n", ylab = "density",
     xlab = "x", main = "The Chisquare probabilty density distribution")
for(df in 1:6){
  lines(xs, dchisq(xs, df), lwd = 3, col = df+1)
}
abline(v = 5.2, lwd = 3, lty = 2, col = 1)
legend("topright", legend = 1:6, title = "degrees of freedom",
      lwd = 3, col = 2:7, bty = "n")
```

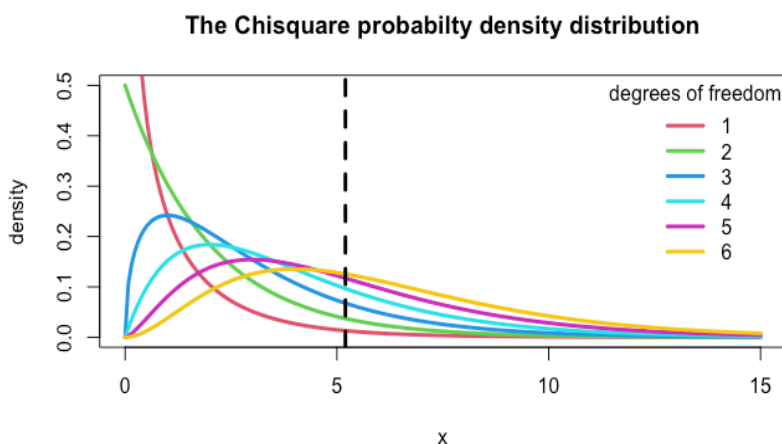


Figura 65 - Cosa succede alla distribuzione del Chi quadrato cambiando i gradi di libertà.

Questo fa sì che lo stesso valore di λ_{LR} corrisponda a probabilità maggiori se il numero di parametri di differenza tra i due modelli è più alto!

Predittori o non predittori?

Il *LRT* che abbiamo appena effettuato era basato su un valore di λ_{LR} **del tutto ipotetico**.

Ma cosa succede nel caso dei campioni s_1 e s_2 che entravano dalle nostre porte?

Ricreiamoli e riplottiamoli...

```
n = 20
mu = 164.7
sigma = 7.07
set.seed(2)
s_1 = rnorm(n = n, mean = mu, sd = sigma)
s_2 = rnorm(n = n, mean = mu, sd = sigma)
mu_1 = mean(s_1)
mu_2 = mean(s_2)
sigma_1 = sd(s_1)
sigma_2 = sd(s_2)
xs = seq(mu_2-4*sigma_2, mu_2+4*sigma_2, l = 100) # interval..
normal_pd_1 = dnorm(x = xs, mean = mu_1, sd = sigma_1)
normal_pd_2 = dnorm(x = xs, mean = mu_2, sd = sigma_2)
hist(s_1, xlim = range(c(s_1, s_2))+c(-10,10), col = SetAlpha(4, 0.25),
     freq = F, ylim = c(0, 0.06), xlab = "H", main = "Two samples")
abline(v = mu_1, col = 4, lwd = 3, lty = 2)
lines(xs, normal_pd_1, col = 4, lwd = 3)
hist(s_2, xlim = range(c(s_1, s_2))+c(-10,10), col = SetAlpha(2, 0.25),
     freq = F, ylim = c(0, 0.06), xlab = "H", add = T)
abline(v = mu_2, col = 2, lwd = 3, lty = 2)
lines(xs, normal_pd_2, col = 2, lwd = 3)
legend("topleft", title = "Door", bty = "n", lwd = 3, col = c(2,4),
```

```
legend = c("right", "left")
```

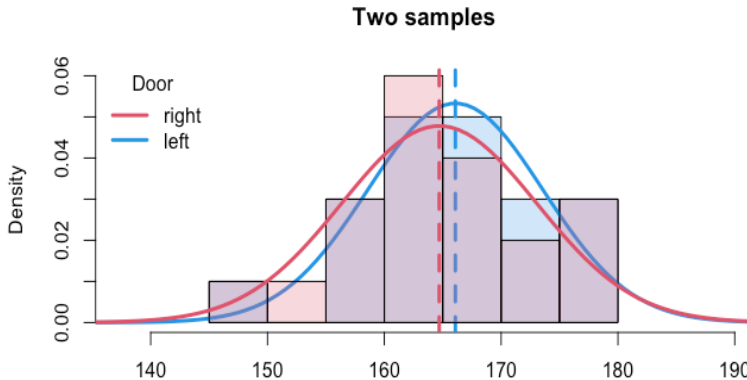


Figura 66 - Confronto tra due modelli 'fittati' a due campioni della stessa popolazione.

Sono molto simili. Sembra **ben ragionevole** che possano provenire da **una stessa popolazione**. Ma cosa dice il *Likelihood Ratio Test*?

```
fit_1 = lm(s_1 ~ 1)
fit_2 = lm(s_2 ~ 1)
fit_all = lm(c(s_1, s_2) ~ 1)

lambda = -2*(logLik(fit_all) - (logLik(fit_1) + logLik(fit_2)))
df = 2

pchisq(lambda, df, lower.tail = F)
## 'log Lik.' 0.7570118 (df=2)
xs = seq(0, 15, l = 200)
plot(xs, dchisq(xs, df), lwd = 3, col = 2, type = "l",
      ylab = "density", xlab = "x",
      main = "The Chisquare probability density distribution")
abline(v = lambda, lwd = 3, lty = 2, col = 2)
xsp = xs[which(xs >= lambda)]
polygon(x = c(xsp, rev(xsp)), y = c(dchisq(xsp, df), rep(0, length(xsp))),
        col = SetAlpha(2, 0.25), border = NA)
text(mean(xsp), max(dchisq(xs, df))*0.6,
      labels = paste("P = ", round(pchisq(lambda, df, lower.tail =
```

```
F), 3)), pos = 3)
arrows(x1 = mean(xsp)*0.7, x0 = mean(xsp),
       y0 = max(dchisq(xs, df))*0.6, y1 = 0.04)
```

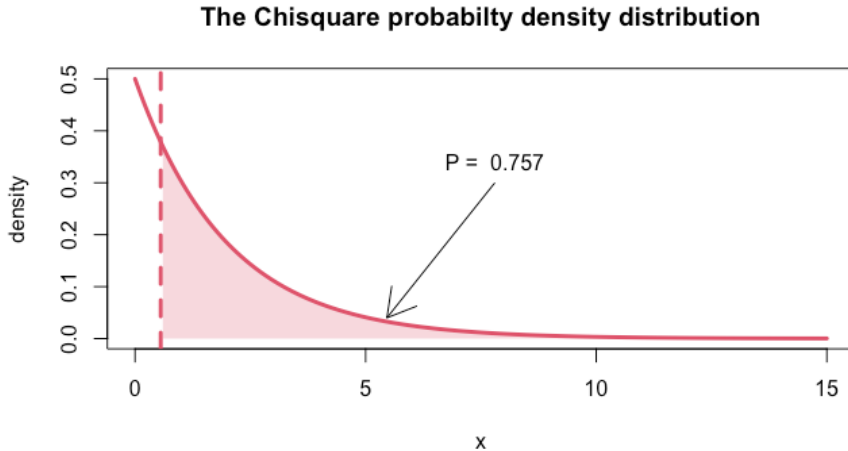


Figura 67 - Il Likelihood Ratio Test indica che è probabile che i campioni provengano dalla stessa popolazione.

In effetti, sembra piuttosto probabile che i due campioni provengano da una sola popolazione!

In altri termini, se dalle due porte arrivano membri **della stessa popolazione**, ai fini della mia scommessa, **è inutile conoscere da quale porta sia entrata una persona**! Non mi aiuta a **predire** la variabile che mi interessa... L'identità della porta **non è un predittore** della variabile **altezza**.

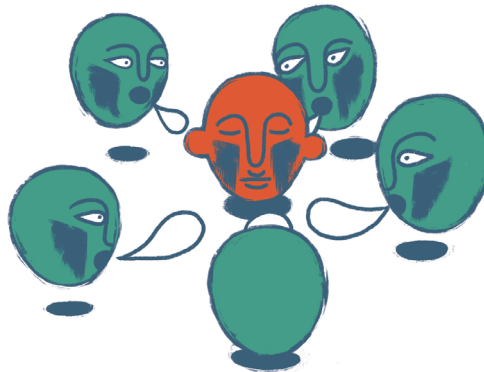


Figura 68 - I predittori circondano e influenzano la variabile dipendente.

Una regressione lineare

I modelli che abbiamo considerato hanno **due parametri** di differenza tra loro. Potrebbe avere senso provare un modello **intermedio**? Uno che abbia solo *un* parametro in più rispetto al modello “a una popolazione”?

Direi di sì. Un modello del genere potrebbe essere un modello in cui le due (ipotetiche) *popolazioni* statistiche hanno **medie diverse** ma **deviazioni standard uguali**.

Potremmo scriverlo come...

$$H = \mu_0 + DC_{D\mu} + \text{Norm}(\theta, \sigma)$$

in cui **D** è sempre la variabile “identità della porta” (0 = “destra”, 1 = “sinistra”), μ_0 è il valore atteso (la media) quando **D** = 0 (porta destra), e C_D è la differenza tra le *medie* della porta destra e della porta sinistra.

Questo è un **vero e proprio linear model** (in questo caso, una *regressione lineare*). Un *linear model* assume che i *residui* siano tutti estratti da una **singola distribuzione normale**, con media 0 e deviazione standard σ . In questo modello, **D** è un “**predittore**” (*predictor*) di **H**, che è la “**variabile di risposta**” (*response*)! Il valore atteso **quando tutti i predittori hanno valore 0** si chiama “**intercetta**” (*intercept*).

Più in generale, scriveremo la formula di un *linear model* con **n** predittori come...

$$R = C_0 + P_1C_1 + \dots + P_nC_n + \text{Norm}(\theta, \sigma)$$

In questa notazione, molto comune,

- **R** è la *risposta*;
- C_0 è l'*intercetta*;
- $P_1 \dots P_n$ sono i *predittori*;
- $C_1 \dots C_n$ sono i *coefficienti* relativi a ciascun predittore.

Adesso calcoliamo il *fit* di un modello del genere ai nostri dati. Per farlo, ci conviene riorganizzare un po' i dati.

Adesso abbiamo ciascun campione in un vettore separato...

```
s_1
```

```
## [1] 158.3588 166.0069 175.9261 156.7082 164.1326 165.6362
```

```
169.7052 163.0053
## [9] 178.7302 163.7188 167.6528 171.6410 161.9236 157.3495
177.3004 148.3607
## [17] 170.9117 164.9532 171.8607 167.7561
s_2
## [1] 179.4821 156.2165 175.9387 178.5194 164.7349 147.3664
168.0741 160.4823
## [9] 170.3009 166.7477 169.9243 166.9551 172.3085 162.6910
159.2089 160.4887
## [17] 152.4973 158.3187 160.7474 162.9572
```

... ma è molto più pratico avere *un solo vettore* con tutti i valori della variabile di risposta...

```
H = c(s_1, s_2)
```

... un solo vettore con tutti i valori corrispondenti del predittore...

```
D = c(rep("right", length(s_1)), rep("left", length(s_2)))
```

... e, possibilmente, unirli in un `data.frame`...

```
d = as.data.frame(cbind(H, D))
```

Diamo un'occhiata al `data.frame`...

```
d[sample(1:nrow(d), size = 6),]
##           H      D
## 4  156.708243983079 right
## 9  178.730230732136 right
## 6  165.636211410574 right
## 32 166.95505003576  left
## 15 177.300358749382 right
## 13 161.923643836588 right
```

Bene... ma non avevamo detto che **D** doveva essere 0 o 1?!

Sì... e potremmo trasformarla noi. Ma la funzione `lm` lo farà da sola se leggerà che `d$D` è un “fattore”!

```
d$D = as.factor(d$D)
```

Fittiamo il modello di cui abbiamo parlato (due popolazioni con la stessa deviazione standard)...

```
fit = lm(H ~ D, data = d)
```

Rifittiamo il modello “nullo” (una sola popolazione):

```
null = lm(H ~ 1, data = d)
```

Calcoliamo λ_{RT} :

```
lambda = -2*(logLik(null)-logLik(fit))
df = 1
```

Calcoliamo la probabilità che i dati provengano dal modello “nullo” rispetto al nuovo modello:

```
pchisq(lambda, df, lower.tail = F)
```

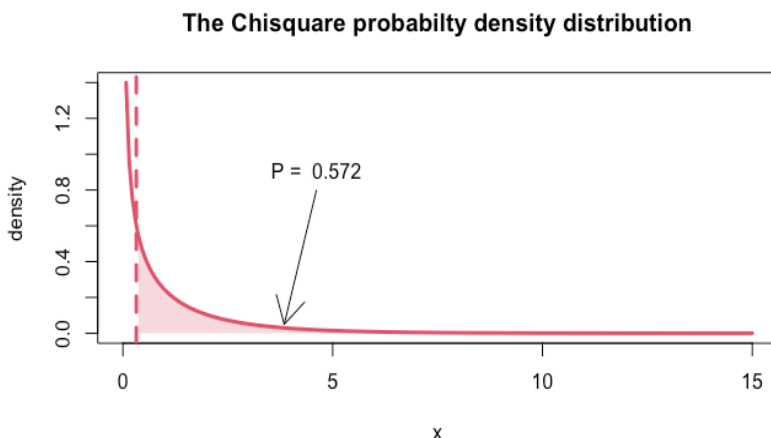


Figura 69 - La probabilità che i dati provengano dal modello “nullo” rispetto al nuovo modello.

Appena meno probabile, rispetto al modello con deviazioni standard indipendenti...

In effetti, c'è **una funzione che (se glielo chiediamo) fa il *Likelihood Ratio Test* per noi** per molti tipi di *linear models*, si chiama `drop1`.

```
drop1(fit, test = "Chisq")
## Single term deletions
##
## Model:
## H ~ D
##      Df Sum of Sq    RSS    AIC Pr(>Chi)
## <none>          2387.6 167.57
## D      1      19.152 2406.8 165.89  0.5719
```

Notate che `drop1` calcola $\Pr(>\chi)$ solo se scriviamo `test = "Chisq"` (può fare altri test o non farne nessuno).

Il risultato è identico. Se il nostro modello avesse incluso **più di un predittore**, `drop1` avrebbe testato **tutti i possibili modelli "ridotti" contro il modello "full"**, eliminando **un predittore alla volta**.

In questo modo, è possibile valutare la "significatività" di **tutti i predittori** di un modello lineare.

Un predittore che predice...

Adesso, per **toglierci la soddisfazione di vedere un test “significativo”**, proviamo a lavorare con due campioni che provengono davvero da due popolazioni diverse, per esempio nel caso in cui le porte dell’esempio fossero queste ...



Figura 70 - Nel nuovo esempio, i campioni che escono dalla due porte provengono da due popolazioni diverse (es. maschi e femmine).

Creiamo di nuovo due campioni, stavolta da popolazioni con media (e, in effetti, deviazioni standard) diverse. Questi valori di media e deviazione standard sono pubblicati *qui*.

```
mu_f = 164.7
sigma_f = 7.07

mu_m = 178.4
sigma_m = 7.59
```

Creiamo direttamente un `data.frame` con una colonna `D` che ci dice, come prima, se la persona esce dalla porta destra o sinistra. Adesso sappiamo che, dalle due porte, escono persone che, effettivamente, appartengono a due *popolazioni* diverse (notate che questo esempio evidenzia bene la differenza tra il significato della parola *popolazione* in ambito *statistico* e il suo significato in ambito *biologico*).

```
d = data.frame(D = sample(c("right", "left"), size = 40, replace = T))
```


Calcoliamo le differenze tra le medie dei maschi e delle femmine:

```
C_D_mu = mu_m - mu_f
C_D_sigma = sigma_m - sigma_f
```

“Codifichiamo” la variabile **D** come 0 (porta destra) o 1 (sinistra):

```
d$D[d$D == "right"] = 0
d$D[d$D == "left"] = 1
d$D = as.integer(d$D)
```

E creiamo il campione, direttamente in una nuova colonna **H** nel **data.frame** **d**, usando una formula come quella che abbiamo incontrato prima...

$$H = \mu_f + DC_{D\mu} + \text{Norm}(\theta, \sigma)$$

In linguaggio R...

```
set.seed(4)
d$H = mu_f + d$D*C_D_mu + rnorm(n = nrow(d), mean = 0, sd = sigma_f
+ d$D*C_D_sigma)
```

Ritrasformiamo la variabile **D** in “right” o “left” (non è necessario, anzi, come dicevamo **lm** la trasformerà internamente in 0/1, ma così mi sembra che si capisca meglio).

```
d$D[d$D == 0] = "right"
d$D[d$D == 1] = "left"
d$D = as.factor(d$D)
```

E, finalmente, “fittiamo” il nostro *linear model*...

```
fit = lm(H ~ D, data = d)
```

Lo plottiamo ...

```

xs = seq(mean(d$H)-6*sd(d$H), mean(d$H)+6*sd(d$H), l = 100) # interval..
normal_pd_left = dnorm(x = xs,
                        mean = fit$coefficients["(Intercept)"],
                        sd = summary(fit)$sigma)
normal_pd_right = dnorm(x = xs,
                        mean = fit$coefficients["(Intercept)"] +
                          fit$coefficients["Dright"],
                        sd = summary(fit)$sigma)
hist(d$H[d$D == "left"], xlim = range(d$H+c(-20,20)),
     col = SetAlpha(2, 0.25),
     freq = F, ylim = c(0, 0.08), xlab = "H", main = "Two doors")
abline(v = fit$coefficients["(Intercept)"], col = 2, lwd = 3, lty = 2)
lines(xs, normal_pd_left, col = 2, lwd = 3)
hist(d$H[d$D == "right"], xlim = range(d$H+c(-20,20)), col =
SetAlpha(4, 0.25),
     freq = F, add = T)
abline(v = fit$coefficients["(Intercept)"] + fit$coefficients["Dright"],
     col = 4, lwd = 3, lty = 2)
lines(xs, normal_pd_right, col = 4, lwd = 3)

legend("topleft", title = "Door", bty = "n", lwd = 3, col = c(2,4),
      legend = c("right", "left"))

```

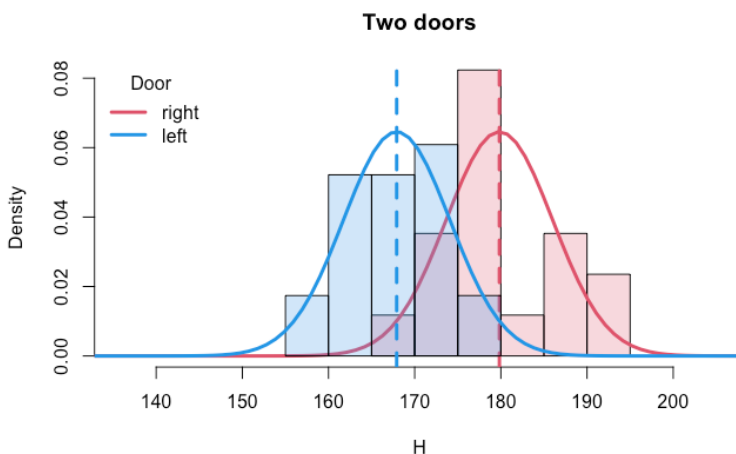


Figura 71 - Distribuzione dei campioni che escono dalle due porte e densità di probabilità calcolata dal modello 'fittato'

E, testiamolo...

```
drop1(fit, test = "Chisq")
## Single term deletions
##
## Model:
## H ~ D
##      Df Sum of Sq    RSS   AIC Pr(>Chi)
## <none>          1450.6 147.63
## D      1      1389 2839.6 172.50 2.18e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

...Ahhh!! :)

Adesso sappiamo (o crediamo di sapere...) che la porta **è un predittore** dell'altezza!

Se sappiamo che la persona uscirà dalla porta destra, invece che dalla porta sinistra, **la nostra scommessa sarà diversa!**

In altre parole, il **valore atteso** della nostra **risposta** (l'altezza) **dipende dal valore del predittore** ("destra" o "sinistra", oppure 0 o 1).

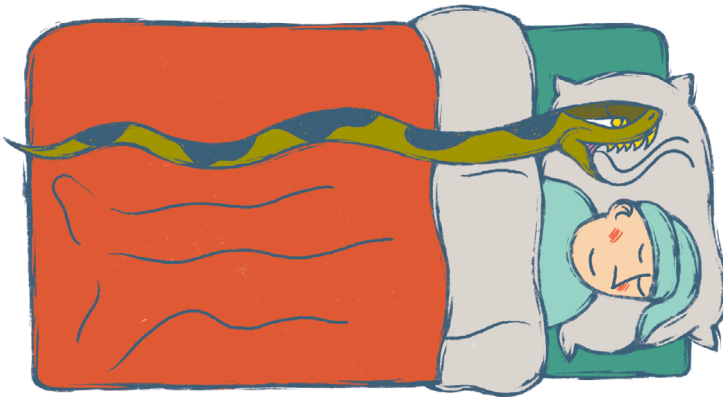


Figura 72 - Stimare.... ovvero adattare il modello ai dati, provando e riprovando per capire quando saremo pronti a mangiarci il fenomeno che stiamo studiando.

E le assunzioni?!

Già... le assunzioni!!

Il calcolo della *likelihood* (da cui dipende il *fit*, e quindi le predizioni, e i test) ha senso all'interno delle **assunzioni**... **senza le assunzioni, tutto quello che abbiamo fatto non ha senso!**

Se ci sono **buoni motivi** per pensare che le assunzioni che abbiamo fatto **non reggano**, tutti i calcoli non reggono...

Andiamo quindi a **recuperare le assunzioni che abbiamo fatto** lungo la strada...

Il campione che abbiamo osservato è **rappresentativo** della **popolazione** che ci interessa (ai fini della scommessa, è quella che comprende le persone che usciranno d'ora in poi...).

La distribuzione della variazione è *simmetrica*, anzi, abbiamo precisato poi, **la distribuzione dei residui è normale**.

- Assumere una distribuzione di probabilità è necessario per calcolare la *likelihood* di ciascuna osservazione.

Il campione è composto da osservazioni **indipendenti** tra loro. Anche in questo caso, quello che conta veramente, è che **i residui siano indipendenti**.

- L'indipendenza è necessaria per poter calcolare la *likelihood* del campione moltiplicando tra loro le *likelihood* di ciascuna osservazione.

Alla fine, per analizzare i nostri dati con una *regressione lineare* (*linear model* in senso stretto), abbiamo anche assunto che la distribuzione dei residui fosse *la stessa* per le due porte. Non solo dello stesso *tipo* (normale), ma proprio con la stessa *deviazione standard* (la media è sempre zero). In termini più generali...

La distribuzione dei residui non dipende dal valore atteso della risposta (omoschedasticità).

Adesso vediamo cosa possiamo dire sulle nostre assunzioni, relativamente all'esempio di cui ci siamo occupati:

Rappresentatività. Ovviamente, non sarà analizzando il *campione* che abbiamo che potremo capire se è *rappresentativo* della *popolazione* che ci interessa. Dobbiamo sperare che non ci siano dei motivi che ci sono sfuggiti per cui il campione che abbiamo preso non sia rappresenta-

tivo. Possiamo dire che ci dobbiamo fidare che i dati provengano da un buon “disegno dell’esperimento” (**experimental design**). Aggiungeremo qualcos’altro a proposito dell’assunzione n. 3.

Normalità. I residui li abbiamo. La funzione `residuals` li estrae dal risultato di `lm`, quindi possiamo controllare che sembri realistico che provengano da una distribuzione normale. Il modo più tipico è un *quantile-quantile plot* (in questo caso, lo realizziamo con la funzione `qqPlot` del pacchetto “cars”):

```
qqPlot(residuals(fit),
  distribution = "norm",
  # this is only cosmetics..
  pch = 21, col = 2, col.lines = 2, id = F,
  envelope = list(border = F), grid = F,
  main = "Q-Q plot"
)
```

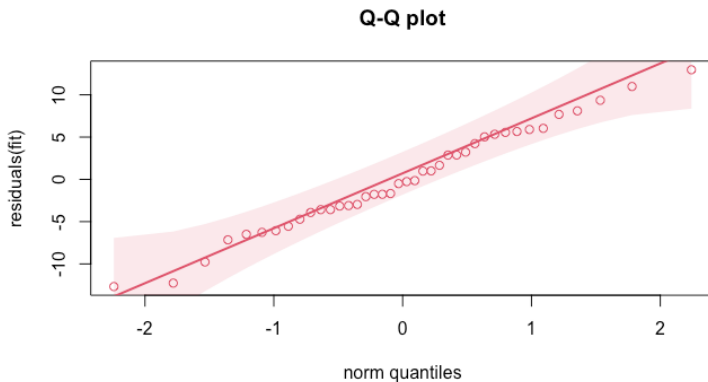


Figura 73 - Q-Q Plot dei quantili.

Una perfetta distribuzione normale corrisponderebbe all’allineamento di tutti i punti sulla retta di riferimento. L’intervallo di confidenza viene aggiunto per dare un’idea della “significatività” dello scostamento.

Anche un buon vecchio istogramma può essere utile...

```
hist(residuals(fit), col = SetAlpha(2, 0.25), freq = F)
xs = seq(min(residuals(fit)), max(residuals(fit)), 1 = 100)
lines(xs, dnorm(xs, 0, summary(fit)$sigma), lwd = 3, col = 2)
```

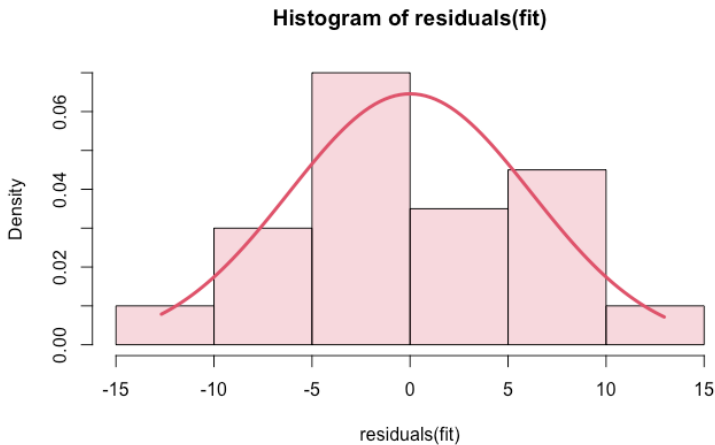


Figura 74 - Istogramma dei residui.

... ma non molto, in questo caso. Con campioni piccoli c'è assolutamente da aspettarsi che la "forma" dell'istogramma non somigli molto ad una curva normale.

Si può anche "testare" la normalità dei residui con il **Test di Shapiro-Wilk...**

```
shapiro.test(residuals(fit))
##
##  Shapiro-Wilk normality test
##
## data:  residuals(fit)
## W = 0.98675, p-value = 0.9131
```

Normalmente, si preferisce **semplicemente controllare visivamente**. Molti pensano che, per campioni piccoli, ci si potrebbe affidare al test, mentre, per campioni grandi, sarebbe più saggio usare l'ispezione visuale (perché il test diventa troppo sensibile anche a deviazioni leggere, che non influenzerebbero gravemente i risultati).

Indipendenza. I residui (cioè, ricordiamolo, *quanto ciascuna osservazione si discosta dal valore atteso!*) **non sono indipendenti tra loro** quando ci sono processi che *influenzano la risposta* in modo **sistematico** e che **non** sono descritti dal modello!

Prendiamo il caso delle porte dei bagni... Noi sappiamo che l'identità della porta *predice* (è in *correlazione* con) l'altezza delle persone che escono (e entrano). Immaginiamo, però, un modello che cerchi di predire l'altezza media senza considerare l'identità della porta... In effetti, lo abbiamo già incontrato:

```
null = lm(H ~ 1, data = d)
```

Per evidenziare meglio il problema, immaginiamo di stimare il modello “nullo” ed il modello “giusto” ad un campione molto più grande, diciamo 1000 osservazioni, e con una differenza tra le due porte molto più marcata.

Il modo più semplice di aumentare la differenza è diminuire la deviazione standard (`sigma_f`, prima era 7.07), e per semplicità, diciamo che adesso le deviazioni standard sono uguali (`C_D_sigma`)

```
sigma_f = 2
C_D_sigma = 0
set.seed(0)
d = data.frame(D = sample(c("right", "left"), size = 1000, replace = T))

d$D[d$D == "right"] = 0
d$D[d$D == "left"] = 1
d$D = as.integer(d$D)

d$H = mu_f + d$D*C_D_mu + rnorm(n = nrow(d), mean = 0, sd = sigma_f +
+ d$D*C_D_sigma)

d$D[d$D == 0] = "right"
d$D[d$D == 1] = "left"
d$D = as.factor(d$D)

fit = lm(H ~ D, data = d)
null = lm(H ~ 1, data = d)
```

Plottiamo entrambi i modelli...

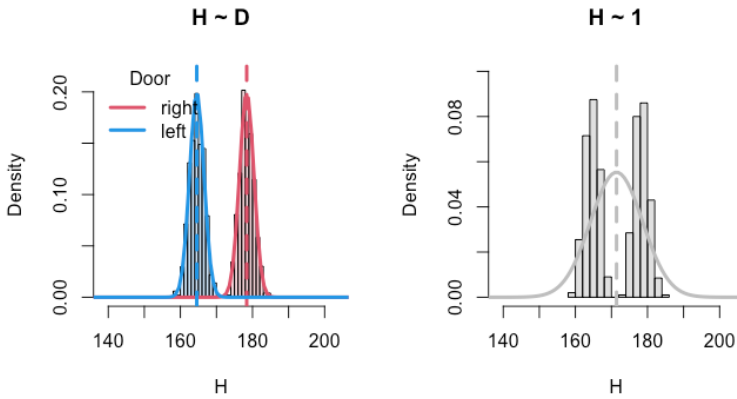


Figura 75 - Plot dei due modelli.

Ed osserviamone i residui...

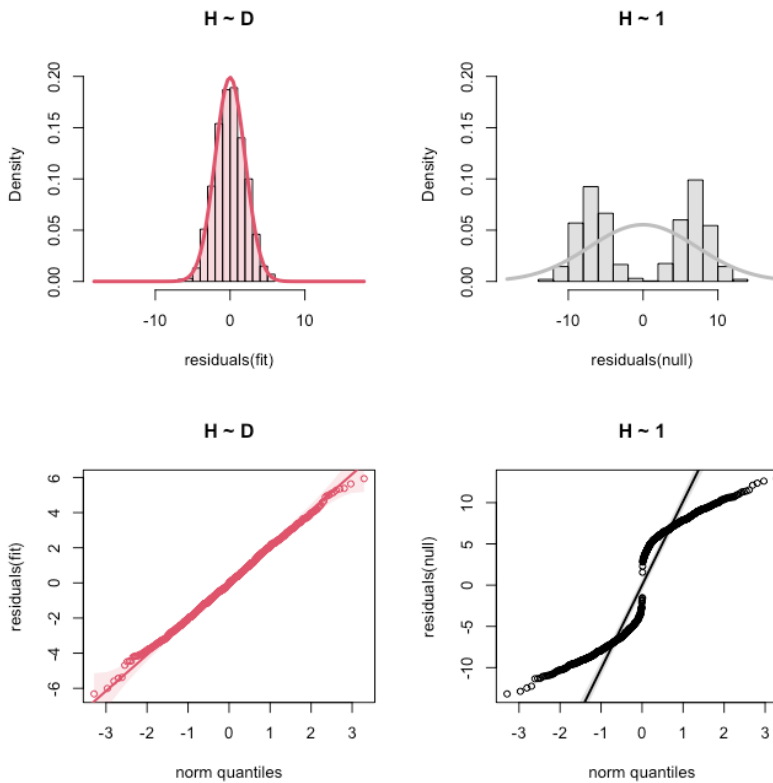


Figura 76 - Plot dei residui dei due modelli.

Non c'è bisogno di un test per vedere che c'è qualche problema...

Come conseguenza del fatto che le osservazioni sono **raggruppate tra loro** (in questo caso, in due gruppi *molto* ben definiti!) i residui del modello con *una sola popolazione*:

- **non** hanno più una **distribuzione normale**!
- sono **molto più ampi** (il modello “spiega” meno bene i dati)!

Questo...

1. Fa sì che i nostri calcoli della *likelihood* non siano corretti (abbiamo moltiplicato assumendo eventi indipendenti, invece non lo erano!).

1. “Sballa” le nostre misure dell’incertezza associata alle stime!

Vediamo nell’esempio. La funzione `predict` applicata a un risultato di `lm` (un modello “fittato”) è in grado di restituire, per ogni osservazione, il **valore atteso** (*fitted*), insieme ad un **intervallo di confidenza**. Questo *intervallo di confidenza*, per *default* è al 95%. Questo vuol dire che, in teoria, il 95% delle osservazioni dovrebbe ricadere al suo interno.

Adesso vediamo quante osservazioni ricadono effettivamente nell’intervallo di confidenza al 95%...

```
test_CI = numeric(2)
names(test_CI) = c("H ~ D", "H ~ 1")

p = predict(fit, interval = "prediction")
tt = table(factor(d$H > p[, "lwr"] & d$H < p[, "upr"], levels =
c("TRUE", "FALSE")))
test_CI[1] = tt[1]/sum(tt)

p = predict(null, interval = "prediction")
tt = table(factor(d$H > p[, "lwr"] & d$H < p[, "upr"], levels = c("TRUE",
"FALSE")))
test_CI[2] = tt[1]/sum(tt)

test_CI*100
## H ~ D H ~ 1
## 95.6 100.0
```

Nel modello con due popolazioni ($H \sim D$), quasi esattamente il 95% delle osservazioni ricade nell’intervallo di confidenza.

Nel modello con una popolazione ($H \sim 1$), *tutte* le osservazioni ricadono nell'intervallo. Questo secondo modello **sovrastima** l'ampiezza dell'intervallo di confidenza.

Vediamo adesso, però, cosa succede se il nostro campione non è “bilanciato” (ovvero, se abbiamo osservato molte più persone uscire dalla porta destra che dalla porta sinistra...).

Usiamo l'argomento `prob` della funzione `sample` per creare un campione “sbilanciato”:

```
set.seed(1)
d = data.frame(D = sample(c("right", "left"), size = 1000, replace = T,
                          prob = c(0.9, 0.1)))

d$D[d$D == "right"] = 0
d$D[d$D == "left"] = 1
d$D = as.integer(d$D)
d$H = mu_f + d$D*C_D_mu + rnorm(n = nrow(d), mean = 0,
                                sd = sigma_f + d$D*C_D_sigma)

d$D[d$D == 0] = "right"
d$D[d$D == 1] = "left"
d$D = as.factor(d$D)
fit = lm(H ~ D, data = d)
null = lm(H ~ 1, data = d)
```

Plottiamo entrambi i modelli...

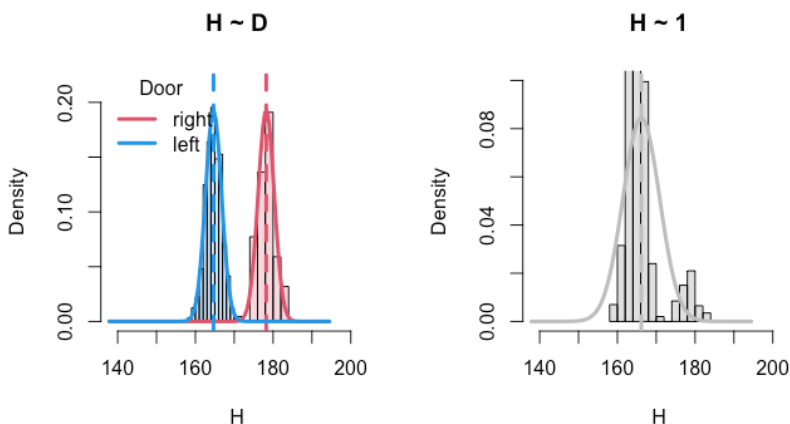


Figura 77 - Plot dei due modelli.

Osserviamone i residui...

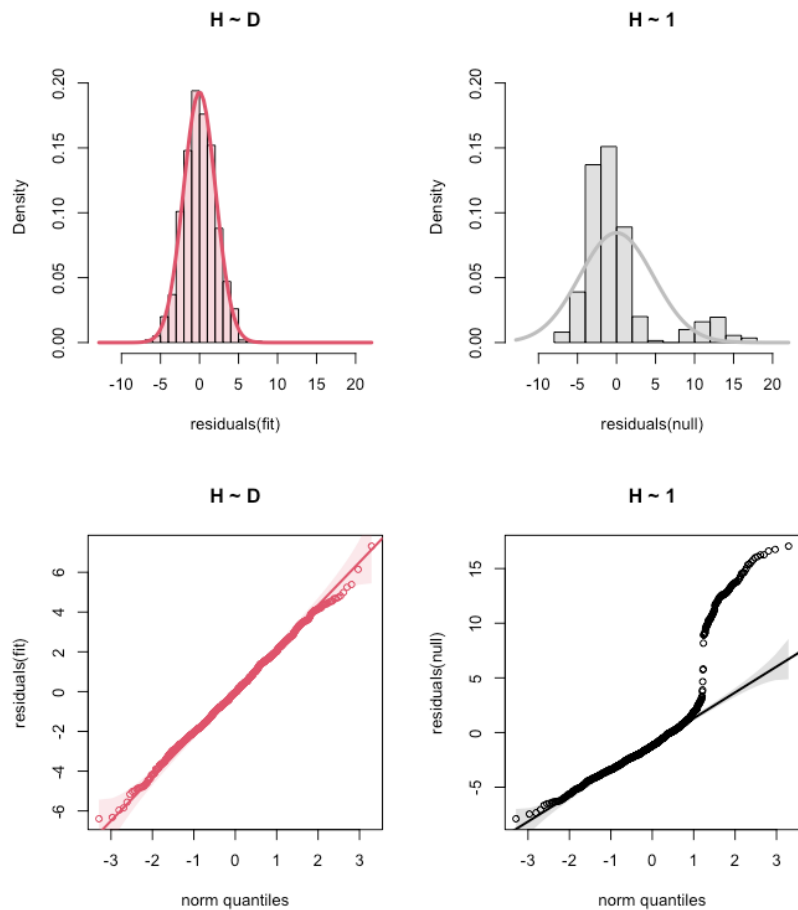


Figura 78 - Plot dei residui dei due modelli.

Vediamo quante osservazioni ricadono effettivamente nell'intervallo di confidenza al 95%...

```
## H ~ D H ~ 1
## 94.5 90.0
```

Questa volta è successo esattamente **il contrario!!** Il modello $H \sim 1$ ha **sottostimato** l'ampiezza dell'intervallo di confidenza!

In effetti, le conseguenze esatte di una violazione dell'assunzione di **indipendenza dei residui** non sono sempre facili da prevedere. La violazione

può portare sia a **sovracconfidenza** (*overconfidence*) che a una **sottoconfidenza** (*underconfidence*). La tendenza verso l'una o l'altra dipende dalla struttura del campione che abbiamo a disposizione in relazione al processo che lo ha generato. Nel nostro esempio, nel primo caso abbiamo un campione "bilanciato", e tendiamo a esagerare l'ampiezza dell'intervallo. Nel secondo caso, con un campione "sbilanciato" (seppure *rappresentativo*, perché potremmo benissimo avere che – effettivamente – dalla porta destra escano più persone che dalla porta sinistra), abbiamo sottostimato l'ampiezza dell'intervallo. Con fenomeni più complessi, le conseguenze possono essere ancora più serie e indurre a trarre conclusioni anche **molto sbagliate**.

Omoschedasticità. Abbiamo formulato il nostro *linear model* come...

$$R = C_0 + P_1 C_1 + \dots + P_n C_n + \text{Norm}(\theta, \sigma)$$

Come abbiamo già sottolineato, in questa equazione esiste un solo termine di errore, rappresentato da *una* distribuzione normale con *una* deviazione standard (σ).

Noi abbiamo creato i nostri campioni da popolazioni con medie **e** deviazioni standard diverse:

```
mu_f = 164.7
sigma_f = 7.07

mu_m = 178.4
sigma_m = 7.59
```

Il modello che stiamo considerando, quindi, **non descrive il processo in modo del tutto corretto**.

Come abbiamo appena visto, una violazione delle assunzioni del modello tende a invalidare i nostri risultati. Un aspetto cruciale dei risultati che chiediamo a un *linear model* è che sia in grado di fornire una stima **onesta** dell'**incertezza** associata alle stime.

Dall'onestà di questa valutazione dipende sia la nostra fiducia nelle **previsioni** che la nostra **fiducia nel P-value** (un *P-value* "affidabile" corrisponde con la **frequenza attesa di errori di tipo I**).

Possiamo usare, di nuovo, una **simulazione** per mettere alla prova l'“onestà” della stima dell'incertezza.

Generiamo un *dataset molto grande* ($n = 10000$), in modo che il risultato sia più *preciso*...

```
set.seed(0)
n = 10000
x = rep(NA, n_rep)

d = data.frame(
  # campioni estratti da due popolazioni
  H = c(rnorm(n = n, mean = mu_f, sd = sigma_f),
        rnorm(n = n, mean = mu_m, sd = sigma_m)),
  # crea un “fattore” door
  door = as.factor(rep(c(“left”, “right”), each = n)))
```

Fittiamo un modello lineare...

```
fit = lm(H ~ door, data = d)
```

Calcoliamo l'intervallo di confidenza delle predizioni...

```
p = predict(fit, interval = “prediction”)
```

E contiamo quante osservazioni ricadono nell'intervallo di confidenza...

```
tt = table(factor(d$H > p[,“lwr”] & d$H < p[,“upr”], levels =
c(“TRUE”, “FALSE”)))
tt[1]/sum(tt)
##      TRUE
## 0.95085
```

Praticamente perfetto!

Cosa vuol dire? Che, in generale, **la violazione dell'omoschedasticità non è un problema**, o che, **in questo caso**, la violazione non è tale da creare un problema importante?

In effetti, qui avevamo solo una **violazione molto piccola**...

$$\mu_f = 164.7$$

$$\sigma_f = 7.07$$

$$\mu_m = 178.4$$

$$\sigma_m = 7.59$$

Proviamo a vedere cosa succederebbe con una violazione **ben più grande**...

Queste sono le **nuove popolazioni** (stesse medie, ma deviazioni standard molto diverse)...

```
mu_f = 164.7
sigma_f = 3

mu_m = 178.4
sigma_m = 12
```

Generiamo e plottiamo i dati ...

```
n = 10000

x = rep(NA, n_rep)

d = data.frame(
  # campioni estratti da due popolazioni
  H = c(rnorm(n = n, mean = mu_f, sd = sigma_f),
        rnorm(n = n, mean = mu_m, sd = sigma_m)),
  # crea un "fattore" door
  door = as.factor(rep(c("left", "right"), each = n))
)

# plot dei dati
par(mar = c(5.1, 4.1, 2.1, 2.1))
hist(d$H[d$door == "right"], col = SetAlpha(2, 0.5),
     xlim = range(d$H), ylim = c(0, n/3), main = "", xlab = "H")
```

```
hist(d$H[d$door == "left"], col = SetAlpha(4, 0.5), add = T)
legend("topleft", title = "Door", bty = "n", pch = 22,
      pt.bg = c(SetAlpha(c(2,4))), legend = c("right", "left"))
```

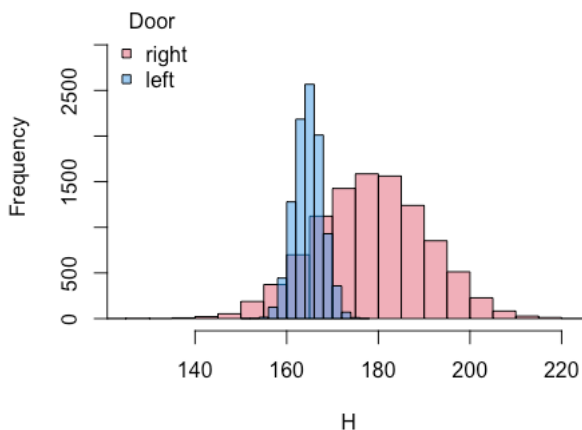


Figura 79 - Distribuzioni dei campioni estratti dalle nuove popolazioni.

Fittiamo il modello...

```
fit = lm(H ~ door, data = d)
```

Calcoliamo l'intervallo di confidenza delle predizioni...

```
p = predict(fit, interval = "prediction")
```

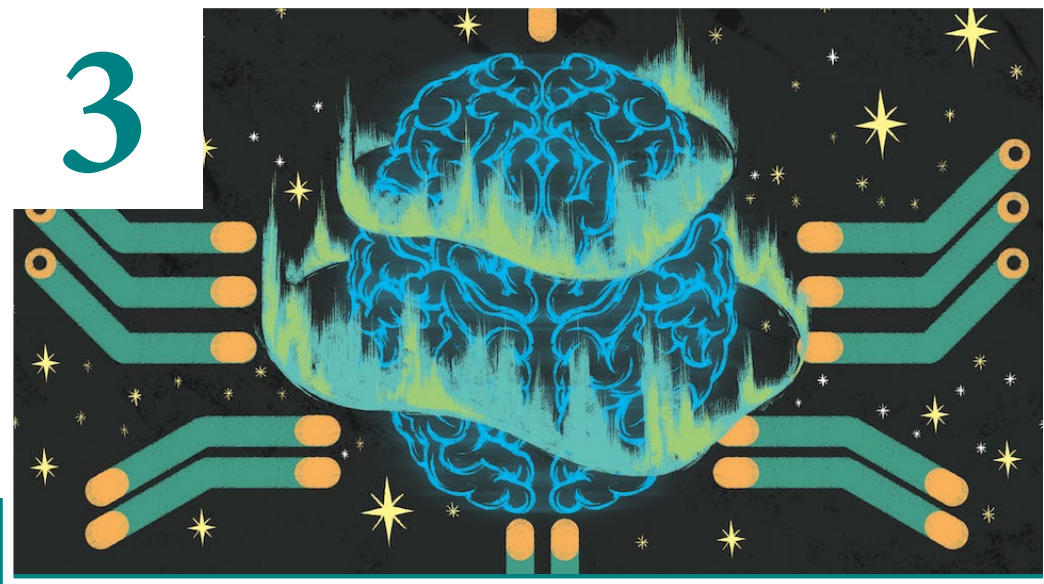
E contiamo quante osservazioni ricadono nell'intervallo di confidenza...

```
tt = table(factor(d$H > p[, "lwr"] & d$H < p[, "upr"], levels =
c("TRUE", "FALSE")))
tt[1]/sum(tt)
## TRUE
## 0.9239
```

Non è una differenza enorme, ma **c'è**.

Applicando il nostro modello a questi dati, avremmo una **sottostima dell'incertezza associata alle nostre predizioni!**

3



Machine learning in R

Introduzione

Il **machine learning** è una branca dell'informatica che studia la progettazione di algoritmi in grado di imparare e apprendere nuove informazioni. Lo scopo è quello di "insegnare" alla macchina così da renderla capace di generare predizioni sulla base dell'informazione ricevuta. In poche parole, è come se istruissimo il cervello elettronico a sviluppare un proprio modo di ragionare; tanto più saranno i dati disponibili, più questo sarà in grado di apprendere informazioni e restituire risultati accurati. Per quanto ciò possa assomigliare ad un romanzo di Gibson (non sapete chi sia? Molto male!) o al film *Blade Runner*, la risposta è: no, il machine learning in R non vi permetterà di creare un'intelligenza artificiale cosciente con cui parlare e che risolva magicamente i vostri problemi. Infatti, il processo di apprendimento e le modalità con cui l'informazione viene trasmessa alla macchina, sono le parti più critiche dell'utilizzo di questi algoritmi. Tutto dipenderà ovviamente dalla quan-

tità (e la qualità) dei dati che immettiamo nel cervello e, ovviamente, il tipo di risposta che desideriamo questo ci restituisca.

Non trattandosi quindi di una scatola magica, spetta a noi supervisionare e indirizzare l'addestramento della macchina in modo da ottenere dei risultati il più possibile accurati. In questo capitolo, ci occuperemo di descrivere alcune nozioni base dei modelli di machine learning, la gestione dei dati e gli algoritmi più utilizzati. Quindi rimbocchiamoci le maniche e cominciamo a capire come insegnare ad una macchina a *pensare*! Magari, alla fine di questo capitolo, anche voi potrete affermare *Ho visto cose che voi umani...*

Innanzitutto, è opportuno spiegare nel dettaglio le due grosse famiglie in cui vengono comunemente divisi gli algoritmi di machine learning, ossia il **supervised learning** e l'**unsupervised learning**.

Supervised Learning

Un **supervised learner** consiste in un modello predittivo utilizzato per la previsione di un *output* utilizzando altre variabili (o *input*) nel *set* di dati. In un modello di questo genere, infatti, vengono messe in relazione la variabile di *output* (quella che viene predetta) e i parametri di *input* (variabili predittive). Il termine *supervised* si riferisce al fatto che l'utente ha il compito di dare istruzioni su cosa la macchina deve imparare. In particolare, dato un insieme di dati, l'algoritmo di apprendimento tenta di ottimizzare una data funzione (a seconda del modello) per trovare la combinazione di valori che si traducono in un valore predetto che è il più vicino possibile al *target* effettivo.

Esempi di modellizzazione predittiva possono essere:

- predire le scelte di acquisto di una persona sulla base della sua età, della zona geografica in cui vive, del suo reddito annuale ecc.;
- predire la probabilità di presenza di una specie animale ad una determinata latitudine, sulla base delle temperature medie annuali, caratteristiche dell'habitat, inquinamento ecc.;
- predire la probabilità di contagio di un virus sulla base del numero di contatti tra persone, umidità, ricircolo dell'aria, tasso di igienizzazione ecc.

Ovviamente, appare subito chiaro che migliore è la conoscenza dell'argomento da parte dell'utente, più facile sarà capire quali informazioni possano essere realmente utili per la predizione del nostro *output*. Di sicuro, utilizzare il numero medio di mele che vengono consumate dalle persone nel mondo non sarà un dato molto utile per predire la probabilità di precipitazione in Costa Rica. Per quanto si possa sempre provare!

La maggior parte degli algoritmi di apprendimento supervisionati possono essere raggruppati in due categorie, **regressione** o **classificazione**.

Quando l'obiettivo del nostro apprendimento supervisionato è predire un risultato numerico, ci riferiamo a questo come a un problema di **regressione** (da non confondere con la modellizzazione di regressione lineare). I problemi di regressione ruotano intorno alla previsione che cade su un *continuum* numerico.

Quando l'obiettivo del nostro apprendimento supervisionato è quello di prevedere un risultato categorico, ci riferiamo a questo come un problema di **classificazione**. Problemi di classificazione ruotano invece intorno alla predizione di una risposta binaria o multinomiale.

La figura seguente ci mostra un tipico problema di classificazione:

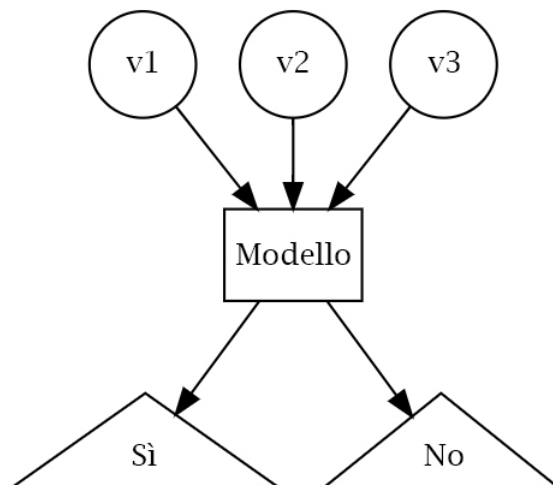


Figura 80 - Esempio di un problema di classificazione.

Unsupervised learning

L'**unsupervised learning** include una serie di strumenti statistici per comprendere e descrivere meglio i dati, ma esegue l'analisi senza una variabile *target*. In sostanza, l'apprendimento senza supervisione riguarda l'identificazione dei gruppi in un *set* di dati. I gruppi possono essere definiti dalle righe (cioè *clustering*) o dalle colonne (riduzione delle dimensioni).

L'apprendimento non supervisionato viene spesso eseguito come parte di un'analisi esplorativa dei dati. Tuttavia, l'esercizio tende ad essere più soggettivo, e non c'è un obiettivo semplice per l'analisi, come la previsione di una risposta. Inoltre, può essere difficile valutare la qualità dei risultati ottenuti con questi metodi di apprendimento. La ragione è piuttosto semplice, se ci pensate: se ci adattiamo ad un modello predittivo usando una tecnica di apprendimento supervisionata, allora è possibile controllare il nostro lavoro vedendo quanto bene il nostro modello predice la risposta Y su osservazioni non utilizzate per adattare il modello. Tuttavia, nell'apprendimento senza supervisione, non c'è modo di controllare il nostro lavoro perché non conosciamo la vera risposta: il problema non è supervisionato!

Nonostante la sua soggettività, l'importanza dell'apprendimento non supervisionato non deve essere trascurata, e tali tecniche possono essere utilizzate per esempio in casi come:

- dividere i consumatori in diversi gruppi omogenei in modo che strategie di marketing personalizzate possano essere sviluppate e implementate per ogni segmento;
- identificare specie che hanno un *pattern* di distribuzione molto simile sulla base di gradienti ambientali;
- identificare gruppi di acquirenti online con storie di navigazione e di acquisto simili, così come gli elementi che sono di particolare interesse per gli acquirenti all'interno di ogni gruppo.

Dati

I *set* di dati scelti per questo libro ci permetteranno di illustrare le diverse caratteristiche degli algoritmi di machine learning presentati. Di conseguenza, i *set* di dati esemplari che vi consigliamo di utilizzare sono quelli più comunemente utilizzati nel mondo di *R* (es. *mtcars*, *iris*, *geyser*).

L'***iris dataset*** può essere facilmente scaricato ed esplorato utilizzando il pacchetto `datasets`:

```
library(datasets)
data(iris)
summary(iris)
```

##	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
##	Min. :4.300	Min. :2.000	Min. :1.000	Min. :0.100
##	1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600	1st Qu.:0.300
##	Median :5.800	Median :3.000	Median :4.350	Median :1.300
##	Mean :5.843	Mean :3.057	Mean :3.758	Mean :1.199
##	3rd Qu.:6.400	3rd Qu.:3.300	3rd Qu.:5.100	3rd Qu.:1.800
##	Max. :7.900	Max. :4.400	Max. :6.900	Max. :2.500
##	Species			
##	setosa	:50		
##	versicolor	:50		
##	virginica	:50		
##				
##				
##				

Ugualmente per quanto riguarda l'***mtcars dataset***:

```
data(mtcars)
head(mtcars, 6)
```

##	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
##	Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4
##	Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4
##	Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4

## Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
## Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
## Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

Divisione dei dati

Come abbiamo già detto, un obiettivo principale del processo di apprendimento automatico è quello di trovare un algoritmo che predica con maggior precisione i valori di *output* sulla base di variabili di *input*. In poche parole, vogliamo un algoritmo che non solo si adatti bene ai nostri dati passati, ma soprattutto, che riesca a predire accuratamente un risultato futuro. Questa si chiama **generalizzazione** del nostro algoritmo. La modalità con la quale “spendiamo” i nostri dati ci aiuterà a capire come il nostro algoritmo generalizza i dati futuri.

Si suppone che l'utente abbia già imparato le principali funzioni per esplorare i *dataset* e osservare più da vicino le variabili che li descrivono. In questo paragrafo approfondiremo una parte molto importante della modellizzazione tramite machine learning: la ripartizione dei dati. Questo *step* è di fondamentale importanza dato che da questo dipendono fattori come appunto la capacità generalizzante del modello.

Per fornire una comprensione accurata della generalizzazione del nostro modello ottimale finale, possiamo dividere i nostri dati in *set* di dati di **training** e **test**:

- **training set**: questi dati vengono utilizzati per addestrare i nostri algoritmi, ottimizzare i parametri di apprendimento, confrontare i modelli e tutte le altre attività necessarie per scegliere un modello finale;
- **test set**: dopo aver scelto un modello finale, questi dati vengono utilizzati per stimare le prestazioni del modello, a cui ci riferiamo come **errore di generalizzazione**.

Data una quantità fissa di dati, le raccomandazioni tipiche per dividere i dati in divisioni *training-test* includono il 60%(*training*)-40%(*test*), il 70%-30%, o l'80%-20%.

In generale, queste sono linee guida appropriate da seguire; tuttavia, è bene tenere a mente i seguenti punti: (i) spendere troppa informazione

(ad esempio, $>80\%$) non ci permetterà di ottenere una buona valutazione delle prestazioni predittive. Possiamo trovare un modello che si adatti molto bene ai dati del *training*, ma è poco generalizzabile; (ii) allocare troppi dati nel *test set* ($>40\%$) non ci permetterà di ottenere una buona valutazione dei parametri del modello durante l'addestramento.

I due modi più comuni di dividere i dati sono il **campionamento casuale semplice** e il **campionamento stratificato**.

Campionamento casuale semplice

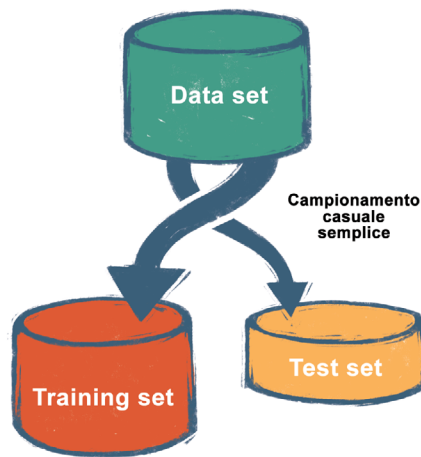


Figura 81 - Campionamento casuale semplice.

Il modo più semplice per dividere i dati in *training* e *test set* è prendere un **campione casuale semplice**. Ciò, ovviamente, non tiene conto di un campionamento secondo la distribuzione della nostra variabile di risposta *Y*. Dato che il campionamento è un processo casuale, occorre quindi impostare il generatore di numeri casuali con un funzione `set.seed` per ottenere dei risultati riproducibili. In questo libro, useremo spesso il `set.seed(123)` per la riproducibilità, ma il numero stesso non ha un significato speciale.

```
set.seed(123) # per la riproducibilita'
index_n <- sample(1:nrow(iris), round(nrow(iris) * 0.7))
training_set <- iris[index_n, ]
test_set <- iris[-index_n, ]
```

Con una dimensione del campione sufficiente, questo metodo di campionamento si tradurrà in genere in una distribuzione normale della nostra variabile Y (nel caso sia numerica) tra il *training* ed il *test set*, oppure in una simile distribuzione delle categorie nel caso in cui il nostro sia un problema di classificazione, come illustrato di seguito:

```
table(training_set$Species)
##
##      setosa versicolor  virginica
##         36         32         37
table(test_set$Species)
##
##      setosa versicolor  virginica
##         14         18         13
```

Questo è già un primo passo per renderci conto se il nostro modello ha abbastanza diversità di categorie per poter apprendere le relazioni tra le variabili.

Campionamento stratificato

Se il nostro insieme di dati non è abbastanza grande da permettere un approccio casuale, o se vogliamo controllare esplicitamente il campionamento in modo che il nostro *training* e *test set* abbiano simili distribuzioni della variabile Y, allora è necessario utilizzare un **campionamento stratificato**. Questo approccio è comunemente più utilizzato con problemi di classificazione in cui la variabile di risposta può essere gravemente squilibrata (ad esempio, 90% delle osservazioni con risposta “Sì” e 10% con risposta “No”). Tuttavia, possiamo anche applicare il campionamento stratificato a problemi di regressione per *set* di dati che hanno una piccola dimensione del campione, ed in cui la variabile di risposta si discosta fortemente dalla normalità.

Il modo più semplice per eseguire il campionamento stratificato su una variabile di risposta è usare il pacchetto `rsample`, dove si specifica

la variabile di risposta. Applicando il campionamento stratificato sul nostro iris *dataset*, ovviamente scopriremo che le variabili categoriche sono ugualmente presenti nel nostro insieme:

```
library(rsample) # per la riproducibilità
prop.table(table(iris$Species))
##
##      setosa versicolor  virginica
## 0.3333333 0.3333333 0.3333333
split_n <- initial_split(iris, prop = 0.7, strata = "Species")
training_set_strata <- training(split_n)
test_set_strata <- testing(split_n)
prop.table(table(training_set_strata$Species))
##
##      setosa versicolor  virginica
## 0.3333333 0.3333333 0.3333333
prop.table(table(test_set_strata$Species))
##
##      setosa versicolor  virginica
## 0.3333333 0.3333333 0.3333333
```

In questo caso quindi, il campionamento stratificato ci aiuta ad ottenere un uguale proporzione delle classi in ogni insieme di dati anche se non abbiamo ottenuto un risultato molto diverso da quello casuale. Ma supponiamo di avere un *dataset* con una distribuzione della variabile categorica sbilanciata, come ad esempio una risposta **Si** o **No**:

```
library(wakefield)
yn_data <- answer(200, x = c("No", "Si"),
                  prob = c(0.8, 0.2), name = "Risposte")
prop.table(table(yn_data))
## yn_data
##   No   Si
## 0.81 0.19
```


In una situazione come questa, acquista molta più importanza ripartire i dati in modo che il numero di risposte **Si** e **No** siano ugualmente distribuiti tra il *training* e il *test set*.

Metodi di ricampionamento

Forse starete scalpitando per poter sapere come mettere le mani sul vostro codice di machine learning e cominciare ad addestrare la vostra macchina. Purtroppo (o per fortuna) i passi da fare prima di arrivare a coronare il vostro sogno sono molti. Saper utilizzare algoritmi di machine learning e ottenere buoni risultati richiede un'attenta preparazione degli ingredienti e una conoscenza di tutti gli strumenti, esattamente come per preparare un piatto complesso. Come abbiamo già detto, una volta diviso il nostro insieme di dati, avremo due differenti partizioni, il *training* e il *test set*. Tuttavia, durante il processo di addestramento del modello non utilizzeremo il *test set* per valutare le prestazioni del modello durante il suo apprendimento. Quindi, come valutiamo le prestazioni di generalizzazione del modello durante la fase di *training*? Un'opzione è quella di valutare un errore metrico basato sui dati di *training*. Sfortunatamente, questo porta a risultati parziali in quanto alcuni modelli possono eseguire molto bene sui dati di formazione, ma non generalizzare bene ad un nuovo *set* di dati. Un secondo metodo è quello di utilizzare quello che viene chiamato un **validation approach**, che comporta un'ulteriore suddivisione dell'insieme del *training set* per creare due parti: un *training* e un **validation set**. Possiamo quindi addestrare il nostro modello sul nuovo *set* di *training* e contemporaneamente stimare le prestazioni sul *validation set*, che verrà utilizzato dall'algoritmo per stimare l'errore di predizione durante l'addestramento. In alcuni casi, per esempio quando si lavora con un *set* di dati non molto grande, questa validazione può essere altamente variabile e inaffidabile. Possiamo dire che, come la dimensione del nostro *set* di dati aumenta, più questa preoccupazione si riduce. I **metodi di ricampionamento** forniscono un approccio alternativo permettendoci di adattare ripetutamente un modello di interesse a parti dei dati di *training* e di testarne le prestazioni su altre parti. I

due metodi di ricampionamento più comunemente usati includono la **cross-validation** ed il **bootstrapping**.

Cross-validation

La ***k-fold cross-validation*** (o più comunemente ***k-fold CV***) è un metodo di ricampionamento che divide casualmente i dati di *training* in k gruppi di dimensioni approssimativamente uguali. Il modello viene addestrato utilizzando i $k-1$ gruppi mentre un gruppo viene utilizzato per calcolare le prestazioni del modello. Questa procedura viene ripetuta k volte; ogni volta, un gruppo diverso viene usato come convalida. Questo processo si traduce in k stime dell'errore di generalizzazione. In questo modo, la stima del *k-fold CV* è calcolata usando la media degli errori di prova di k , fornendoci un'approssimazione dell'errore che potremmo aspettarci su dati di test. Di conseguenza, con *k-fold CV*, ogni osservazione nei dati di addestramento sarà effettuata una sola volta per essere inclusa nella serie di *test*. In pratica, si usa tipicamente $k=5$ o $k=10$. Non esiste una regola formale per quanto riguarda le dimensioni di k ; tuttavia, man mano che k diventa più grande, la differenza tra le prestazioni stimate e le prestazioni reali da vedere sul *set* di prova diminuirà. D'altra parte, usare k troppo grande può introdurre oneri computazionali.

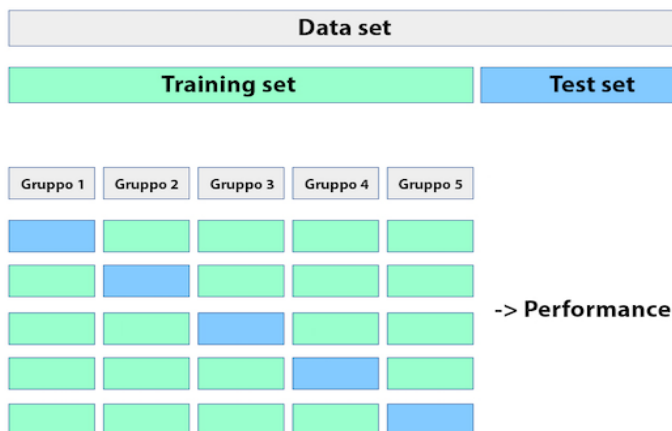


Figura 82 - Esempio schematico di *k-fold cross validation*.

Alcuni pacchetti di machine learning (es. `h2o`) permettono di definire i gruppi per la *k-fold* CV all'interno delle loro funzioni. Se ciò non fosse possibile, non spaventatevi, ecco a voi un esempio su come fare:

```
library(randomForest)
## randomForest 4.7-1.2
## Type rfNews() to see new features/changes/bug fixes.
# In questo esempio, utilizziamo l'iris data set per predire
# la "Sepal Length" a partire dalle altre variabili
# utilizzando una Random Forest (che vedremo piu' avanti).
data <- iris
k = 5 # Folds
# Campione da 1 a k, nrow volte (il numero di osservazioni nel dataset)
data$id <- sample(1:k, nrow(data), replace = TRUE)
list <- 1:k
# il "prediction" e "testset" dataframes a cui aggiungiamo ogni
# iterazione di k
prediction <- data.frame()
testsetCopy <- data.frame()
for (i in 1:k){
  trainingset <- subset(data, id %in% list[-i])
  testset <- subset(data, id %in% c(i))
  # modello random forest
  mymodel <- randomForest(trainingset$Sepal.Length ~ .,
                           data = trainingset, ntree = 100)
  # rimuovere la variabile Y da predire
  temp <- as.data.frame(predict(mymodel, testset[, -1]))
  # attaccare questa predizione al dataframe "prediction"
  prediction <- rbind(prediction, temp)
  # attaccare il test set dell'iterazione al data frame
  "testsetCopy"
  # tenere solamente la colonna Sepal Length
  testsetCopy <- rbind(testsetCopy, as.data.frame(testset[, 1]))
}
# aggiungere predizioni e i valori osservati di Sepal Length
```

```

result <- cbind(prediction, testsetCopy[, 1])
names(result) <- c("Predicted", "Actual")
result$Difference <- abs(result$Actual - result$Predicted)
# Come esempio, utilizziamo il Mean Absolute Error per il calcolo
dell'errore
# (vedi piu' avanti per le metriche di errore)
summary(result$Difference)
##      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
## 0.0005586 0.1168428 0.2451925 0.3070998 0.4674791 1.2108659

```

Non vi preoccupate se il funzionamento del modello di `randomForest` vi appare poco chiaro, avremo modo di esplorare in dettaglio i vari algoritmi di apprendimento nei prossimi paragrafi.

Bootstrapping

Un campione *bootstrap* è un campione casuale dei dati presi con metodo di ricampionamento (in inglese **resample**). Ciò significa che, dopo che un'osservazione è stata selezionata per essere inclusa nel sottoinsieme, è ancora disponibile per un'ulteriore selezione. Un campione di *bootstrap* è della stessa dimensione del *set* di dati originale da cui è stato costruito. La figura seguente fornisce uno schema di campionamento del *bootstrap* in cui ogni campione di *bootstrap* contiene 9 osservazioni, proprio come nel *set* di dati originale. Inoltre, il campionamento di *bootstrap* conterrà approssimativamente la stessa distribuzione di valori (rappresentata dai colori) del *set* di dati originale.

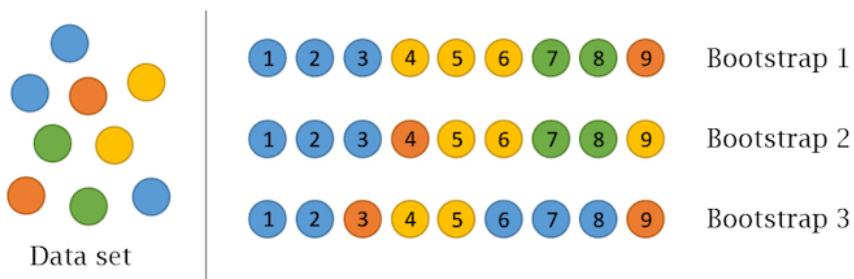


Figura 83 - Esempio schematico di bootstrapping.

Data la procedura di *resample*, ogni campione di *bootstrap* è probabile che contenga valori duplicati. Le osservazioni originali non contenute in un particolare campione di *bootstrap* sono considerate **out-of-bag (OOB)**. Quando si avvia, un modello può essere costruito sui campioni selezionati e validato sui campioni OOB; questo è spesso fatto, ad esempio, nelle *random forest* (vedi più avanti).

Dal momento che le osservazioni sono replicate in *bootstrap*, c'è una minore variabilità nella misura dell'errore rispetto a *k-fold CV*. Tuttavia, questo può anche aumentare la distorsione della stima dell'errore. Questo può essere problematico con *set* di dati più piccoli; tuttavia, per la maggior parte dei *set* di dati medio-grandi (maggiori di 1000) questa preoccupazione è spesso trascurabile.

Per fare delle prove con il campionamento *bootstrapping* possiamo utilizzare semplicemente le funzioni base di R in questo esempio:

```
resample <- sample(1:nrow(iris), round(nrow(iris) * 0.7), replace =
TRUE)
bootstrap_set <- iris[resample, ]
OOB_set <- iris[-resample, ]
```

Stima della performance del modello

L'approccio più valido alla valutazione delle prestazioni del modello consiste nel valutare l'accuratezza predittiva attraverso le **loss function**. Le *loss function* sono metriche che confrontano i valori predetti con il valore reale (l'*output* di una *loss function* è spesso indicato come l'errore o pseudo residuo). Quando eseguiamo metodi di ricampionamento, valutiamo i valori previsti per un insieme di convalida rispetto al valore obiettivo effettivo. Per esempio, in una regressione, un modo per misurare l'errore è prendere la differenza tra il valore effettivo e quello previsto per una data osservazione (questa è la definizione usuale di un residuo nella regressione lineare ordinaria). L'errore complessivo di convalida del modello è calcolato aggregando gli errori nell'intero *set* di dati di convalida.

Ci sono molte *loss function* tra cui scegliere quando si valutano le prestazioni di un modello predittivo, ognuna delle quali fornisce una

comprensione unica della precisione predittiva e differisce tra modelli di regressione e classificazione. Inoltre, il modo in cui una *loss function* è calcolata tenderà a sottolineare alcuni tipi di errori rispetto ad altri e può portare a drastiche differenze nel modo in cui interpretiamo il “modello ottimale”. È importante considerare il contesto del problema quando si identifica la metrica di prestazione preferita da utilizzare. E quando confrontiamo più modelli, dobbiamo confrontarli attraverso la stessa metrica.

Vi presentiamo qui una lista delle metriche più utilizzate per modelli di regressione e classificazione, rispettivamente. La maggior parte dei pacchetti di algoritmi di machine learning permette l'utilizzo di queste metriche attraverso le rispettive funzioni. In alcuni casi, i nomi delle metriche sono mantenuti in inglese per un miglior riscontro con le funzioni in R.

Modelli di regressione

MSE: il **Mean Squared Error** è la stima dell'errore quadratico medio. Il componente quadrato comporta errori più grandi con penalità più grandi. Questa (insieme a l'*RMSE*) è la metrica di errore più comunemente usata.

RMSE: **Root mean squared error**. Questo prende semplicemente la radice quadrata della metrica *MSE*, in modo che il nostro errore sia nelle stesse unità della nostra variabile di risposta.

MAE: **Mean Absolute Error**. Molto simile all'*MSE* ma, invece del quadrato, prende la differenza assoluta media fra i valori reali e previsti. Ciò si traduce in meno enfasi su errori più grandi di *MSE*.

Devianza: abbreviazione di deviazione residua media. In sostanza, fornisce un grado al quale un modello spiega la variazione in un insieme di dati quando si utilizza la stima della massima probabilità. Essenzialmente questo confronta un modello saturo (i.e. completamente caratterizzato) ad un modello insaturo (i.e. intercettare solo o media). Se la distribuzione della variabile di risposta è gaussiana, allora sarà approssimativamente uguale a *MSE*. Quando non lo è, di solito dà una stima più utile di errore. La devianza è spesso utilizzata anche con i modelli di classificazione.

R^2 : questa è una metrica molto popolare che rappresenta la proporzione della varianza nella variabile dipendente che è prevedibile dalla variabile indipendente. Purtroppo, ha diversi limiti. Per esempio, due modelli costruiti da due diversi *set* di dati potrebbero avere lo stesso *RMSE*, ma se uno ha meno variabilità nella variabile di risposta allora avrebbe un minore R^2 rispetto all'altro. Non si dovrebbe porre troppa enfasi su questa metrica nel caso in cui i dati di *training* siano diversi.

Modelli di classificazione

Misclassification: questo è l'errore generale. Per esempio, supponiamo di predire 3 classi (*Blu*, *Giallo*, *Rosso*) e ogni classe ha 15, 40, 35 osservazioni rispettivamente (90 osservazioni totali). Se si classificano male 4 osservazioni di classe *Blu*, 5 di classe *Giallo*, e 3 di classe *Rosso*, allora si misclassificano 12 su 90 osservazioni con conseguente tasso di errore di classificazione del 13.3%.

Media per errore di classe: questo è il tasso di errore medio per ogni classe. Per l'esempio precedente, questa sarebbe la media di 4/15, 5/40, 3/35 (15.9%). Ovviamente, se le classi sono bilanciate, questo errore sarà identico a quello della *misclassification*.

MSE: calcola la distanza tra 1 e la probabilità suggerita. Così, diciamo che abbiamo tre classi, *Blu*, *Giallo*, e *Rosso*, e il modello prevede una probabilità di 0,91 per *Blu*, 0,07 per *Giallo*, e 0,02 per *Rosso*. Se la risposta corretta è *Blu*, allora $MSE = 0.0081$ ($0,09^2$), se è *Giallo* $MSE = 0.8649$ ($0,93^2$), se è *Rosso* $MSE = 0.9604$ ($0,98^2$).

Cross-entropy: simile all'*MSE* ma incorpora un logaritmo della probabilità prevista moltiplicato per la classe vera. Di conseguenza, questa metrica punisce in modo sproporzionato le previsioni in cui prevediamo una piccola probabilità per la vera classe.

Gini index: utilizzato principalmente con metodi a base di **classification tree** e comunemente indicato come misura di *purezza* dove un piccolo valore indica che un nodo contiene prevalentemente osservazioni da una singola classe.

Quando applichiamo i modelli di classificazione, usiamo spesso una **matrice di confusione** per valutare determinate misure di perfor-

mance. Una matrice di confusione è semplicemente una matrice che confronta i livelli categorici effettivi con i livelli categorici previsti. Quando prevediamo il giusto livello, ci riferiamo a questo come ad un vero positivo. Tuttavia, se prevediamo un livello che non è accaduto questo è chiamato un falso positivo. Al contrario, quando non prevediamo un livello e questo non avviene, questo è chiamato un falso negativo.

	No reali	Sì reali
No predetti	Veri negativi	Falsi negativi
Sì predetti	Falsi positivi	Veri positivi

Figura 84 - Matrice di confusione.

Possiamo estrarre diversi livelli di prestazioni per i classificatori binari. Ad esempio, data la matrice di confusione illustrata nella figura precedente, possiamo valutare quanto segue:

Accuracy: nel complesso, quanto spesso il classificatore è corretto?
Esempio: $(VP + VN) / \text{Totale}$.

Precision: con quale precisione il classificatore predice gli eventi? Questa metrica si occupa di massimizzare i veri positivi a falsi positivi. In altre parole, per il numero di previsioni che abbiamo fatto, quante erano corrette? Esempio: $VP / (VP + FP)$.

Sensitivity: con quale precisione il classificatore classifica gli eventi reali? Questa metrica si occupa di massimizzare il rapporto tra veri positivi e falsi negativi. In altre parole, per gli eventi che si sono verificati, quanti ne abbiamo previsti? Esempio: $VP / (VP + FN)$.

Specificity: con quale precisione il modello classifica i non-eventi effettivi? Esempio: $VN / (VN + FP)$.

Concludendo...

Lo sappiamo, tutta questa matematica vi ha spaventati un po'. Man a mano che illustreremo i principali algoritmi di machine learning utilizzati in R, vi renderete conto quanto la conoscenza di queste nozioni sia più utile del riuscire a far girare il vostro personale modello di predizione. Infatti, dall'accortezza che avrete nel sapere dividere i vostri dati e nell'utilizzare la giusta metrica di errore, il vostro modello potrebbe portare a risultati drasticamente diversi! Ma non vi preoccupate, come si dice: *sbagliando si impara*, e dopo numerose prove illustrate in questo capitolo riuscirete meglio a capire come orientarvi in questo labirinto.

Quindi, siamo arrivati al momento che tutti stavate aspettando! Rullo di tamburi e...

Cominciamo a parlare di algoritmi e a smanettare un po' sul codice!

Algoritmi di Machine Learning in R

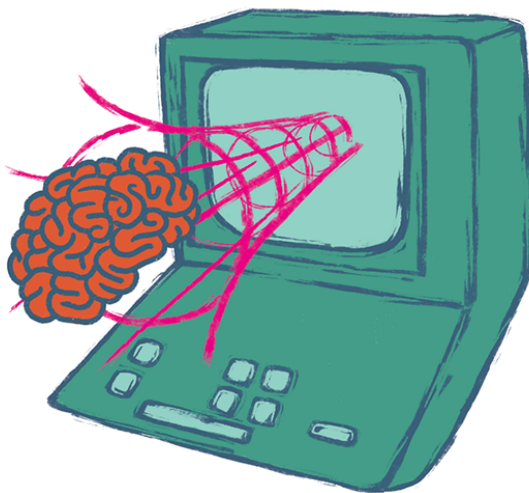


Figura 85 - Nel machine learning si tratta di insegnare al computer come apprendere dai "dati".

Decision Trees

I modelli basati su **Decision Trees** sono una classe di algoritmi non parametrici che funzionano partizionando lo spazio delle funzionalità in un certo numero di regioni più piccole (non sovrapposte) con valori di risposta simili, utilizzando un insieme di regole di suddivisione. Possono essere utilizzati sia per problemi di classificazione che di regressione. Le previsioni sono ottenute adattando un modello più semplice (ad esempio, una costante come il valore medio di risposta) in ogni regione. Tali metodi di *divide-and-conquer* possono produrre semplici regole che sono facili da interpretare e visualizzare con i diagrammi ad albero. Come vedremo, i *decision trees* offrono molti benefici; tuttavia, tipicamente mancano di prestazioni predittive rispetto ad algoritmi più complessi come le reti neurali. Tuttavia, nei prossimi paragrafi vedremo algoritmi di insieme, come le *random forests*, che sono costruiti combinando insieme molti alberi di decisione.

Ci sono molte metodologie per costruire alberi decisionali, ma il più noto è l'**algoritmo di classificazione e regressione (CART)**. Un albero decisionale di base suddivide i dati del *training* in sottogruppi omogenei (cioè gruppi con valori di risposta simili), e quindi inserisce una costante semplice in ogni sottogruppo (ad esempio, la media dei valori di risposta all'interno del gruppo per la regressione). I sottogruppi (chiamati anche nodi) sono formati ricorsivamente usando partizioni binarie formate facendo semplici domande *yes-or-no* su ogni caratteristica (ad esempio, la larghezza del petalo è maggiore di 1,8?). Questo viene fatto un certo numero di volte fino a quando un criterio di arresto adeguato è soddisfatto (ad esempio, una profondità massima dell'albero è raggiunta). Dopo che tutto il partizionamento è stato fatto, il modello predice l'*output* basato su: (1) i valori medi di risposta per tutte le osservazioni che rientrano in quel sottogruppo (problema di regressione), o (2) la classe che ha la rappresentazione maggioritaria (problema di classificazione). Per la classificazione, le probabilità previste possono essere ottenute utilizzando la proporzione di ciascuna classe all'interno del sottogruppo.

Vediamo come costruire un nostro albero per un problema di classificazione utilizzando l'iris *dataset*. Per farlo utilizzeremo la funzione del pacchetto `rpart`:

```

library(rpart)
library(rpart.plot)
set.seed(123)
# creazione training e test set
xdata <- iris
index_n <- sample(1:nrow(xdata), round(nrow(xdata) * 0.7))
training_set <- xdata[index_n, ]
test_set <- xdata[-index_n, ]
# modello
fit <- rpart(Species ~., data = training_set, method = 'class')
# Species ~.: Formula del Decision Trees, vogliamo predire la specie
# sulla base delle altre variabili
# data: i dati di training
# method = 'class': in questo caso, il nostro è un modello di classificazione
rpart.plot(fit)

```

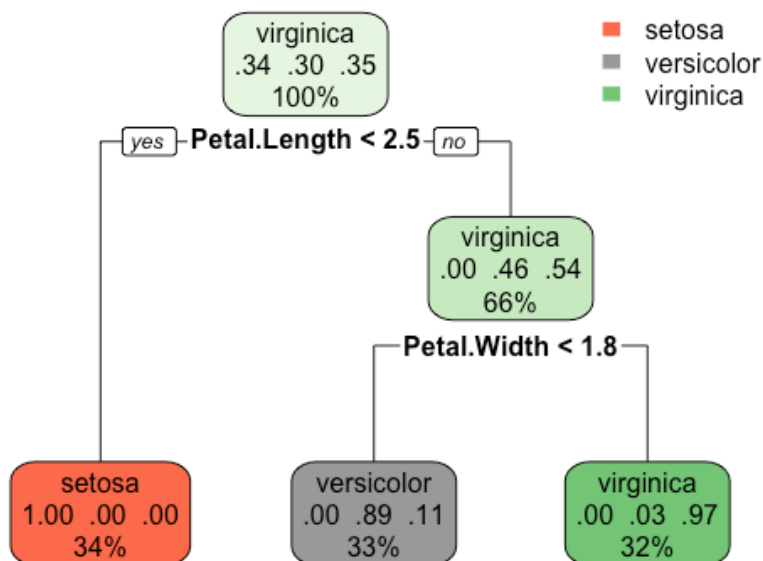


Figura 86 - Rappresentazione del classification tree ottenuto.

Complimenti! Avete appena “costruito” il vostro modello di machine learning! Facile, no? Ma non è il momento di aprirsi una birra e rilassarsi.

Come abbiamo già detto in precedenza, ci sono molte cose da esplorare e da tenere in conto per capire se il nostro modello ha una buona performance predittiva o meno.

Innanzitutto, guardiamo più da vicino il nostro modello grazie al pacchetto `rpart.plot`. Gli algoritmi di *decision tree* permettono di poter guardare facilmente all'interno degli ingranaggi del modello. Cosa che è molto più difficile fare con modelli di reti neurali e *random forests* (chiamati in gergo, per l'appunto, **black-boxes**). Partendo dal nodo **radice** (in alto) possiamo osservare:

- in cima, abbiamo la probabilità complessiva di appartenenza alle classi;
- il nodo successivo “chiede” se il `Petal Length` è maggiore di 2,5. Se sì, allora si scende al nodo figlio sinistro della radice, dove il 100% degli individui appartiene alla specie *I. setosa*;
- nel secondo nodo, si chiede se il `Petal width` è minore di 1,8. Se sì, allora la probabilità di che si tratti di *I. versicolor* è dell'89%. Altrimenti c'è una probabilità del 97% che si tratti di *I. virginica*.

Si noti che, una delle molte qualità dei *Decision Trees* è che richiedono una preparazione dei dati molto limitata. In particolare, non richiedono ridimensionamento.

Per impostazione predefinita, la funzione `rpart()` usa la misura del Gini Index per dividere i nodi. Più alto è il coefficiente di Gini, più diverse sono le istanze all'interno del nodo.

Proviamo a predire i nostri dati di *test* utilizzando il modello appena costruito.

```
predicted <- predict(fit, test_set, type = 'class')
table_mat <- table(test_set$Species, predicted)
table_mat
```

##		predicted		
##		setosa	versicolor	virginica
##	setosa	14	0	0
##	versicolor	0	18	0
##	virginica	0	1	12

WOW! Il nostro modello ha predetto le nostre classi molto bene! Ma con quale **Accuracy** di preciso? Vediamo:

```
accuracy_Test <- sum(diag(table_mat)) / sum(table_mat)
print(paste('Accuracy for test', accuracy_Test))
## [1] "Accuracy for test 0.977777777777778"
```

Possiamo dire: ottimo! Il nostro modello ha un'accuratezza del 97% sui dati indipendenti. Più che soddisfacente!

I *decision trees* in R hanno vari parametri che controllano gli aspetti del *fit*. Nella libreria `rpart` è possibile controllare questi parametri utilizzando la funzione `rpart.control()`. Qui di seguito, ci sono alcuni parametri che potete regolare:

- `minsplit`: imposta il numero minimo di osservazioni nel nodo prima che l'algoritmo esegua una divisione.
- `minbucket`: imposta il numero minimo di osservazioni nella nota finale, ossia *la foglia* dell'albero.
- `maxdepth`: imposta la profondità massima di qualsiasi nodo dell'albero finale. Il nodo radice è trattato con una profondità 0.

Random Forests

Le **Random Forests** consistono in un insieme di algoritmi che costruiscono una vasta collezione di alberi de-correlati per migliorare ulteriormente le prestazioni predittive. Sono diventati un algoritmo di apprendimento "*out-of-the-box*" molto popolare che gode di buone prestazioni predittive con relativamente poca sintonizzazione degli iperparametri. Esistono molte implementazioni moderne di *random forests*; tuttavia, l'algoritmo di Leo Breiman è diventato in gran parte la procedura più autorevole.

Le *random forests* sono costruite utilizzando gli stessi principi degli alberi di decisione. L'algoritmo di costruzione degli alberi introduce un componente casuale nel processo di apprendimento, costruendo molti alberi su copie *bootstrapped* dei dati di formazione. Questa aggregazione riduce la varianza della procedura complessiva e si traduce in una migliore performance predittiva.

Lo schema di campionamento predefinito per le *random forests* è un *bootstrapping* dove il 100% delle osservazioni sono campionate con la

sostituzione (in altre parole, ogni copia *bootstrap* ha le stesse dimensioni dei dati di *training* originali); tuttavia, possiamo regolare sia la dimensione del campione sia se campionare con o senza sostituzione. Il parametro della dimensione del campione determina il numero di osservazioni da effettuare per l'addestramento di ciascun albero. Diminuendo la dimensione del campione si ottengono alberi più diversi e quindi si riduce la correlazione tra gli alberi, il che può avere un effetto positivo sulla precisione di previsione. Di conseguenza, se ci sono alcune caratteristiche dominanti nel vostro insieme di dati, la riduzione della dimensione del campione può anche aiutare a ridurre al minimo la correlazione tra gli alberi.

Anche le *random forests* possono essere utilizzate per problemi di classificazione e regressione. Dato che abbiamo già utilizzato l'esempio con l'*iris dataset*, questa volta proviamo a costruire il nostro modello utilizzando una variabile di *output* continua (e quindi una regressione).

In questo caso, quindi, utilizziamo l'*mtcars dataset* e, prima di addestrare il modello, normalizziamo i nostri dati in un intervallo tra 0 e 1 per facilitare il processo di apprendimento.

```
# normalizzazione dei dati
library(vegan)
xdata <- mtcars
xdata_n <- decostand(xdata, method = "range")
head(xdata_n)
```

##		mpg	cyl	disp	hp	drat	wt
##	Mazda RX4	0.4510638	0.5	0.2217511	0.2049470	0.5253456	0.2830478
##	Mazda RX4 Wag	0.4510638	0.5	0.2217511	0.2049470	0.5253456	0.3482485
##	Datsun 710	0.5276596	0.0	0.0920429	0.1448763	0.5023041	0.2063411
##	Hornet 4 Drive	0.4680851	0.5	0.4662010	0.2049470	0.1474654	0.4351828
##	Hornet Sportabout	0.3531915	1.0	0.7206286	0.4346290	0.1797235	0.4927129
##	Valiant	0.3276596	0.5	0.3838863	0.1872792	0.0000000	0.4978266

##		qsec	vs	am	gear	carb
##	Mazda RX4	0.2333333	0	1	0.5	0.4285714
##	Mazda RX4 Wag	0.3000000	0	1	0.5	0.4285714
##	Datsun 710	0.4892857	1	1	0.5	0.0000000
##	Hornet 4 Drive	0.5880952	1	0	0.0	0.0000000

```
##   Hornet   Sportabout   0.3000000   0   0   0.0   0.1428571
## Valiant           0.6809524   1   0   0.0  0.0000000
```

Bene. Ora che i nostri dati sono normalizzati, passiamo alla costruzione del modello:

```
library(randomForest)
# creazione training e test set
index_n <- sample(1:nrow(xdata_n), round(nrow(xdata_n) * 0.7))
training_set <- xdata_n[index_n, ]
test_set <- xdata_n[-index_n, ]
# modello
rf.fit <- randomForest(mpg ~., data = training_set,
                        ntree = 500, # numero di alberi
                        mtry = 3) # numero di variabili predittive
                                # (scelte casualmente) che verranno
                                # utilizzate per la costruzione di
ogni albero
print(rf.fit)
##
## Call:
## randomForest(formula = mpg ~ ., data = training_set, ntree =
500,      mtry = 3)
##
##           Type of random forest: regression
##           Number of trees: 500
## No. of variables tried at each split: 3
##
##           Mean of squared residuals: 0.01401642
##           % Var explained: 79.95
```

Possiamo osservare già da subito alcune cose. Sotto la voce *Type* è indicato *regression*. Il modello ha riconosciuto da solo che si tratta di una variabile numerica continua. Inoltre, il modello calcola l'*RMSE* sui dati di *training* utilizzando le osservazioni **out-of-bag (OOB)** (vedi sopra: **Metodi di ricampionamento**) per la stima dell'errore, dato che il processo di campionamento dei dati avviene attraverso un *bootstrapping*.

Adesso, utilizziamo il modello per predire le nuove osservazioni e calcolare l'errore di predizione utilizzando questa volta R^2 :

```
# togliamo dal test_set la colonna con i valori da predire
rf.pred <- predict(rf.fit, newdata = test_set[-1])
obsPred <- cbind(rf.pred, test_set[, 1])
colnames(obsPred) <- c("Predicted", "Observed")
obsPred
##                Predicted Observed
## Merc 240D           0.5428932 0.5957447
## Merc 280C           0.3999806 0.3148936
## Merc 450SE          0.2168960 0.2553191
## Merc 450SL          0.2160671 0.2936170
## Cadillac Fleetwood 0.1264838 0.0000000
## Dodge Challenger    0.2500200 0.2170213
## AMC Javelin         0.3031936 0.2042553
## Camaro Z28          0.2081350 0.1234043
## Pontiac Firebird    0.2607244 0.3744681
## Ferrari Dino        0.3971294 0.3957447
rss <- sum((rf.pred - test_set[, 1]) ^ 2) ## residui della somma
dei quadrati
tss <- sum((test_set[, 1] - mean(test_set[, 1])) ^ 2) ## somma
totale dei quadrati
rsq <- 1 - rss/tss
rsq
## [1] 0.7272982
```

Ricordate il discorso delle *black-boxes*? Vi sarete resi conto che, a meno che non andiamo a vedere da vicino ognuno dei 500 alberi (fol-
lia!!!), non è possibile vedere quali variabili predittive incidano di più
sul processo predittivo. Per ovviare a ciò, il pacchetto `randomForest` offre
una funzione che stima l'importanza delle variabili.

```
varImpPlot(rf.fit)
```

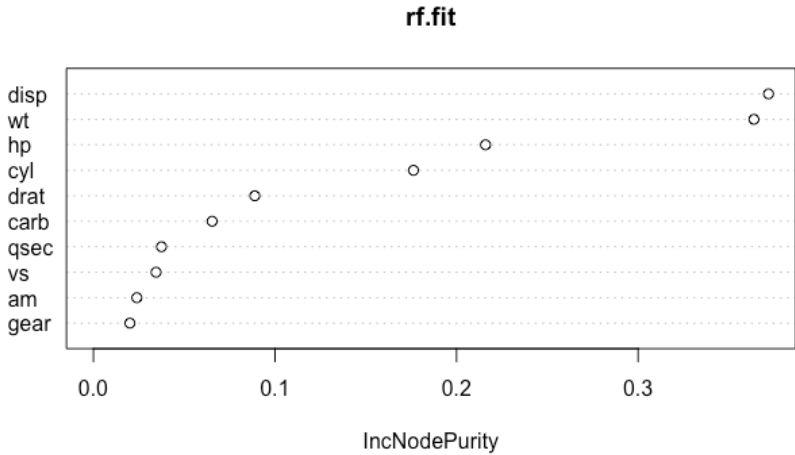



Figura 87 - Rappresentazione dell'importanza delle variabili predittive.

L'importanza di ciascuna variabile è valutata sulla base di due criteri:

MeanDecreaseAccuracy: fornisce una stima approssimativa della perdita di prestazioni predittive quando tale variabile particolare viene omessa dall'insieme degli allenamenti. Avvertenza: se due variabili sono un po' ridondanti, allora omettere una di esse potrebbe non portare ad enormi guadagni nelle prestazioni di previsione, ma renderebbe la seconda variabile più importante.

MeandecreaseGini: Gini è una misura dell'impurità dei nodi. Pensatela in questo modo: se usate questa funzione per dividere i dati, quanto saranno puri i nodi? Massima purezza significa che ogni nodo contiene solo elementi di una singola classe. La valutazione della diminuzione di Gini quando quella caratteristica è omessa conduce ad una comprensione di quanto importante quella caratteristica sia per dividere correttamente i dati.

Si noti che queste misure sono utilizzate per classificare le variabili in termini di importanza e, quindi, i loro valori assoluti potrebbero essere ignorati.

4



(Cenni di) Analisi genetiche in R

Parlare di genetica fa spesso venire in mente grandi laboratori con banconi in marmo pieni di provette, reagenti, pipette e personale in camice bianco. Tuttavia, un importante aspetto della genetica e delle sue applicazioni (come la genetica di popolazione o la genetica evolutiva) risiede nell'analisi numerica. Basti pensare che *Gregor Mendel*, uno dei padri della genetica, teorizzò la presenza di unità ereditarie, i geni, basandosi solo sull'osservazione e sul conteggio di piselli di diversa forma e colore!

Nell'introduzione al suo libro dedicato alle analisi filogenetiche, Emmanuel Paradis evidenzia alcune caratteristiche che fanno di *R* un potente strumento per l'analisi di dati genetici, soprattutto nei campi della biologia evolutiva (nelle sue diverse sfumature) e dell'ecologia molecolare. I quattro aspetti fondamentali sono la possibilità d'**integrazione**, l'**interattività**, la **programmabilità** e, ultimo ma non per importanza, l'**evolvibilità**.

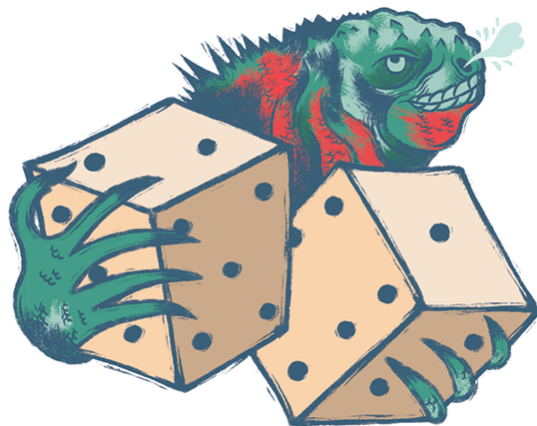


Figura 88 - La genetica ed il ruolo del caso.

Questo capitolo non è un compendio di genetica, né tantomeno una fonte esaustiva per la descrizione di tutte le possibili analisi genetiche che si possono fare in *R*. Un capitolo basterebbe a malapena per la trattazione dettagliata di tutti i vari formati e classi di dati utilizzate per rappresentare e per gestire l'informazione genetica in una sessione di lavoro in *R*. Questo è ancora più vero da quando, con l'introduzione circa 15 anni fa delle nuove tecnologie di sequenziamento in parallelo e con la sempre maggiore disponibilità di potenza di calcolo a prezzi relativamente contenuti, si sta effettuando una transizione da analisi **gen-etiche** ad analisi **gen-omiche**. La differenza tra i due approcci analitici, fondamentalmente, risiede nella quantità di dati che vengono utilizzati (e di conseguenza nel tipo di problemi, metodi e quesiti scientifici che possono essere proposti ed analizzati). Si passa infatti dalla produzione di pochi frammenti relativamente piccoli di DNA, tipici del sequenziamento Sanger, alla produzione di fino a 20 Tb¹ di basi nucleotidiche organizzate in frammenti più o meno grandi a seconda della tecnologia utilizzata.

¹ È sempre utile ricordare che l'unità base delle analisi genetica è la base, simboleggiata con la lettera "*b*", da non confondere con la lettera "*B*", utilizzata nel linguaggio informatico indicare il byte. Per semplificare, 1 Mb ≠ 1 MB, con 1 Mb = 10⁶ basi, e 1 MB = 10⁶ bytes.

Definizione	Simbolo	Basi
Kilobase	1 kb	1,000
Megabase	1 Mb	1,000,000
Gigabase	1 Gb	1,000,000,000
Terabase	1 Tb	1,000,000,000,000
Petabase	1 Pb	1,000,000,000,000,000

Tabella 4 - Notazione generalmente utilizzata per descrivere il numero di basi nucleotidiche prodotte dalle diverse tecnologie di sequenziamento. Se la vecchia tecnologia Sanger permetteva la produzione di frammenti con ordine di grandezza in kb, le più recenti tecnologie di sequenziamento, come ad esempio la ONT (Oxford Nanopore Technology), consente la produzione fino a 20 Tb di basi.

In questo capitolo utilizzeremo *dataset* liberamente accessibili per effettuare una carrellata di analisi genetiche tra le più comuni, ma non per questo banali, che si possono fare in *R*. Lo scopo, quindi, non è quello di una guida esaustiva sull'argomento, ma più che altro un'introduzione alla possibilità di utilizzare *R* anche per le analisi genetiche e genomiche. Prima di cominciare è doveroso fare una dovuta precisazione. La maggior parte dei contenuti presentati nel corso del capitolo, incluso il codice per analizzare i dati, ricalcano i contenuti di due ottimi testi, in inglese, ai quali si rimandano i lettori più interessati ad approfondire sia i dettagli matematici che quelli analitici e procedurali dell'analisi di dati genetici/genomici in *R*.

Pacchetti da installare

Nel corso del capitolo faremo largo uso di alcuni pacchetti che offrono numerose funzioni per effettuare analisi genetiche a largo spettro, dalla semplice analisi delle frequenze alleliche e genotipiche alla costruzione di *Skyline Plots*. In particolare, i pacchetti che si raccomanda di installare e caricare nell'ambiente di lavoro sono: **pegas**, **ape**, **adegenet**, **coalescentMCMC**, **phangorn**, **Rsamtools**, **ips**. La maggior parte dei pacchetti sono disponibili direttamente sul *CRAN* e sarà quindi possibile installarli utilizzando l'apposito comando. Per altri si consiglia di leggere la documentazione relativa.

Una volta installati correttamente sarà possibile caricare tutte librerie necessarie nella sessione di lavoro corrente con il seguente *script*:

```
useful_packs <- c("pegas", "ape", "adegenet",
                  "coalescentMCMC", "phangorn", "Rsamtools",
                  "ips", "vcfR")
lapply(useful_packs, require, character.only = T)
rm(useful_packs)
```

Per completezza e riproducibilità, riporto qui sotto le informazioni della versione di R e dei suoi pacchetti al momento della scrittura di questo capitolo.

```
sessionInfo()
## R version 4.4.2 (2024-10-31)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sequoia 15.3.2
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/
Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/
Resources/lib/libRlapack.dylib; LAPACK version 3.12.0
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
```

```
##
## time zone: Europe/Rome
## tzcode source: internal
##
## attached base packages:
## [1] stats4      stats      graphics  grDevices  utils      datasets  methods
## [8] base
##
## other attached packages:
## [1] phangorn_2.12.1      adegenet_2.1.10      ade4_1.7-22
## [4] pegas_1.3            ips_0.0.12           Rsamtools_2.22.0
## [7] Biostrings_2.74.1    XVector_0.46.0       GenomicRanges_1.58.0
## [10] GenomeInfoDb_1.42.3  IRanges_2.40.1       S4Vectors_0.44.0
## [13] BiocGenerics_0.52.0  coalescentMCMC_0.4-4 coda_0.19-4.1
## [16] ape_5.8-1            vcfR_1.15.0          seqinr_4.2-36
## [19] dplyr_1.1.4          vegan_2.6-10         lattice_0.22-6
## [22] permute_0.9-7        rpart.plot_3.1.2     rpart_4.1.24
## [25] randomForest_4.7-1.2 wakefield_0.3.6      rsample_1.2.1
## [28] DescTools_0.99.59    car_3.1-3            carData_3.0-5
## [31] reshape2_1.4.4
##
## loaded via a namespace (and not attached):
## [1] bitops_1.0-9          gld_2.6.7            readxl_1.4.3
## [4] rlang_1.1.5           magrittr_2.0.3       furrr_0.3.1
## [7] e1071_1.7-16          compiler_4.4.2       mgcv_1.9-1
## [10] vctrs_0.6.5           quadprog_1.5-8       stringr_1.5.1
## [13] pkgconfig_2.0.3       crayon_1.5.3         fastmap_1.2.0
## [16] promises_1.3.2        rmarkdown_2.29       haven_2.5.4
## [19] UCSC.utils_1.2.0      purrr_1.0.2          xfun_0.50
## [22] zlibbioc_1.52.0       jsonlite_1.8.9       later_1.4.1
## [25] BiocParallel_1.40.0   parallel_4.4.2       cluster_2.1.8
## [28] R6_2.5.1              stringi_1.8.4         parallely_1.42.0
## [31] boot_1.3-31           cellranger_1.1.0     Rcpp_1.0.14
## [34] bookdown_0.42         knitr_1.49           httpuv_1.6.15
## [37] Matrix_1.7-2          splines_4.4.2        igraph_2.1.4
## [40] tidyselect_1.2.1      rstudioapi_0.17.1    abind_1.4-8
```

## [43] yaml_2.3.10	codetools_0.2-20	listenv_0.9.1
## [46] tibble_3.2.1	plyr_1.8.9	shiny_1.10.0
## [49] withr_3.0.2	evaluate_1.0.3	pinfsc50_1.3.0
## [52] future_1.34.0	proxy_0.4-27	pillar_1.10.1
## [55] generics_0.1.3	ggplot2_3.5.1	hms_1.1.3
## [58] munsell_0.5.1	scales_1.3.0	rootSolve_1.8.2.4
## [61] xtable_1.8-4	globals_0.16.3	class_7.3-23
## [64] glue_1.8.0	lmom_3.2	tools_4.4.2
## [67] data.table_1.16.4	forcats_1.0.0	Exact_3.3
## [70] mvtnorm_1.3-3	XML_3.99-0.18	fastmatch_1.1-6
## [73] grid_4.4.2	tidyr_1.3.1	colorspace_2.1-1
## [76] nlme_3.1-167	GenomeInfoDbData_1.2.13	Formula_1.2-5
## [79] cli_3.6.3	expm_1.0-0	viridisLite_0.4.2
## [82] gtable_0.3.6	digest_0.6.37	htmltools_0.5.8.1
## [85] lifecycle_1.0.4	httr_1.4.7	mime_0.12
## [88] MASS_7.3-64		

Software di terze parti

Per il corretto svolgimento di tutti gli esempi presentati è necessario installare alcuni software aggiuntivi. La nota positiva è che useremo delle funzioni in *R* che richiamano gli eseguibili di questi software senza dover lasciare il nostro terminale. In particolare, i software di cui faremo uso sono **mafft**, un software per l'allineamento delle sequenze di DNA o RNA, e **PhyML**, un software che, a partire da sequenze nucleotidiche, ci aiuta a stimare il corretto modello evolutivo del DNA in esame. Le istruzioni per l'installazione di questi software esulano dallo scopo di questo capitolo e si confida nella capacità del lettore per la loro corretta installazione ed integrazione con *R*.

Datasets

In questo capitolo faremo utilizzo di 3 *dataset* principali. Di seguito alcune informazioni e riferimenti generali sui *dataset* e le istruzioni su come accedere ai dati e prepararli per le analisi successive.

Uccelli canori del genere *Sylvia*

Il primo *set* di dati è composto da 25 specie differenti di uccelli canori appartenenti al genere *Sylvia*. Le sequenze del gene *cytochrome b*, depositate su GenBank, verranno usate per ricostruire una filogenesi dei campioni di questo genere.

Il codice che segue serve per ottenere il *dataset* e formattarlo secondo le nostre esigenze. Usiamo la funzione `read.GenBank` dal pacchetto `ape` per accedere al database dell'NCBI e scaricare le sequenze che ci interessano. I codici identificativi delle sequenze sono disponibili nella pubblicazione originale.

```
# vettore con i codici identificativi di GenBank
sw_seq <- paste("AJ5345", 26:49, sep = "")
sw_seq <- c("Z73494", sw_seq)
# eseguiamo funzione per scaricare da GenBank
# è necessaria una connessione ad internet!
sw_seq <- read.GenBank(sw_seq)
# ispezioniamo il nostro oggetto
sw_seq

## 25 DNA sequences in binary format stored in a list.
##
## Mean sequence length: 1134.84
##   Shortest sequence: 1041
##   Longest sequence: 1143
##
## Labels:
## Z73494
## AJ534526
## AJ534527
## AJ534528
## AJ534529
## AJ534530
## ...
##
## Base composition:
```



```
##      a      c      g      t
## 0.270 0.354 0.131 0.245
## (Total: 28.37 kb)
```

Le sequenze, alcune più lunghe e altre più corte, sono state correttamente scaricate. Ispezionando i *file* scaricati ci rendiamo conto che i nomi specifici rivelano la presenza di più generi nel nostro *dataset*.

```
head(attr(sw_seq, "species"))
## [1] "Sylvia_atricapilla_atricapilla" "Chamaea_fasciata"
## [3] "Sylvia_nisoria"                "Sylvia_layardi"
## [5] "Devioeca_papuana"              "Sylvia_boehmi"
```

Gli identificativi delle sequenze, al contrario, indicano tutte specie appartenenti al genere *Sylvia* tranne il secondo elemento, che è una sequenza usata come *outgroup* dagli autori.

```
head(attr(sw_seq, "description"))
## [1] "Z73494.2 Sylvia atricapilla atricapilla mitochondrial cytb
gene for cytochrome b"
## [2] "AJ534526.1 Chamaea fasciata mitochondrial cytb gene for
cytochrome b"
## [3] "AJ534527.1 Sylvia nisoria mitochondrial cytb gene for
cytochrome b"
## [4] "AJ534528.1 Sylvia layardi mitochondrial cytb gene for
cytochrome b"
## [5] "AJ534529.1 Sylvia subcaeruleum mitochondrial cytb gene for
cytochrome b"
## [6] "AJ534530.1 Sylvia boehmi mitochondrial cytb gene for
cytochrome b"
```

Creiamo un *file* in cui manteniamo il codice identificativo di GenBank associato solo al nome della specie preso dalla descrizione delle sequenze.

```
taxaName <- attr(sw_seq, "description")
taxaName <- lapply(lapply(taxaName, strsplit, " "), unlist)
```

```
taxaName <- unlist(lapply(taxaName,
                          function(x) paste(x[2], x[3], sep = " ")))
names(taxaName) <- names(sw_seq)
head(taxaName, 10)
##           Z73494           AJ534526           AJ534527
## "Sylvia atricapilla" "Chamaea fasciata" "Sylvia nisoria"
##           AJ534528           AJ534529           AJ534530
## "Sylvia layardi" "Sylvia subcaeruleum" "Sylvia boehmi"
##           AJ534531           AJ534532           AJ534533
## "Sylvia buryi" "Sylvia lugens" "Sylvia leucomelaena"
##           AJ534534
## "Sylvia hortensis"
```

Il *dataset* è ora pronto per un'analisi filogenetica.

mtDNA Gatto Dorato Asiatico

Il secondo *dataset* comprende 40 genomi mitocondriali di altrettanti individui di *Catopuma temminckii*, il gatto dorato asiatico. I genomi sono anch'essi depositati su GenBank con numeri identificativi che vanno da KX224490 a KX224529. Possiamo procedere a scaricare i dati utilizzando lo stesso tipo di codice impiegato per gli uccelli canori.

```
agc_id <- paste("KX224", 490:529, sep = "")
agc_mtG <- read.GenBank(agc_id)
agc_mtG
## 40 DNA sequences in binary format stored in a list.
##
## All sequences of same length: 15582
##
## Labels:
## KX224490
## KX224491
## KX224492
## KX224493
```

```
## KX224494
## KX224495
## ...
##
## Base composition:
##      a      c      g      t
## 0.329 0.258 0.139 0.274
## (Total: 623.28 kb)
```

Le sequenze sono state correttamente scaricate. Avanti col prossimo *dataset*.

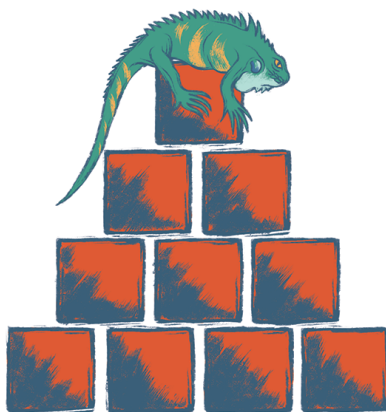


Figura 89 - La scala della diversità.

Microsatelliti giaguaro

Ottenere il terzo *dataset* è più semplice dal momento che fa parte di uno dei *dataset* precompilati all'interno di un pacchetto. Il *dataset* comprende 59 individui di giaguaro (*Panthera onca*) campionati da 4 popolazioni e tipizzati con 13 marcatori nucleari polimorfici (microsatelliti). I dati possono essere caricati nella nostra sessione di lavoro dopo aver caricato il pacchetto *pegas*.

```
# richiamiamo il dataset
```

```

data(jaguar)
# ispezioniamo il dataset
str(jaguar)
## Classes 'loci' and 'data.frame': 59 obs. of 14 variables:
## $ FCA742 : Factor w/ 35 levels "0/0","0/170",...: 24 27 18 13
31 24 28 30 1 30 ...
## $ FCA723 : Factor w/ 13 levels "0/0","220/236",...: 9 5 5 4 9
6 10 1 7 8 ...
## $ FCA740 : Factor w/ 11 levels "0/0","300/300",...: 9 9 4 8 11
9 4 10 1 7 ...
## $ FCA441 : Factor w/ 11 levels "157/165","157/169",...: 5 8 8
1 1 7 8 1 4 4 ...
## $ FCA391 : Factor w/ 21 levels "0/0","215/223",...: 7 5 19 7
19 21 20 12 1 19 ...
## $ F98 : Factor w/ 8 levels "0/0","189/189",...: 3 3 3 3 3 5
7 3 5 5 ...
## $ F53 : Factor w/ 23 levels "0/0","160/160",...: 8 22 7 4
17 5 4 16 2 12 ...
## $ F124 : Factor w/ 20 levels "0/0","199/211",...: 7 9 4 10 6
13 13 1 16 13 ...
## $ F146 : Factor w/ 9 levels "170/170","170/173",...: 5 5 9 7
4 4 7 5 8 1 ...
## $ F85 : Factor w/ 27 levels "0/0","0/147",...: 7 7 23 17 9
24 2 20 25 3 ...
## $ F42 : Factor w/ 20 levels "0/0","247/267",...: 15 13 10
14 20 17 20 4 12 16 ...
## $ FCA453 : Factor w/ 17 levels "0/0","0/192",...: 7 11 5 5 7
16 2 9 2 3 ...
## $ FCA741 : Factor w/ 7 levels "175/175","175/179",...: 5 5 6 2
2 2 5 4 1 7 ...
## $ population: Factor w/ 4 levels "Green Corridor",...: 1 1 1 1 1
1 1 1 1 1 ...
## - attr(*, "locicol")= int [1:13] 1 2 3 4 5 6 7 8 9 10 ...

```

Al momento non ci soffermiamo troppo sulla classe di dati, ma ci basta sapere che sono stati caricati correttamente e sono pronti all'uso.

Approccio descrittivo

Prima di avventurarsi in analisi più complesse, è sempre meglio esplorare e descrivere i propri dati. Ad esempio, in presenza di *dataset* composti da un elevato numero di individui e marcatori molecolari potrebbe essere molto utile descrivere la variabilità genetica presente nel nostro campione.

Variabilità in marcatori altamente polimorfici

Analizziamo la variabilità del *dataset* di microsatelliti tipizzati in 59 individui di giaguaro. I microsatelliti sono marcatori polimorfici che generalmente presentano un elevato grado di variabilità. Iniziamo quindi con il calcolare il numero di alleli e genotipi presenti nel *dataset*. Due funzioni del pacchetto *pegas* fanno al caso nostro.

```
lengths(getAlleles(jaguar))
## FCA742 FCA723 FCA740 FCA441 FCA391 F98 F53 F124 F146 F85 F42
##      16      7      6      5      9      5     11      9      5     14     10
##                                     FCA453                                     FCA741
##              6              4
lengths(getGenotypes(jaguar))
## FCA742 FCA723 FCA740 FCA441 FCA391 F98 F53 F124 F146 F85 F42
##      35     13     11     11     21     8     23     20      9     27     20
##                                     FCA453                                     FCA741
##      17          7
```

Come era prevedibile, il *dataset* presenta un elevato numero di alleli e genotipi in ognuno dei marcatori molecolari. Per una più rapida visualizzazione della distribuzione delle frequenze alleliche è anche possibile presentare i risultati sotto forma di grafici.

```
s <- summary(jaguar)
plot(s, what = "alleles",
      layout = 16, col = "gray", las = 2)
```

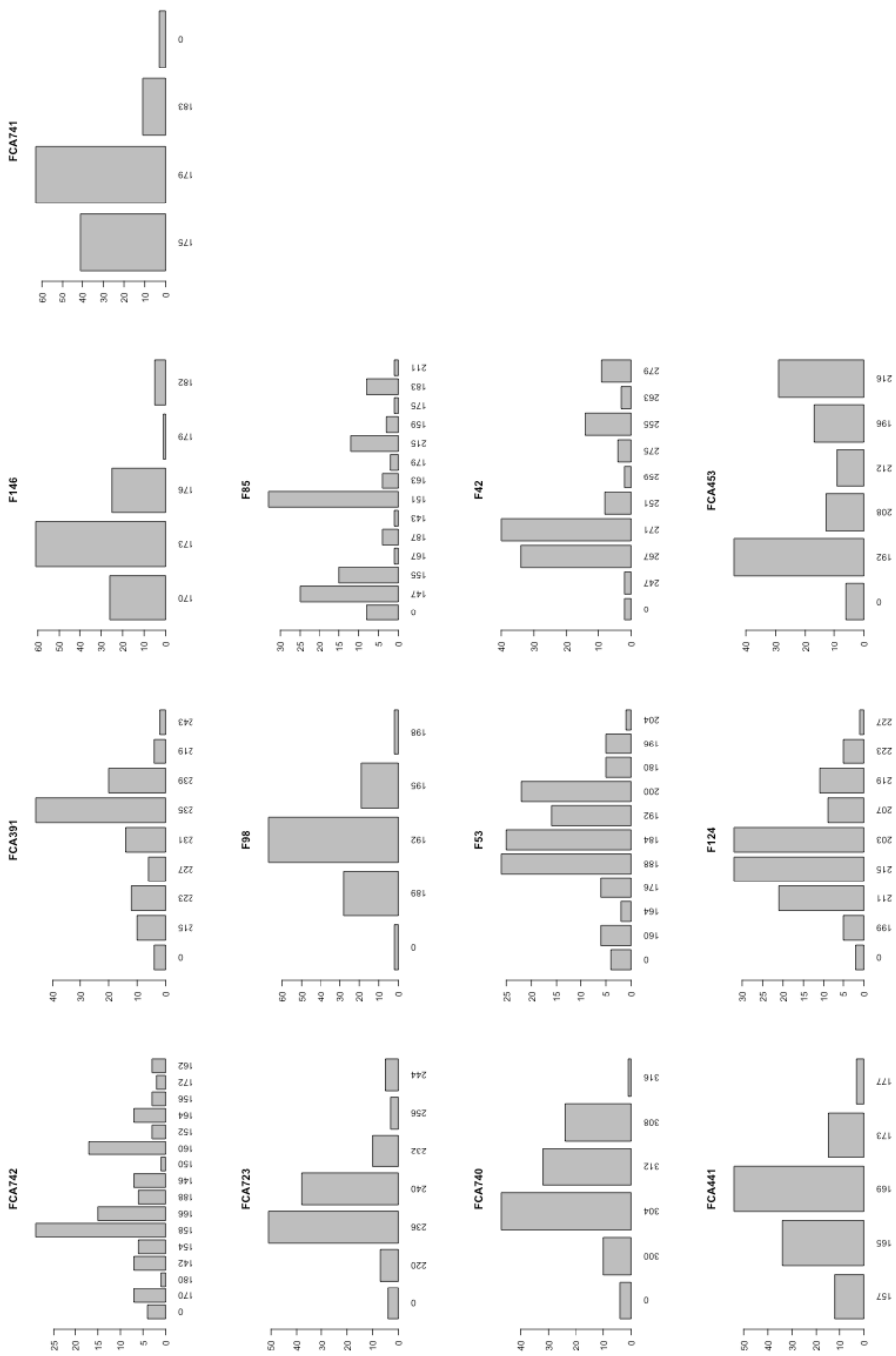


Figura 90 - Frequenze alleliche dei microsatelliti del dataset di giaguari.

Queste informazioni sono molto interessanti, ma non considerano un aspetto importante del nostro *dataset*. I campioni sono stati raccolti in 4 popolazioni differenti, quindi potrebbe essere interessante andare a valutare la variabilità per popolazione, ad esempio calcolando il numero di alleli per locus per popolazione.

```
allelicrichness(jaguar, method = "raw")
```

##	FCA742	FCA723	FCA740	FCA441	FCA391	F98	F53	F124	F146	F85	F42
## Green Corridor	10	6	6	4	8	4	10	8	4	12	9
## Morro do Diabo	4	2	3	2	4	3	4	4	3	3	5
## Ivinhema	8	2	4	4	5	4	7	6	3	6	3
## Porto Primavera	10	6	4	4	6	3	8	7	4	7	6

##	FCA453	FCA741
## Green Corridor	6	4
## Morro do Diabo	3	1
## Ivinhema	3	3
## Porto Primavera	5	3

Il risultato ci dice chiaramente che alcune popolazioni presentano una ricchezza allelica molto più elevata rispetto ad altre. Possiamo usare quest'informazione preliminare per descrivere come questa variabilità contribuisca a separare le varie popolazioni le une dalle altre. Procediamo quindi a fare un'analisi delle componenti principali.

```
X <- loci2genind(na.omit(jaguar))
acp.jaguar <- prcomp(X@tab, scale. = T)
vars <- acp.jaguar$sdev^2
head(round(vars/sum(vars), 3), 5)
## [1] 0.100 0.076 0.066 0.060 0.053
```

Le prime due componenti principali spiegano quasi il 20% della varianza tra i campioni di giaguaro.

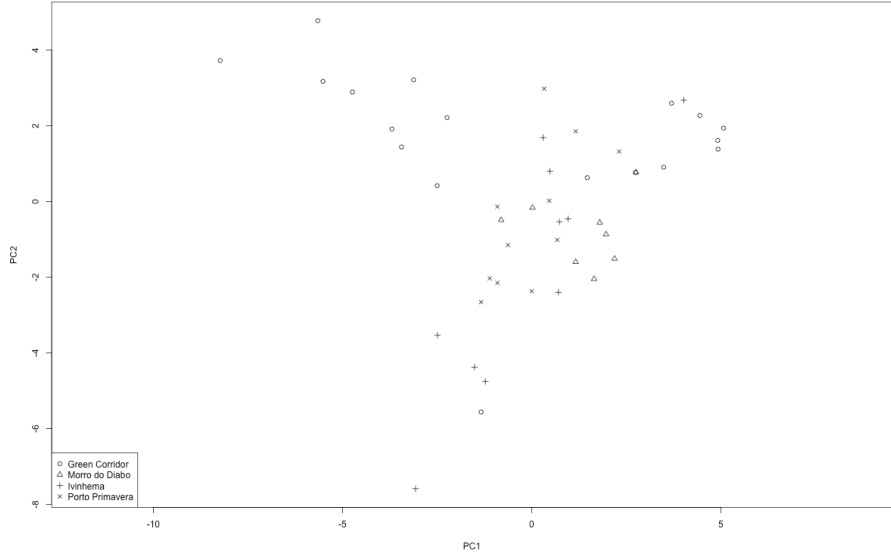


Figura 91 - Analisi delle componenti principali nel dataset di microsatelliti di giaguaro.

Variabilità aplotipica

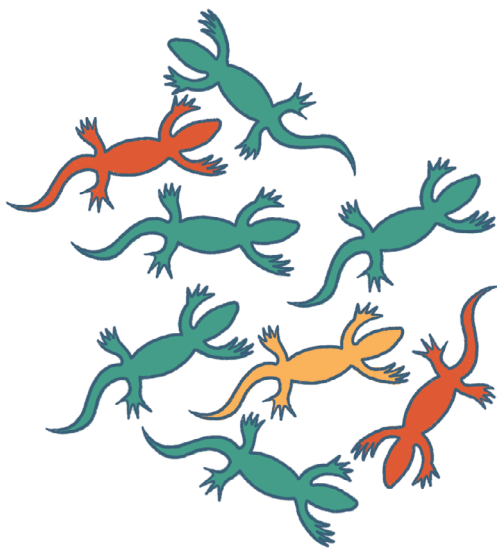


Figura 92 - La bellezza della diversità.

Un approccio descrittivo simile può essere utilizzato per descrivere la variabilità di sequenze di DNA. Valutiamo, ad esempio, la variabilità

genetica del *dataset* composto da individui di Gatto Dorato Asiatico. Il *dataset* è composto da 40 individui e per ogni individuo abbiamo a disposizione l'intero genoma mitocondriale. Un primo passaggio per valutare la variabilità delle sequenze di DNA è quello di allineare i vari genomi insieme. Per effettuare l'allineamento usiamo software `mafft` attraverso l'interfaccia fornita dal pacchetto `ips`².

```
agc_mtG_ali <- ips::mafft(agc_mtG)
checkAlignment(agc_mtG_ali, plot = F)
##
## Number of sequences: 40
## Number of sites: 15582
##
## No gap in alignment.
##
## Number of segregating sites (including gaps): 226
## Number of sites with at least one substitution: 226
## Number of sites with 1, 2, 3 or 4 observed bases:
##      1      2      3      4
## 15356   226      0      0
```

Il resoconto dell'allineamento ci dice che ci sono almeno 226 posizioni nelle quali abbiamo almeno 1 sostituzione. In effetti, in 226 siti sono osservate 2 basi differenti, ma non abbiamo nessun sito polimorfico per 3 o 4 basi nucleotidiche. In casi come questo, con pochi individui e con un numero relativamente piccolo di paia di basi analizzate, potrebbe essere utile visualizzare il risultato dell'allineamento. Lo facciamo con uno dei comandi di R base.

```
image(agc_mtG_ali)
```

Il grafico ci dice anche che ci sono un certo numero di basi mancanti (N). Per valutare se questo possa rappresentare un problema per le

² È necessario che l'eseguibile di `mafft` sia disponibile nel percorso del vostro personal computer, altrimenti il programma non potrà funzionare.

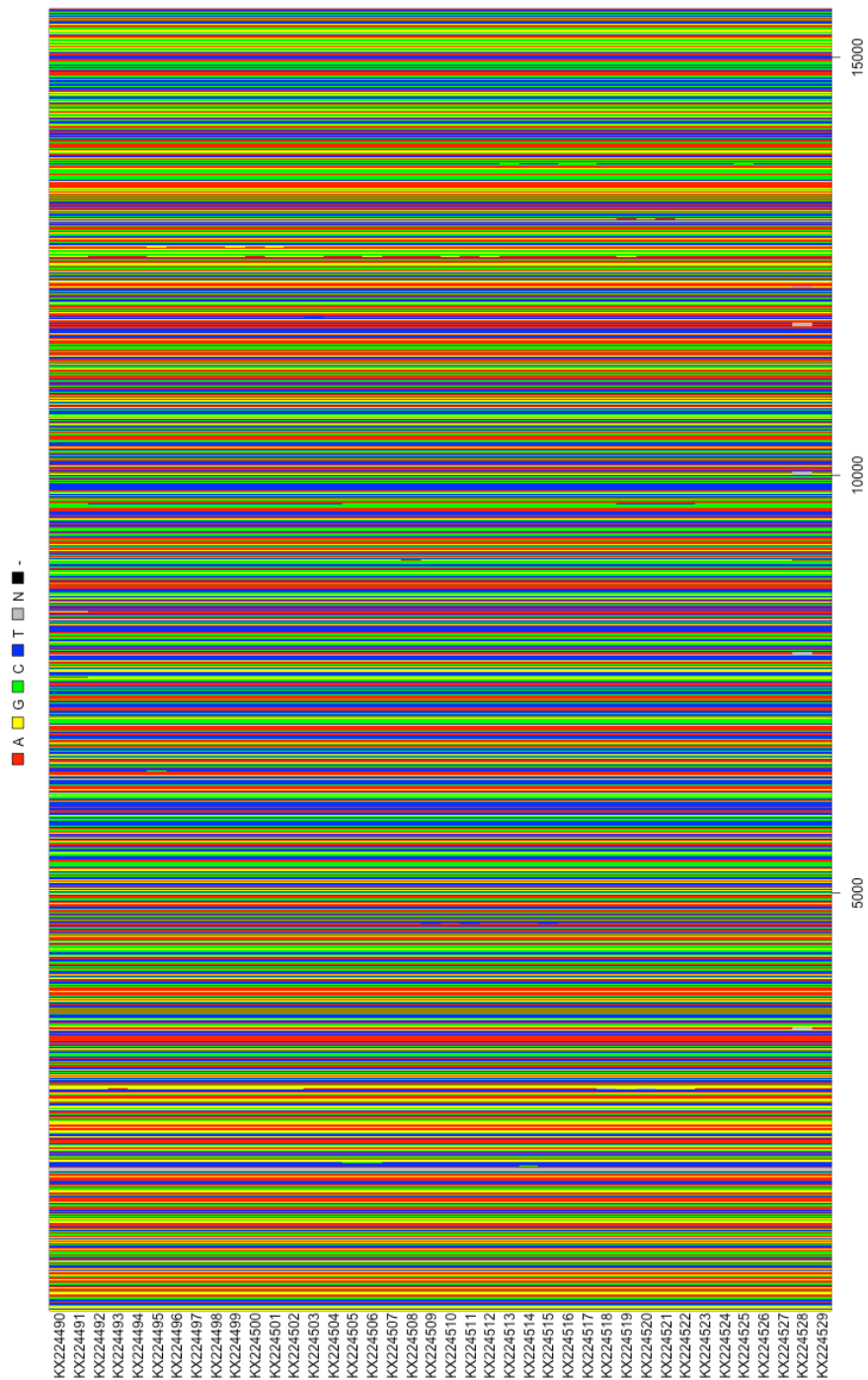


Figura 93 - Allineamento di 40 genomi mitocondriali di *Catopuma temminckii*.

analisi successive procediamo al calcolo delle varie basi nelle nostre sequenze.

```
base.freq(agc_mtG_ali, all = T)
##           a           c           g           t           r           m
## 0.327210884 0.257232704 0.138315043 0.273196637 0.000000000
0.000000000
##           w           s           k           y           v           h
## 0.000000000 0.000000000 0.000000000 0.000000000 0.000000000
0.000000000
##           d           b           n           -           ?
## 0.000000000 0.000000000 0.004044731 0.000000000 0.000000000
```

Lo 0.4% di basi mancanti non dovrebbe costituire un grosso problema. Possiamo quindi procedere all'estrazione delle sequenze uniche (aplotipi) dal nostro *dataset*.

```
acg_mtG_h <- haplotype(agc_mtG_ali)
acg_mtG_h
##
## Haplotypes extracted from: agc_mtG_ali
##
##      Number of haplotypes: 40
##      Sequence length: 15582
##
## Haplotype labels and frequencies:
##
##      I      II      III      IV      V      VI      VII      VIII      IX      X
##      1      1      1      1      1      1      1      1      1      1
##      XI     XII     XIII     XIV     XV     XVI     XVII     XVIII     XIX     XX
##      1      1      1      1      1      1      1      1      1      1
##      XXI    XXII    XXIII    XXIV    XXV    XXVI    XXVII    XXVIII    XXIX    XXX
##      1      1      1      1      1      1      1      1      1      1
##      XXXI   XXXII   XXXIII   XXXIV   XXXV   XXXVI   XXXVII   XXXVIII   XXXIX   XL
##      1      1      1      1      1      1      1      1      1      1
```

I risultati ci indicano che le sequenze sono tutte diverse una dall'altra, e che quindi ci sono 40 aplotipi differenti. Nella preparazione del *dataset* abbiamo constatato che ci sono 226 siti variabili. Possiamo identificare quali sono con il comando:

```
head(seg.sites(agc_mtG_ali))
## [1] 17 86 268 389 814 839
```

Un ultimo passaggio per la descrizione della variabilità nelle nostre sequenze di gatto dorato asiatico è quello di calcolare e visualizzare le distanze a coppie tra i vari aplotipi.

```
hist(dist.dna(agc_mtG_ali, "raw"),
     main = "Istogramma distanze a coppie tra\naplotipi",
     xlab = "Distanza (grezza)", ylab = "Frequenza")
```

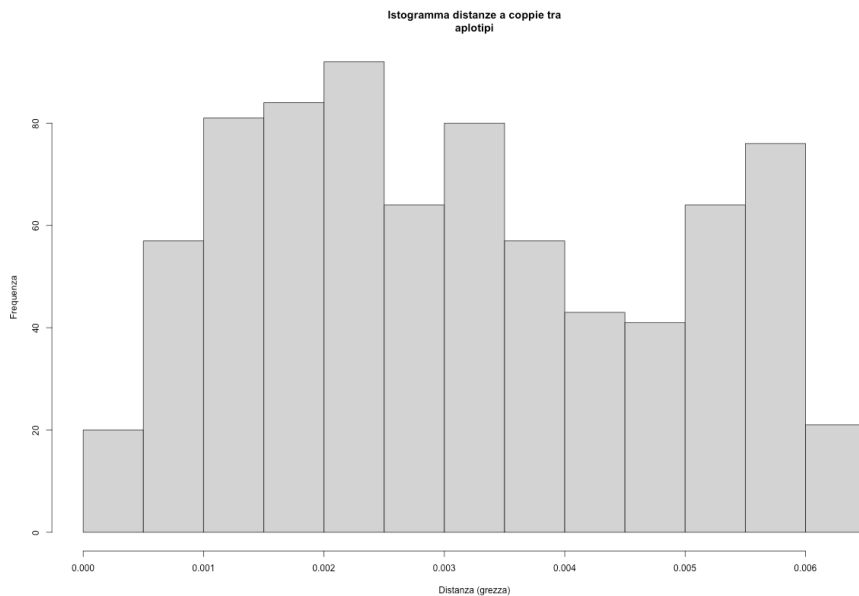


Figura 94 - Distribuzione delle distanze a coppie tra gli aplotipi come calcolate da un algoritmo che non prevede l'utilizzo di modelli evolutivi per le sequenze stesse.

Alberi filogenetici



Figura 95 - Un ipotetico albero filogenetico.

La ricostruzione delle relazioni evolutive tra i diversi organismi e le diverse specie è sempre stato uno dei temi più affascinanti della biologia in generale, e della biologia evoluzionistica in particolare. Ci sono diversi approcci alla ricostruzione dei rapporti evolutivi tra gli organismi, ma tutti i metodi condividono gli stessi principi. In particolare, tutti i metodi devono avere un criterio per la valutazione di una certa topologia della ricostruzione evolutiva rispetto ad un'altra. Quindi, ogni metodo deve avere una formula per il calcolo della lunghezza dei rami dell'albero, ogni metodo deve avere un algoritmo per l'esplorazione dello spazio dell'albero.

Utilizziamo il *dataset* degli uccelli canori del genere *Sylvia* per ricostruire i rapporti filogenetici tra i membri di questo gruppo. Le sequenze di DNA in esame hanno lunghezza diversa ed è quindi necessario effettuare un allineamento³. Per effettuare l'allineamento possiamo richiamare il software **mafft** utilizzando la funzione del pacchetto **ips**.

³ Lo *step* dell'allineamento è in realtà un passaggio obbligato, anche nel caso di sequenze della stessa lunghezza.

```
sw_seq_ali <- ips::mafft(sw_seq)
sw_seq_ali
## 25 DNA sequences in binary format stored in a matrix.
##
## All sequences of same length: 1143
##
## Labels:
## Z73494
## AJ534526
## AJ534527
## AJ534528
## AJ534529
## AJ534530
## ...
##
## Base composition:
##      a      c      g      t
## 0.270 0.354 0.131 0.245
## (Total: 28.58 kb)
```

Anche qui possiamo procedere con la visualizzazione dell'allineamento.

```
image(sw_seq_ali)
```

Una volta completato e verificato l'allineamento è possibile procedere con la ricostruzione filogenetica. I metodi per la stima della filogenesi sono molteplici. In quest'esempio ci soffermeremo su un approccio filogenetico basato sulla massima verosimiglianza (*maximum-likelihood* o *ML*). Il principio è fondamentalmente semplice. Immaginiamo di aver misurato un carattere in 3 specie differenti A, B e C. Con un modello statistico che descrive l'evoluzione di questo carattere e una filogenesi è possibile calcolare qual è la probabilità che quel carattere abbia uno specifico valore a partire dall'albero stesso.

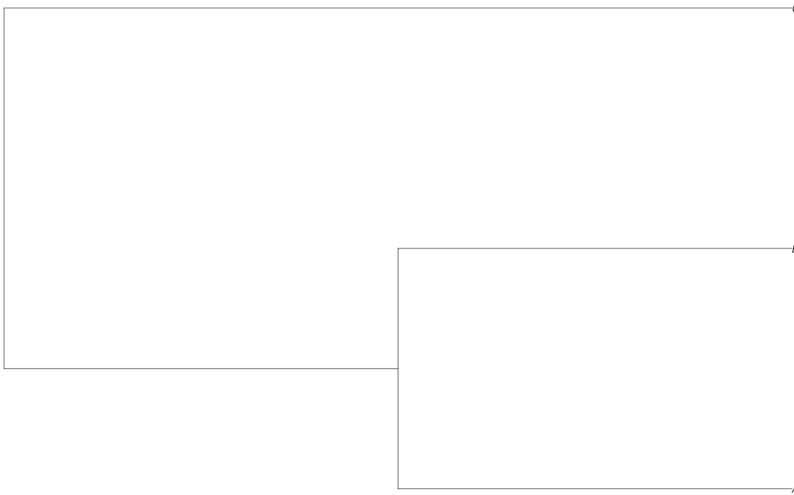


Figura 97 - Esempio di filogenesi.

Il modello statistico utilizzato per descrivere il processo evolutivo del carattere considerato assume quindi una grande importanza. Quando si parla di ricostruzioni filogenetiche basate su sequenze di DNA è importante scegliere il modello di sostituzione nucleotidica che meglio descrive i nostri dati. A questo proposito, ci viene in aiuto il secondo dei software che abbiamo installato all'inizio del capitolo: *PhyML*. Lo scopo di questo software è quello di valutare una serie di modelli di sostituzione nucleotidica alternativi (28 in tutto) e valutare quale riesce a spiegare meglio la variabilità delle sequenze in analisi.

Come accennato in precedenza, è possibile accedere all'eseguibile di *PhyML* senza lasciare il terminale di *R*. Prima di poter far questo, però, è necessario creare un *file* di testo contenente l'allineamento delle nostre sequenze. Le due funzioni che ci servono sono `write.dna()` e `phymltest()`, ed appartengono sempre al pacchetto `ape`.

```
write.dna(sw_seq_ali, "sylvia.txt")
sw_phyl <- phymltest("sylvia.txt",
  execname = "~/Desktop/Libro_MIR_Final/PhyML-3.1_macOS-MountainLion",
  append = F)
```

L'*output* della funzione è un data frame contenente i valori di verosimiglianza e dei vari modelli e l'*Akaike Information Criterion* (AIC), utile per selezionare quale dei vari modelli fa al caso nostro. È possibile utilizzare la funzione `plot` per visualizzare in maniera semplice ed intuitiva i risultati del test.

```
plot(sw_phyl, col = "black", main = "AIC")
```

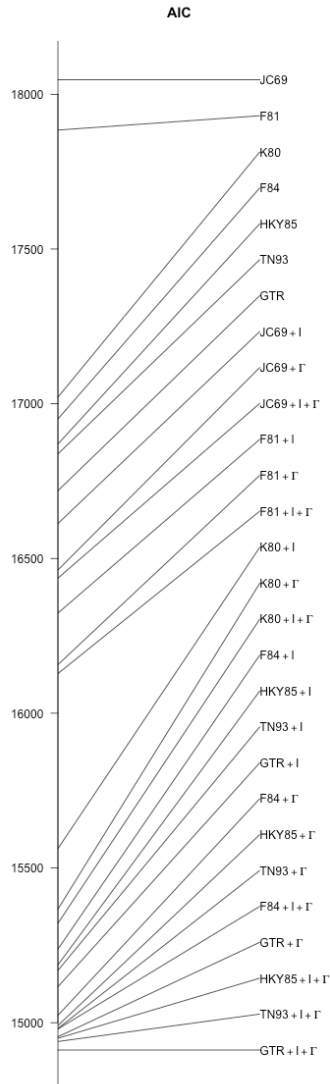


Figura 98 - Risultato di `phymtest`. Di tutti i modelli evolutivi testati, quello con il valore AIC più basso, e quindi quello che meglio aderisce alle sequenze di citocromo b nel genere *Sylvia*, è il GTR+I+Γ.

Una volta identificato il modello migliore possiamo finalmente procedere alla visualizzazione della nostra ricostruzione filogenetica. Uno degli *output* della funzione `phym1test` è un *file* di testo, che nel nostro caso si chiamerà `sylvia.txt_phym1_tree`. In questo *file* di testo sono contenuti gli alberi filogenetici, in formato Newick, costruiti seguendo i 28 modelli evolutivi testati. La prima cosa da fare è quindi quella di caricare il *file* in R per poi selezionare, basandosi sui risultati dell'analisi precedente, l'ultimo degli alberi, ovvero il ventottesimo dal *file*. A questo punto possiamo andare a modificare i nomi delle “foglie” del nostro albero per fare in modo che riflettano le specie di uccelli piuttosto che i codici di GenBank. La specie *Chamaea fasciata* è usata come *outgroup*.

```
TR <- read.tree("sylvia.txt_phym1_tree.txt")
tree_m1 <- TR[[28]]
tree_m1$tip.label <- taxaname[tree_m1$tip.label]
tree_m1 <- ape::root(tree_m1, "Chamaea fasciata")
plot(tree_m1, no.margin = T)
ape::add.scale.bar(length = 0.01,)
```

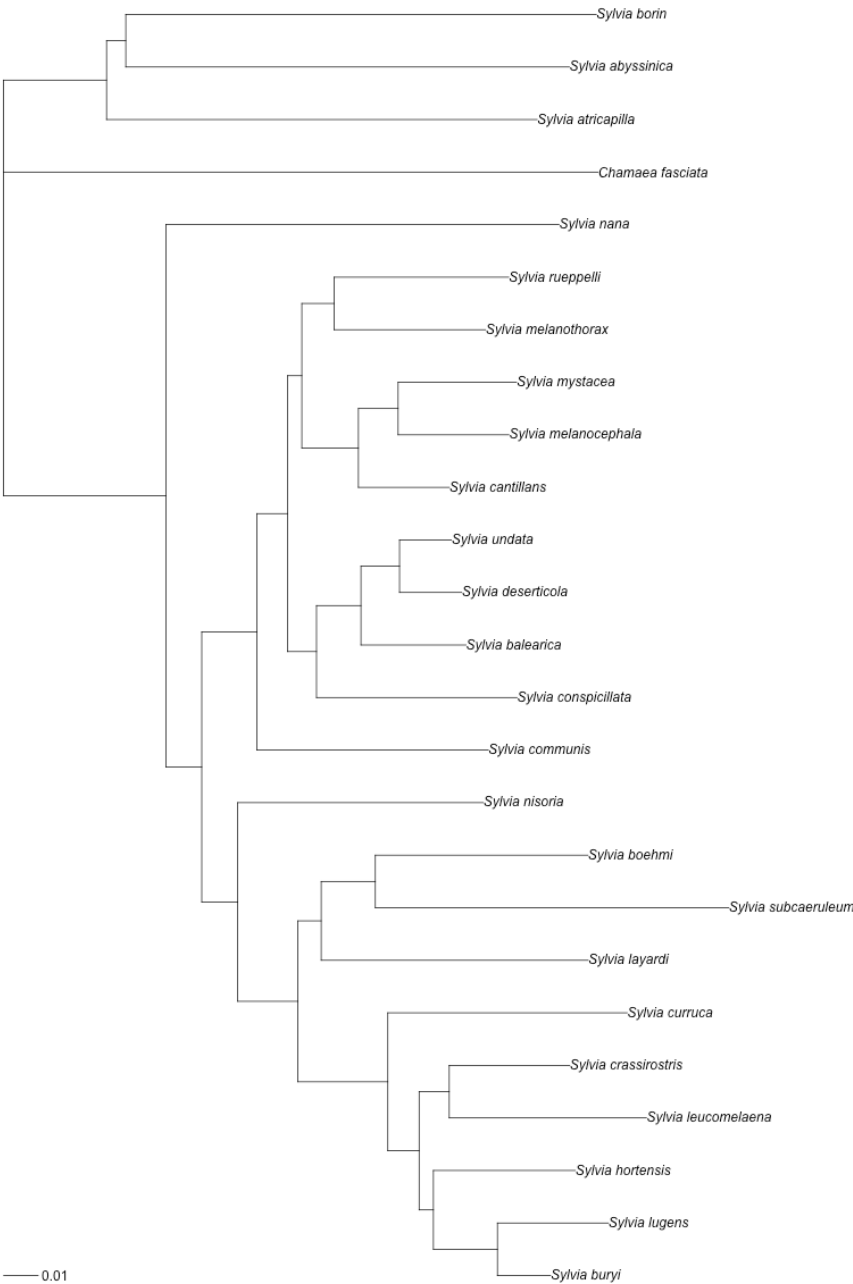


Figura 99 - Albero filogenetico, basato sulla massima verosimiglianza, a partire da 25 sequenze di citocromo b in individui del genere *Sylvia*. Il modello di sostituzione nucleotidica selezionato è il GTR+I+ Γ .

5



(Cenni di) Analisi di serie storiche in R

Ah, il tempo! Questa quarta dimensione spesso così ostica per le nostre ricerche! In effetti, quando parliamo di tempo, introduciamo un elemento speciale nelle nostre analisi, poiché legare tra loro in una sequenza cronologica un certo numero di osservazioni apre una miriade di scenari e di domande:

- esiste una (o anche più di una) dinamica ricorrente che si ripete secondo schemi riconoscibili?
- è presente una tendenza di lungo periodo, che magari non riusciamo a scorgere in un intervallo temporale ristretto?
- come facciamo a prevedere il valore di una certa quantità o la condizione di un certo sistema nel futuro?

Non è possibile affrontare questi aspetti, e non avrebbe senso parlarne nel contesto di un ambiente come R , senza fare un minimo di chiarezza, anche senza la pretesa di presentarne una trattazione formale.

In questo capitolo considereremo un insieme di modelli per serie storiche, accomunati dal fatto che possono essere stimati utiliz-

zando l'approccio dei minimi quadrati (*Ordinary Least Squares, OLS*). Anche se da una parte questo vincolo lascia fuori dal nostro perimetro classi molto importanti di modelli, al contempo ci permette di dare una trattazione più intuitiva del tema. A livello applicato, anche se esistono molte funzioni R , e molti pacchetti, dedicate alla stima dei modelli per serie storiche, daremo precedenza alla "solita" funzione `lm`, generalmente utilizzata per la stima *OLS* dei modelli di regressione. Adotteremo anche gli assunti implicitamente utilizzati da questa funzione, ed in particolare la normalità e l'omoschedasticità degli errori. Partiamo dalla seguente definizione informale: una serie storica

$$\{y_t\}, \quad t = 1, \dots, T$$

è una sequenza di osservazioni rilevata su di una singola unità statistica rispetto ad una variabile in T distinte occasioni. In generale, l'obiettivo dell'analisi sarà ottenere una *previsione* della variabile di interesse in un generico periodo futuro $t+h$. Definiremo quindi h come *l'orizzonte di previsione*. Molto spesso saremo interessati ad ottenere una previsione *ad un passo*, e quindi per il valore di y_{t+1} . Tre esempi ci permettono di fare conoscenza con diversi tipi di serie storiche che ci appaiono a prima vista molto diverse. Nel primo caso, analizziamo l'andamento della concentrazione di CO_2 nell'atmosfera negli ultimi 70 anni usando le misurazioni fatte dall'*Earth System Research Laboratory* del *Global Monitoring Division* (NOAA) del governo USA, presso Mauna Loa, alle Hawaii.

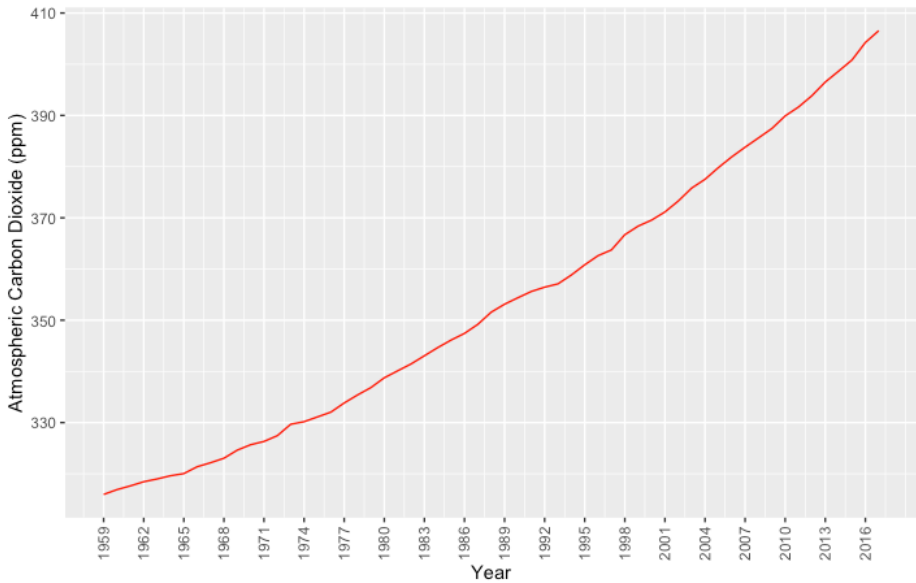


Figura 100 - Andamento della concentrazione di anidride carbonica nell'atmosfera in parti per milione al netto del vapore acqueo.

Il grafico non ha bisogno di commenti: esiste una evidente e monotona tendenza di lungo periodo (un *trend*) verso un aumento della concentrazione di anidride carbonica nell'atmosfera. Il secondo esempio è rappresentato dall'andamento della produttività primaria netta, dal 2013 al 2021, nelle acque dei 3 principali bacini marini che circondano la nostra penisola: il Mare Adriatico, il Mare Tirreno e lo Stretto di Sicilia. In questo caso abbiamo usato le stime ottenute usando il *Vertically Generalized Production Model* e disponibili online presso l'*Ocean Productivity Website*.

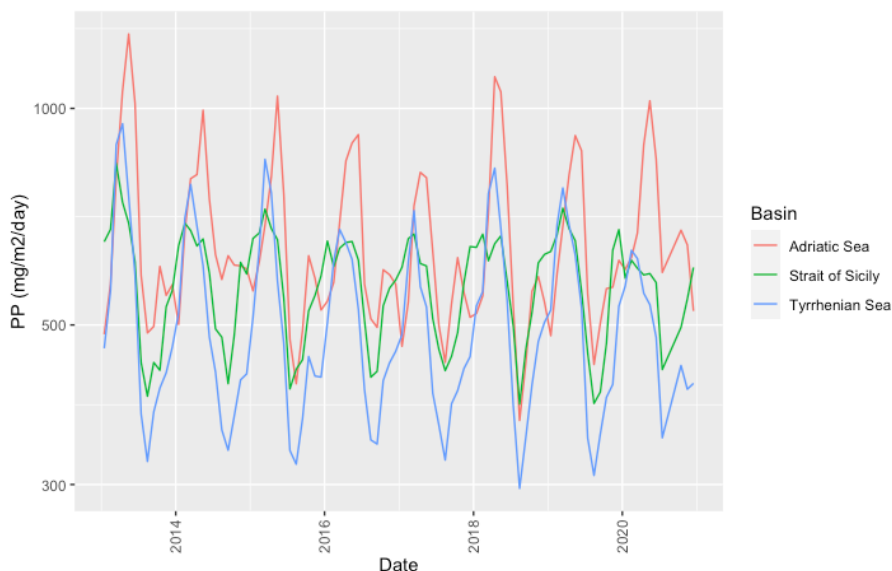


Figura 101 - Andamento della produttività primaria in milligrammi di carbonio per metro quadro per giorno.

Queste serie presentano diverse caratteristiche. Non è immediato cogliere che l'andamento della serie relativa all'Adriatico presenta un ritardo rispetto a quelli degli altri due bacini. È, invece, del tutto evidente che tutte le serie presentano una fluttuazione stagionale, con valori molto più alti in primavera/estate e valori più bassi in inverno.

Il terzo esempio è la serie storica, disponibile su *datahub.io*, del prezzo nominale giornaliero del gas naturale in dollari. In questo caso, non c'è né una tendenza di lungo periodo né un comportamento stagionale. Piuttosto, questa serie si caratterizza per una variabilità che cambia nel tempo. Periodi di alta volatilità si alternano a periodi di volatilità molto più contenuta.

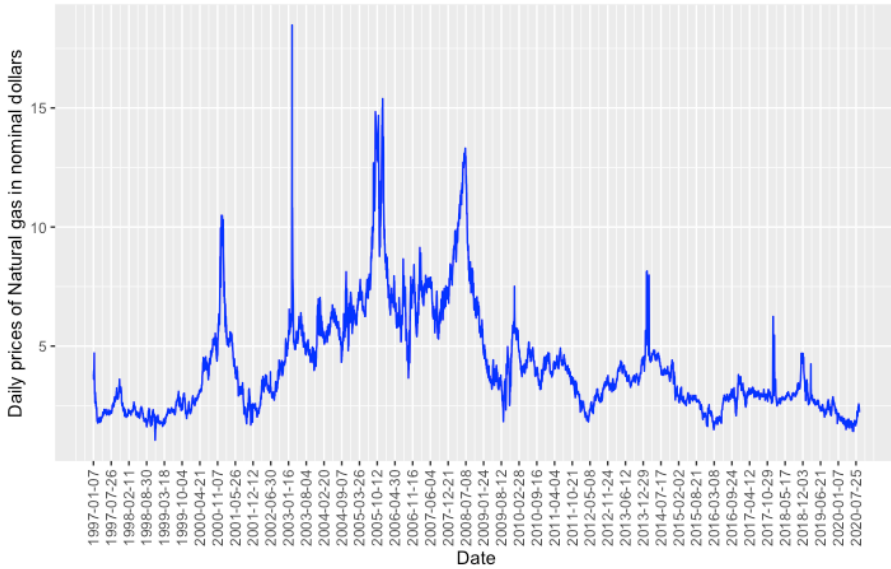


Figura 102 - Prezzo nominale giornaliero del gas naturale in dollari.

Questi tre semplici esempi ci suggeriscono che, quando abbiamo a che fare con le serie storiche, possiamo trovarci nei pasticci perché una singola serie storica può mostrare una o più caratteristiche. Qualunque sia il nostro scopo mentre ci accingiamo ad analizzare una serie storica, diventa cruciale identificare e separare queste componenti.



Figura 103 - Come spesso capita nel passaparola, in cui l'informazione iniziale viene distorta, così nelle serie temporali può capitare che un fenomeno prenda direzioni imprevedibili.

Il dataset

Per sviluppare questo discorso prenderemo in considerazione un esempio molto attuale e istruttivo: un *dataset* contenente due serie storiche mensili relative alla *temperatura marina superficiale* e alla *concentrazione di clorofilla* nei primi metri della colonna d'acqua nel mare Adriatico, ed in particolare la GSA 17 secondo la suddivisione FAO. Il periodo di osservazione parte da gennaio 1999 e arriva a dicembre 2020, per un totale di 264 osservazioni. I fenomeni rilevati sono importanti per diverse ragioni:

- per gli oceanografi, meteorologi e climatologi, essi rappresentano uno dei segnali dello scambio di energia tra l'oceano e l'atmosfera;
- per i biologi marini, essi sono importanti per monitorare la produzione primaria e quindi capire quanta energia è disponibile per gli ecosistemi marini;
- per i pescatori, rilevanti variazioni nella temperatura possono indicare zone particolarmente produttive per la pesca.

In genere, è consigliabile definire un *periodo di stima* ed un *periodo di validazione*. Nel nostro caso:

- le osservazioni nel periodo gennaio 1999 fino a ottobre 2020 saranno utilizzate per identificare un modello, stimarlo e ottenere una previsione;
- una volta ottenute le previsioni, queste saranno messe a confronto con le osservazioni effettivamente rilevate a novembre e dicembre 2020. Questa operazione ci permetterà di commentare le capacità predittive del modello stimato.

Le due parti del *dataset* si trovano nei due *file data.csv* e *validation.csv*.

```
# Importazione dati
url.train =
  'https://raw.githubusercontent.com/tommaso-russo/LibroR2/main/
  TimeSeries/data.csv'
train = read.csv(url.train)
```

Diamo uno sguardo alle prime righe del *dataset*

```
head(train)
##   Time Year Month      Temp      Chl
## 1     1 1999     1 15.66876 0.08680951
## 2     2 1999     2 14.85220 0.10453716
## 3     3 1999     3 14.98809 0.11338469
## 4     4 1999     4 15.97911 0.12281536
## 5     5 1999     5 19.21883 0.15615801
## 6     6 1999     6 22.68955 0.14836864
```

La prima colonna è un identificativo progressivo delle osservazioni. Oltre all'identificativo, per ogni osservazione abbiamo mese ed anno di riferimento, oltre al valore della temperatura (`Temp`) e del livello di clorofilla (`chl`). Anche se non è necessario, possiamo effettuare una serie di operazioni sui dati, in modo da semplificare la loro gestione nel resto del capitolo.

```
# Dati
y = train$Temp          # Temperatura
z = train$Chl           # Clorofilla
month = as.factor(train$Month) # Mese
year = train$Year       # Anno
```

In particolare, il vettore `month` è stato importato sotto forma di `factor`. Più avanti ne sarà chiarito il motivo.

Procediamo con qualche statistica descrittiva:

```
# Statistiche descrittive
Ti = nrow(train)          # Numero di osservazioni
nYears = length(unique(year)) # Numero di anni
```

Per ogni serie storica:

- abbiamo un numero di osservazioni `Ti` pari a 262 osservazioni mensili, che abbiamo salvato in memoria con il simbolo `Ti` (non è consigliabile usare il simbolo `T`, che in R è il simbolo riservato al valore logico `TRUE`);
- le osservazioni sono relative ad un arco temporale che copre 22 anni diversi (`nYears`).

Il grafico rivela intuitivamente la natura della serie relativa all'andamento della clorofilla:

```
# Grafici
plot.ts(z)
abline(v=(0:nYears)*12, lty=2, col='lightgray')
```

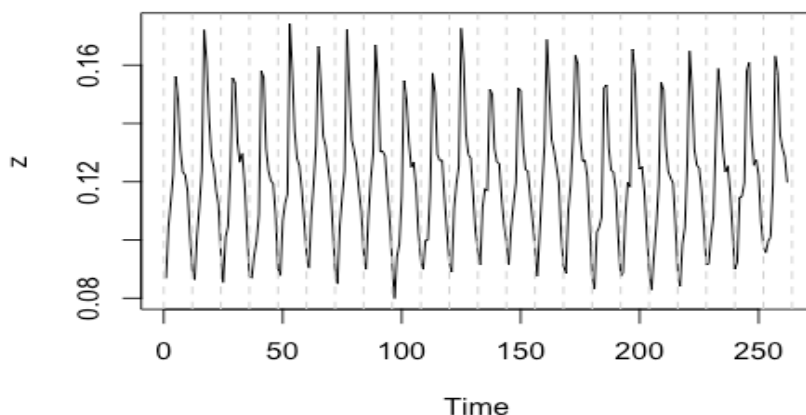


Figura 104 - Andamento della clorofilla nel periodo considerato.

Le linee verticali sono poste in corrispondenza con il dicembre di ogni anno. Si nota che la serie della clorofilla presenta un andamento fortemente stagionale, con un picco negativo in corrispondenza del periodo invernale, ed il minimo che molto spesso si verifica in gennaio. Al contrario, c'è un picco positivo nel periodo primaverile, con il massimo che generalmente si verifica in maggio. Si può inoltre notare che, anche se l'ampiezza delle oscillazioni varia negli anni, il *pattern* della serie è molto simile per molti degli anni presi in considerazione.

L'andamento della temperatura è ancora più regolare:

```
plot.ts(y)
abline(v=(0:nYears)*12, lty=2, col='lightgray')
```

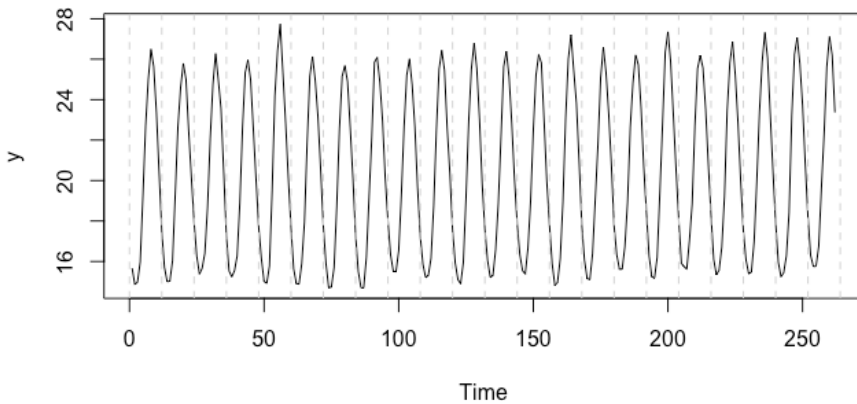


Figura 105 - Andamento della temperatura nel periodo considerato.

Ogni anno abbiamo un andamento molto “liscio” e con oscillazioni di ampiezza molto simile su tutto l’arco temporale considerato.

Obiettivi della ricerca

Immaginiamo adesso di essere alla data dell’ultima osservazione del periodo di stima (ottobre 2020), e che qualcuno ci chieda di esaminare questo insieme di dati per rispondere, dopo un’attenta analisi, ai seguenti quesiti:

- qual è la migliore scommessa per il valore della temperatura nel mese di novembre 2020?
- ... e di dicembre 2020?
- la temperatura sta salendo nel tempo?
- è ragionevole utilizzare la temperatura come un predittore del livello della clorofilla?

Disclaimer: i modelli che verranno presentati in questo capitolo saranno ovviamente troppo semplici per dare una risposta *affidabile* a queste domande. Lo scopo di questo capitolo è solamente quello di fornire una introduzione ai modelli di serie storiche.

I grafici riportati sopra suggeriscono che, per rispondere a queste domande, potremmo stimare un *modello di regressione* in cui il tempo assume il ruolo di regressore. Dovremmo però considerare una importante caratteristica delle serie storiche: generalmente, in questo tipo

di dati, esiste una struttura di dipendenza tra osservazioni successive. Il modello di regressione “standard”, che generalmente utilizziamo per i dati *cross section*, non riesce ad incorporare adeguatamente questa relazione che esiste tra le osservazioni. Noi partiremo comunque da un modello di regressione *naive*, ignorando questa caratteristica. Andremo poi, progressivamente, ad evidenziare le limitazioni della specificazione, e a modificare tale specificazione per permettere al modello di superare queste limitazioni. Alla fine, sarà necessario lavorare sulla specificazione del modello per gestire correttamente la struttura di dipendenza.

Stimatore dei minimi quadrati

In questo paragrafo si riassumono le basi teoriche minime che ci saranno utili per il resto del capitolo. Tutti i modelli che vedremo verranno stimati con il metodo dei minimi quadrati. Regrediremo la nostra variabile risposta y_t su una o più covariate x_{tj} :

$$y_t = \beta_0 + \beta_1 x_{t1} + \dots + \beta_k x_{tk} + \varepsilon_t$$

Indicando con X la *matrice disegno*, cioè quella matrice che raccoglie un regressore costante insieme a tutte le covariate

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1k} \\ 1 & x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{t1} & x_{t2} & \cdots & x_{tk} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{T1} & x_{T2} & \cdots & x_{Tk} \end{pmatrix}$$

Lo stimatore dei minimi quadrati per l'intero vettore di parametri risulta

$$\hat{\beta} = (X'X)^{-1} X'y$$

Se gli errori:

- hanno valore atteso nullo,
- sono omoschedastici,

- sono *indipendenti e identicamente distribuiti*, oppure, più brevemente, se

$$\varepsilon_t \stackrel{\text{iid}}{\sim} D(0, \sigma^2)$$

allora lo stimatore *OLS* è *corretto e consistente*.

Assumendo anche che gli errori siano anche *normalmente distribuiti*, allora lo stimatore *OLS* è anche il più efficiente tra gli stimatori lineari e non distorti.

Possiamo ottenere il vettore dei *valori predetti*

$$\hat{y} = X \hat{\beta}$$

ed il vettore di *residui*

$$e = y - \hat{y}$$

Avendo calcolato i residui, possiamo stimare la varianza della regressione

$$\hat{\sigma}^2 = \frac{1}{T-k-1} e'e$$

Per ottenere intervalli di confidenza e per effettuare verifiche di ipotesi, sarà anche necessario ottenere gli errori standard delle stime. Partendo dalla matrice di covarianza stimata

$$\widehat{\text{Var}}(\hat{\beta}) = \hat{\sigma}^2 (X'X)^{-1}$$

è possibile ottenere gli errori standard delle stime come la radice quadrata degli elementi sulla diagonale di questa matrice

$$ES(\hat{\beta}_j) = \sqrt{[\widehat{\text{Var}}(\hat{\beta})]_{jj}}$$

Previsioni

La stima di un modello ci permette di spiegare il meccanismo di funzionamento del fenomeno di interesse. Ad esempio, tra gli obiettivi di ricerca, ci chiediamo se la temperatura possa essere utile per prevedere l'andamento del livello della clorofilla. Cercheremo anche di capire se, nei prossimi anni, la temperatura sarà stabile ovvero tenderà a scendere o a salire. Tuttavia, il più delle volte, l'analisi delle serie storiche mira ad ottenere una previsione che ci permetta di poter capire cosa ci riserva il futuro. Ad esempio, otterremo una previsione per i "prossimi mesi" (novembre e dicembre 2020).

Previsioni puntuali

Ipotizziamo di aver osservato la serie storica fino al tempo τ , e di essere interessati alla previsione ad h passi. Vogliamo, cioè, la previsione per l'osservazione relativa al tempo $\tau + h$, che presenta il profilo $\mathbf{x}_{\tau+h}$

$$\mathbf{x}'_{\tau+h} = (1, x_{(\tau+h),1}, x_{(\tau+h),2}, \dots, x_{(\tau+h),k})$$

la previsione risulta

$$\hat{y}_{\tau+h|\tau} = \mathbf{x}'_{\tau+h} \hat{\boldsymbol{\beta}} = \hat{\beta}_0 + \hat{\beta}_1 x_{\tau+h,1} + \dots + \hat{\beta}_k x_{\tau+h,k}$$

Intervalli di previsione

Possiamo anche ottenere un intervallo che contenga, ad un dato livello di confidenza, il valore che y_t assumerà al tempo $\tau+h$.

Se è possibile utilizzare il *Teorema del Limite Centrale*, l'intervallo di previsione per $y_{\tau+h}$ è dato da

$$[\hat{y}_{\tau+h|\tau} - z_{1-\alpha/2} \hat{\sigma}_h, \hat{y}_{\tau+h|\tau} + z_{1-\alpha/2} \hat{\sigma}_h]$$

dove il simbolo z_p indica il quantile di una Normale standard di ordine p . In generale, è comune impostare il livello di confidenza al 95%. A questo livello di confidenza, si ha $z_{0.975} = 1.96$.

Possiamo stimare

$$\hat{\sigma}_h = \sqrt{\hat{\sigma}^2 (1 + \mathbf{X}'_{T+h} (\mathbf{X}'\mathbf{X})^{-1} \mathbf{X}_{t+h})}$$

Verifica di ipotesi

Quanto descritto sopra è sufficiente anche per sottoporre a verifica una ipotesi rispetto ad un generico parametro β_j .

Se, ad esempio, vogliamo sottoporre a verifica l'affermazione che il parametro β_j sia pari ad un determinato valore $\beta_{j,0}$, contro l'ipotesi alternativa che invece tale parametro abbia un valore diverso, possiamo formalizzare il seguente sistema di ipotesi

$$\begin{cases} H_0: \beta_j = \beta_{j,0} \\ H_1: \beta_j \neq \beta_{j,0} \end{cases}$$

Per condurre questa verifica di ipotesi, possiamo utilizzare la statistica test

$$t = \frac{\hat{\beta}_j - \beta_{j,0}}{ES(\hat{\beta}_j)}$$

Sotto l'ipotesi nulla, usando ancora il *Teorema del Limite Centrale*, la statistica t si distribuisce come una Normale standard.

Una volta calcolato il valore osservato della statistica test t^{obs} , possiamo calcolare il p -valore come

$$p = 2\Phi(-|t^{obs}|)$$

Un valore basso di p (ad esempio, $p < 0.05$) ci porterà al **rifiuto** dell'ipotesi nulla.

Con piccole modifiche, possiamo condurre test con ipotesi alternativa unilaterale, dove l'ipotesi alternativa è

$$H_1 : \beta_j > \beta_{j,0} \quad \text{o} \quad H_1 : \beta_j < \beta_{j,0}$$

Modelli con *trend*

Il primo approccio che possiamo usare è quello proprio di una regressione lineare. In sostanza, immaginiamo che, al netto delle evidenti fluttuazioni, esista una tendenza di lungo periodo, o *trend*, per la temperatura e/o il livello di clorofilla. A parte la sua ovvia rigidità, questo approccio è molto semplice e ci permette di dare una risposta a (quasi) tutte le domande che ci siamo posti all'inizio. Partiamo da una definizione informale del concetto di *trend*.

Per “trend” intenderemo una tendenza di lungo periodo eventualmente presente nell'evoluzione della serie storica.

I *trend* possono essere *deterministici* o *stocastici*. Qui considereremo modelli contenenti un *trend* deterministico della forma

$$y_t = T_t + \varepsilon_t, \quad \varepsilon_t \stackrel{\text{iid}}{\sim} D(\theta, \sigma^2)$$

Modello con *trend* lineare

Se il *trend* è lineare

$$T_t = \beta_0 + \beta_1 t$$

il modello può essere riscritto come

$$y_t = \beta_0 + \beta_1 t + \varepsilon_t$$

e può essere stimato tramite minimi quadrati ordinari.

Si noti come il vettore

$$t = \{1, 2, \dots, T\}$$

che contiene l'indice delle osservazioni, qui giochi il ruolo di regressore.

Abbiamo già a disposizione l'indice del tempo all'interno del *dataset*, nella variabile `Time`.

```
tt = train$Time
```

Per questo oggetto, utilizzeremo il nome `tt`, in quanto il nome `t` è riservato per la funzione di trasposizione. Possiamo ora stimare il modello di regressione

```
# Modello con trend lineare
```

```
fitT = lm(y ~ tt)
```

Dato che stimeremo diversi modelli, per distinguere i nomi degli oggetti che conterranno le diverse stime, useremo suffissi diversi. Ad esempio, il suffisso `T` nel nome dell'oggetto `fitT` indica che il modello contiene un *trend*. La funzione `summary` ci permette di avere i principali risultati della stima.

```
summary(fitT)
```

```
##
```

```
## Call:
```

```
## lm(formula = y ~ tt)
```

```
##
```

```
## Residuals:
```

```
##      Min       1Q   Median       3Q      Max
```

```
## -5.5610 -4.1390 -0.5099  4.2999  7.7580
```

```
##
```

```
## Coefficients:
```

```
##              Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) 19.747336   0.509187  38.782  <2e-16 ***
```

```
## tt           0.003992   0.003357   1.189    0.235
```

```
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
```

```
## Residual standard error: 4.109 on 260 degrees of freedom
## Multiple R-squared:  0.005412,    Adjusted R-squared:  0.001587 
## F-statistic: 1.415 on 1 and 260 DF,  p-value: 0.2354
```

Dalla tabella di regressione in *output*, il modello stimato risulta

$$\hat{y}_T = 19.747 + 0.004 \cdot t, \quad \hat{\sigma}^2 = 4.109^2$$

Questa equazione ci dice che il valore medio della temperatura cresce di 4 millesimi di grado per ogni mese che passa. Anche se non sembra molto, la variazione che si avrebbe su un arco di 10 o 20 anni sarebbe rilevante. Anche se non stiamo parlando di *global warming*, ma più semplicemente della temperatura superficiale del mare Adriatico, potremmo essere preoccupati di questa conclusione.

Possiamo però sospettare che la temperatura sia in realtà stabile, e che la stima puntuale del coefficiente associato al tempo sia positiva solamente per variabilità campionaria. In altri termini, sospettiamo che, se avessimo stimato lo stesso modello su un altro periodo di tempo, avremmo potuto ottenere valori più piccoli oppure, addirittura, negativi.

Possiamo provare a disegnare il modello stimato

```
# Retta di regressione
plot.ts(y)
abline(fitT$coeff, col = 2)
```

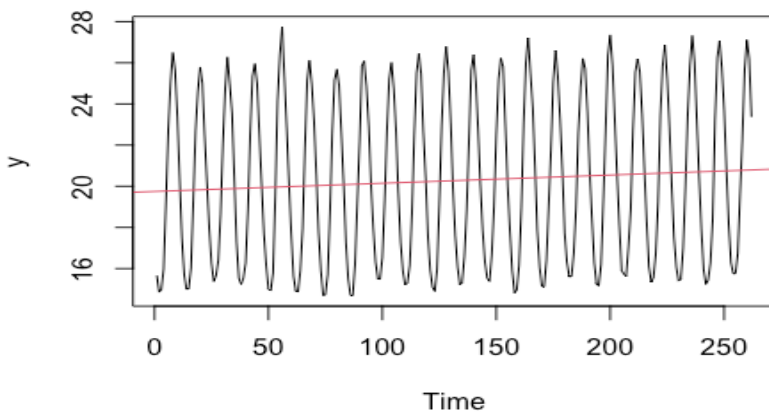


Figura 106 - Andamento della temperatura con retta di regressione.

ma è complesso valutare “ad occhio” se nei dati esiste effettivamente un *trend*, oppure se il valore della stima puntuale della pendenza dipende solamente dal particolare periodo in cui la serie storica è stata raccolta.

Possiamo però fare riferimento ai risultati, presenti nella tabella di regressione, riguardanti la significatività del coefficiente. Dato che il p-valore è piuttosto elevato (23.5%), *non* possiamo confutare l'ipotesi che la temperatura *non* cresca nel tempo. Questa conclusione ci fornisce una risposta al terzo *obiettivo della ricerca*: *se crediamo che la specificazione di questo modello sia corretta*, la temperatura non sta aumentando nel tempo.



Figura 107 - Una covariata importante si comporta come un grosso cane che trascina il padrone (la variabile dipendente).

Previsioni puntuali

Per rispondere, invece, alle prime due domande, ipotizziamo di essere al tempo τ (ottobre 2020), e di voler ottenere una previsione per il tempo (novembre 2020) $\tau + 1$ o, più in generale, per il tempo $\tau + h$.

Dato il modello

$$y_t = \beta_0 + \beta_1 t + \varepsilon_t.$$

utilizzando le stime dei parametri di regressione precedentemente ottenuti, e grazie alle assunzioni su ε_t , possiamo ottenere una stima per $y_{\tau+h}$, avendo osservato la serie storica fino al tempo τ .

$$\hat{y}_{\tau+h|\tau} = \hat{\beta}_0 + \hat{\beta}_1 (\tau + h)$$

Il comando:

```
# Previsioni
predict(fitT, newdata=data.frame(tt=Ti+1))
##      1
## 20.79734
```

restituisce la previsione “ad un passo”

$$y_{263|262} = \underset{(0.509)}{19.747} + \underset{(0.003)}{0.0004} (262 + 1) = 20.797$$

Il secondo argomento della funzione `predict` è un `data.frame` con il valore della covariate (nel nostro caso, il tempo) dell'osservazione per la quale vogliamo una previsione. Per questo modello, basta modificare questo argomento per ottenere previsioni per modificare l'orizzonte di previsione y oppure ottenere la previsione puntuale per diversi orizzonti.

Il nostro periodo di validazione comprende i mesi di novembre e dicembre 2020, quindi per avere una previsione per entrambi gli orizzonti, possiamo scrivere

```
predict(fitT, newdata=data.frame(tt=Ti+(1:2)))
##          1          2
## 20.79734 20.80133
```

L'*output* riporta gli orizzonti temporali h considerati nella prima riga e le rispettive previsioni puntuali per y_{T+h} nella seconda riga. Ovviamente, il primo valore coincide con il risultato precedente.

Intervalli di previsione

Possiamo ottenere anche gli intervalli di previsione aggiungendo l'opzione `interval` all'interno della funzione `predict`

```
PI = predict(fitT, newdata=data.frame(tt=Ti+(1:2)),
interval="prediction")
PI
##          fit          lwr          upr
## 1 20.79734 12.64397 28.95071
## 2 20.80133 12.64725 28.95541
```

L'oggetto `PI` (*Prediction Intervals*) contiene una tabella. Ogni riga di questa tabella si riferisce ad un orizzonte di previsione:

- la colonna `fit` contiene le previsioni puntuali,
- le colonne `lwr` e `upr` contengono il limite inferiore e superiore dell'intervallo di previsione.

Le previsioni puntuali coincidono con quelle già ottenute. Le ultime due colonne riportano gli estremi degli intervalli di confidenza.

Riportiamo i risultati in un grafico:

```
plot.ts(y, xlim=c(Ti-35,Ti+2))
abline(v=(0:nYears)*12, lty=2, col='lightgray')
for(h in 1:2){
  segments(Ti+h, PI[h,2], Ti+h, PI[h,3], col=2)
  points(Ti+h, PI[h,1], pch=20, col=2)
}
```

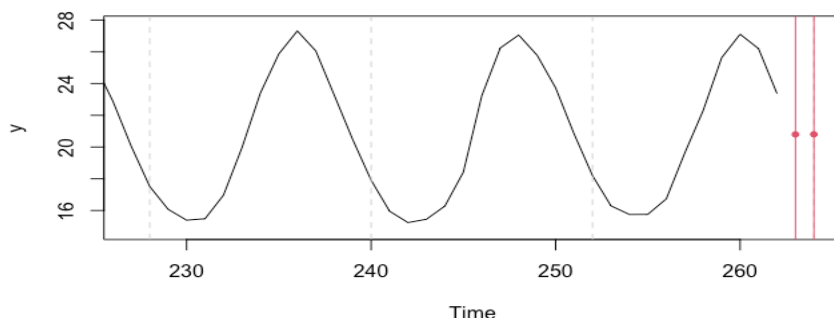


Figura 108 - Andamento osservato e due previsioni.

La figura precedente riporta i valori relativi agli ultimi tre anni della serie storica (le osservazioni $y_{(T-35):T}$), le previsioni ad uno e due passi (in rosso), e i relativi intervalli di previsione. Questi intervalli sono talmente ampi da uscire dall'area del grafico. L'ampiezza di questi intervalli, causata da un valore molto elevato dell'errore standard residuo, aumenta con l'orizzonte h .

Validazione delle previsioni

Abbiamo ottenuto le previsioni facendo finta di essere nell'ottobre 2020, e quindi non potendo conoscere il valore della serie storica per i mesi finali dell'anno. Adesso possiamo confrontare le previsioni ottenute con i valori che si sono effettivamente misurati. Il file `validation.csv` contiene i valori veri per i mesi di novembre e dicembre 2020.

```
# Dati del periodo di validazione
url.test =
'https://raw.githubusercontent.com/tommaso-russo/LibroR2/main/
TimeSeries/validation.csv'
test = read.csv(url.test)
```

Un rapido sguardo...

```
test
##   Time Year Month   Temp      Ch1
## 1  263 2020    11 20.93219 0.10551576
## 2  264 2020    12 18.35922 0.09065305
```

... mostra che il *file* ha la stessa struttura di quello usato finora (questo è sempre un bel momento per chi lavora sui dati). Quindi possiamo definire due nuove variabili:

```
yStar = test$Temp
zStar = test$Ch1
```

Andiamo a vedere dove si collocano le osservazioni di questo *data-set*. Il grafico è identico al precedente, ma sono state aggiunte le nuove osservazioni in blu.

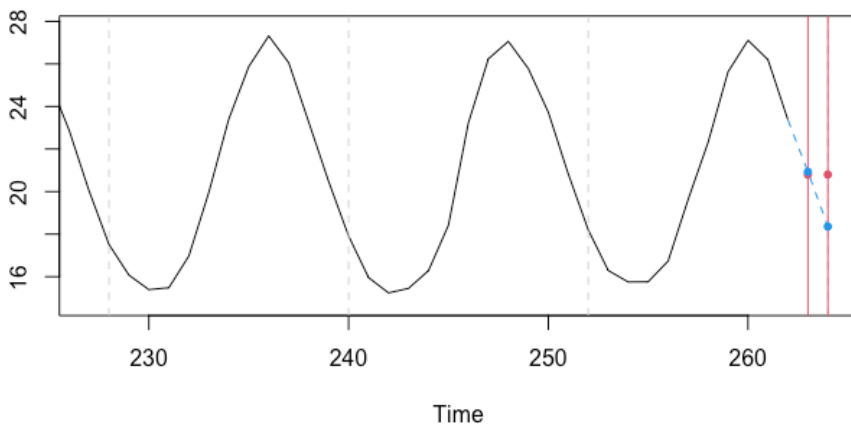


Figura 109 - Andamento osservato e due (nuove) previsioni.

La previsione per novembre ($y_{(T+1)}$) è molto vicina al valore che si è poi verificato, ma questo accade solamente per caso. La previsione per dicembre è piuttosto distante dal valore reale, e le previsioni per i primi mesi del 2021 sarebbero ancora più distanti. In generale, non possiamo dire che il modello fornisca previsioni attendibili.

Ricapitoliamo, comunque, le risposte alle domande che ci eravamo posti nella sezione *obiettivi della ricerca*:

qual è la migliore scommessa per il valore della temperatura nel mese di novembre 2020? La temperatura prevista per novembre 2020 è pari a 20.797 gradi centigradi.

... e di dicembre 2020? Si prevede una temperatura pari a 20.801 gradi centigradi.

la temperatura sta salendo nel tempo? No, la temperatura non sta salendo nel tempo.

è ragionevole utilizzare la temperatura come un predittore del livello della clorofilla? La variabile che misura il livello della clorofilla non è inclusa nel modello, quindi questo modello non può essere di aiuto a rispondere all'ultima domanda.

Diagnostiche del modello

Possiamo ritenere valide tutte queste conclusioni solamente se crediamo che la specificazione di questo modello sia corretta. Ci sono, però, indizi che ci inducono a pensare che questo modello non sia correttamente specificato, e che quindi queste conclusioni possano essere fuorvianti:

- innanzi tutto, è poco plausibile che un modello così semplice possa descrivere adeguatamente le caratteristiche di un fenomeno complesso come la temperatura. Infatti, la tabella di regressione riporta un valore molto basso dell'indice R^2 (**Multiple R-squared**, pari allo 0.5%): la frazione di variabilità spiegata dal modello è modesta, il modello non si adatta bene ai dati della nostra serie storica,
- in particolare, il grafico del modello stimato mostra che il modello non coglie le fluttuazioni infrannuali,
- di conseguenza, le previsioni non sono attendibili e i relativi intervalli sono caratterizzati da una notevole ampiezza.

Controlliamo formalmente se ci sono evidenze contro la specificazione del modello. Se gli assunti del modello sono realistici, il grafico dei residui, disegnati contro l'indice temporale:

- non deve mostrare alcun *pattern*,
- non deve mostrare una varianza che cambia nel tempo.

Il vettore dei residui può essere estratto dall'oggetto che contiene la stima del modello

```
# Residui
e = fitT$residuals
```

e rappresentato come una serie storica.

```
plot.ts(e)
abline(h=0, col = 2)
```

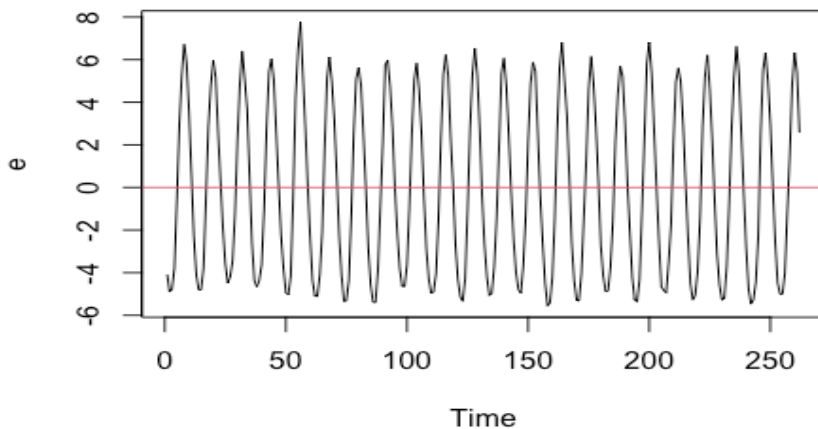


Figura 110 - Serie storica del vettore dei residui.

Questo grafico mostra un evidente *pattern*: poiché il comportamento *stagionale* della serie non è stato catturato dal modello, questo è ancora presente nei residui.

Dato che ci stiamo domandando, tra le altre cose, se la temperatura sta aumentando nel tempo, è sicuramente importante includere un regressore che possa rappresentare la presenza di un *trend* nei nostri dati. La presenza di questo regressore nel modello, però, non basta per cogliere tutte le caratteristiche presenti nei nostri dati. In particolare, è evidente che manca la possibilità di cogliere le fluttuazioni *stagionali*.

La semplicità del modello non sembra compensare la sua mancanza di flessibilità. Abbiamo altre opzioni? Ma certo!

Altri modelli di *trend*

Oltre alla specificazione lineare

***Trend* lineare**

$$T_t = \beta_0 + \beta_1 t$$

abbiamo a disposizione molte specificazioni alternative per il *trend*

***Trend* quadratico**

$$T_t = \beta_0 + \beta_1 t + \beta_2 t^2$$

***Trend* cubico**

$$T_t = \beta_0 + \beta_1 t + \beta_2 t^2 + \beta_3 t^3$$

***Trend* di grado r**

$$T_t = \beta_0 + \beta_1 t + \beta_2 t^2 + \dots + \beta_r t^r$$

Tutti questi modelli rappresentano ancora modelli lineari (nei parametri), quindi possiamo stimarli utilizzando il metodo *OLS*.

Esistono anche modelli *non* lineari, che *non* possono essere stimati con i minimi quadrati. Ad esempio il modello con

***Trend* esponenziale**

$$T_t = \beta_0 \exp \{\beta_1 t\}$$

non può essere stimato con i minimi quadrati se non dopo aver subito una trasformazione.

I dettagli per la stima di questi modelli ci porterebbe lontano dallo scopo del capitolo. In breve, è possibile verificare che nessuno di questi modelli è adeguato per i nostri dati. Intuitivamente, né la funzione

esponenziale né le funzioni polinomiali si prestano a mimare il comportamento *stagionale* della nostra serie storica.

Modelli stagionali

Qui, il termine “stagionalità” ha un significato diverso da quello a cui siamo abituati. A parte piccole fluttuazioni, la nostra serie storica tende a ripetersi con valori molto simili nei diversi mesi dell’anno: i valori dei mesi di gennaio sono simili tra loro, e diversi dai valori degli altri mesi. La stessa cosa vale per gli altri mesi dell’anno.

Sempre in maniera informale, diremo che

una serie storica è caratterizzata da stagionalità se presenta variazioni periodiche ad intervalli fissi.

Possiamo evidenziare questo comportamento utilizzando una serie di `boxplot`, suddivisi per mese

```
# boxplot ‘stagionali’
boxplot(y ~ month)
```

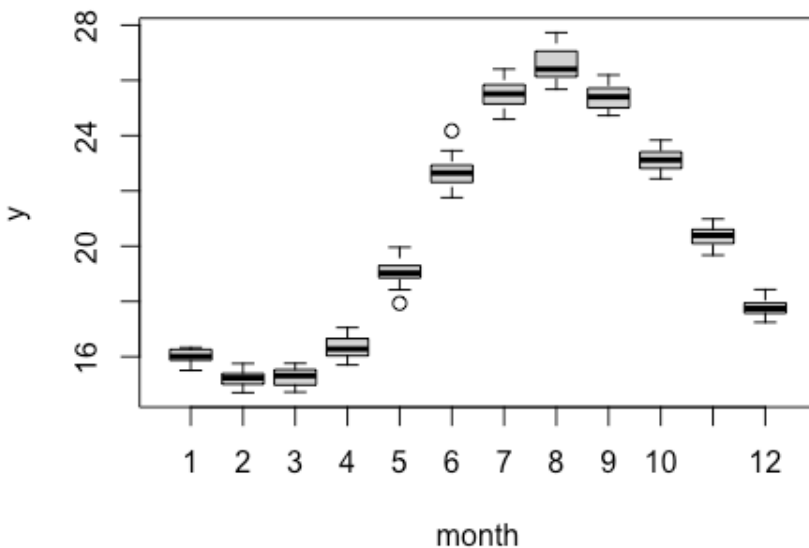


Figura 111 - Boxplots suddivisi per mese.

Qui, i singoli boxplot contengono tutti i valori di un singolo mese nei diversi anni. La natura fortemente stagionale del fenomeno si traduce in una serie di boxplot caratterizzati da un'ampiezza piuttosto limitata rispetto alla variabilità totale.

Un altro tipo di grafico che viene spesso utilizzato per mettere in evidenza la stagionalità della variabile è il *season plot*.

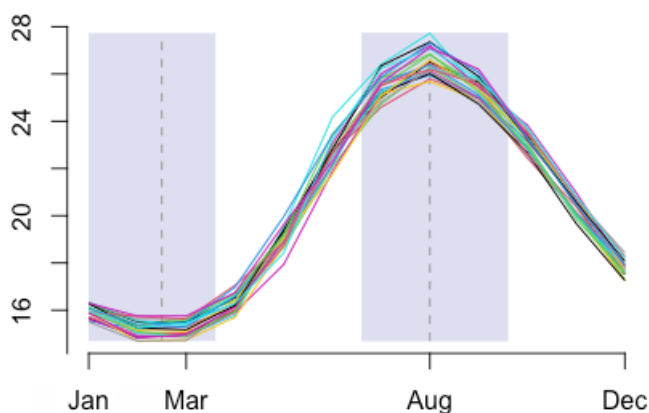


Figura 112 - Season plot.

Il grafico rappresenta i dati di ciascun anno con una linea, ed ogni annualità ha un colore diverso. Sono state anche messe in risalto, con un colore di sfondo, le stagioni astronomiche.

Per i nostri dati

- il *pattern* stagionale è evidente,
- il minimo è spesso raggiunto tra febbraio e marzo, mentre il massimo si ottiene nel mese di agosto.

Gli ultimi due grafici mettono in evidenza una relazione tra la temperatura ed il tempo che non è di lungo periodo: l'*eventuale* presenza di un *trend* lineare, infatti, non esclude altri tipi di forme di dipendenza. Cerchiamo di formalizzare questo tipo di relazione. Se ci chiedessero una previsione per la temperatura del prossimo mese di marzo, ci basterebbe un'occhiata ad uno di questi due grafici per poter azzardare un valore prossimo a 15 gradi. Analogamente, per il prossimo agosto, possiamo prevedere una temperatura pari a circa 26 gradi.

Quindi, possiamo immaginare che la temperatura sia un fenomeno caratterizzato da un valore atteso diverso per ciascuno dei dodici mesi dell'anno. Come possiamo fare per integrare la stagionalità nel nostro modello, ed ottenere questo tipo di previsioni in maniera più formale?

Regressione con variabili *dummy* stagionali

Possiamo tenere conto della stagionalità in un modello di regressione usando delle opportune variabili, dette *dummy*. Le variabili *dummy* ricodificano con valori numerici (tipicamente 0 e 1) i valori di una variabile categorica binaria.

Questo ci permette di utilizzare anche variabili categoriche come predittori in un modello di regressione

- se la variabile assume solamente due valori, basterà ricodificarla usando una singola variabile *dummy*,
- se la variabile assume J valori distinti, possiamo ricodificarla usando una J variabili *dummy*.

Ipotizziamo che la nostra serie storica y_t presenti un comportamento puramente stagionale

$$y_t = S_t + \varepsilon_t$$

e che il *pattern* stagionale si ripeta dopo J periodi. L'idea è quella di includere nel modello J variabili *dummy* per permettere che le osservazioni abbiano un valore atteso diverso per ogni periodo

$$S_t = \gamma_1 D_{t1} + \gamma_2 D_{t2} + \gamma_j D_{tj} + \dots + \gamma_J D_{tJ}$$

dove

- la variabile *dummy* D_{tj} è pari ad 1 se la t -esima osservazione corrisponde al periodo j , e pari a 0 negli altri casi,
- i parametri γ_j sono chiamati *fattori stagionali* e descrivono l'andamento stagionale nel corso dell'anno. Il valore di un generico γ_j rappresenta il livello medio delle osservazioni nel mese j .

Possiamo quindi scrivere il modello di stagionalità pura

$$y_t = \sum_{j=1}^J \gamma_j D_{tj} + \varepsilon_t$$

In questo modello, i parametri γ_j possono essere interpretati come delle intercette, una per ogni periodo. Ne segue che questo modello non contiene alcun parametro di pendenza, né un'intercetta globale.

La matrice disegno di questo modello riunisce tutte le variabili *dummy*

t	D ₁	D ₂	D ₃	...	D _J
1	1	0	0	⋮	0
2	0	1	0	⋮	0
3	0	0	1	⋮	0
⋮	⋮	⋮	⋮	⋮	⋮
J	0	0	0	⋮	1
J + 1	1	0	0	⋮	0
J + 2	0	1	0	⋮	0
⋮	⋮	⋮	⋮	⋮	⋮

Proviamo a stimare il modello utilizzando i dati sulle temperature. La stagionalità si ripete dopo ogni $J = 12$ mesi. Dovremmo quindi definire 12 variabili *dummy*.

In alternativa, è possibile utilizzare la variabile categorica *month*, che abbiamo già disponibile in memoria. In fase di importazione dei dati, avevamo caricato questa variabile sotto forma di *factor*. Inserendo, infatti, un oggetto di questo tipo nel comando di stima di un modello di regressione, R provvederà automaticamente alla conversione di questa variabile nella matrice di variabili *dummy* che intendiamo utilizzare.

```
# Modello di pura stagionalità
fitS = lm(y ~ month - 1)
```

Si noti che

- il suffisso S nel nome dell'oggetto `fitS` sta per *stagionale*,

- il -1 presente nella formula richiede la stima di un modello senza l'intercetta globale,
- la matrice disegno del modello coincide con quella del nostro modello stagionale.

```
head(model.matrix(fitS))
##   month1 month2 month3 month4 month5 month6 month7 month8 month9
month10
## 1      1      0      0      0      0      0      0      0      0
## 2      0      1      0      0      0      0      0      0      0
## 3      0      0      1      0      0      0      0      0      0
## 4      0      0      0      1      0      0      0      0      0
## 5      0      0      0      0      1      0      0      0      0
## 6      0      0      0      0      0      1      0      0      0
##   month11 month12
## 1         0         0
## 2         0         0
## 3         0         0
## 4         0         0
## 5         0         0
## 6         0         0
```

Riportiamo quindi la stima puntuale dei parametri del modello.

```
summary(fitS)
##
## Call:
## lm(formula = y ~ month - 1)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.10678 -0.29876 -0.01214  0.27619  1.48982
##
## Coefficients:
##           Estimate Std. Error t value Pr(>|t|)
## month1    15.98713     0.08821   181.2   <2e-16 ***
```



```
## month2 15.21194 0.08821 172.4 <2e-16 ***
## month3 15.25631 0.08821 173.0 <2e-16 ***
## month4 16.33425 0.08821 185.2 <2e-16 ***
## month5 19.03940 0.08821 215.8 <2e-16 ***
## month6 22.68391 0.08821 257.2 <2e-16 ***
## month7 25.48465 0.08821 288.9 <2e-16 ***
## month8 26.53999 0.08821 300.9 <2e-16 ***
## month9 25.39227 0.08821 287.9 <2e-16 ***
## month10 23.11192 0.08821 262.0 <2e-16 ***
## month11 20.34735 0.09029 225.4 <2e-16 ***
## month12 17.76853 0.09029 196.8 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4137 on 250 degrees of freedom
## Multiple R-squared:  0.9996, Adjusted R-squared:  0.9996
## F-statistic: 5.454e+04 on 12 and 250 DF,  p-value: < 2.2e-16
```

$$\hat{y} = \frac{15.987}{(0.088)} D_{t,1} + \frac{15.212}{(0.088)} D_{t,2} + \dots + \frac{17.769}{(0.09)} D_{t,12},$$

$$\hat{\sigma}^2 = 0.414^2$$

Il valore del j -esimo coefficiente corrisponde al valore atteso della serie nel mese j .

A prima vista, questo modello appare migliore del precedente. Infatti

- tutti i coefficienti risultano significativi,
- il valore della varianza di regressione è diminuito rispetto al modello con *trend* lineare,
- l'indice di determinazione ha un valore molto elevato.

Previsioni

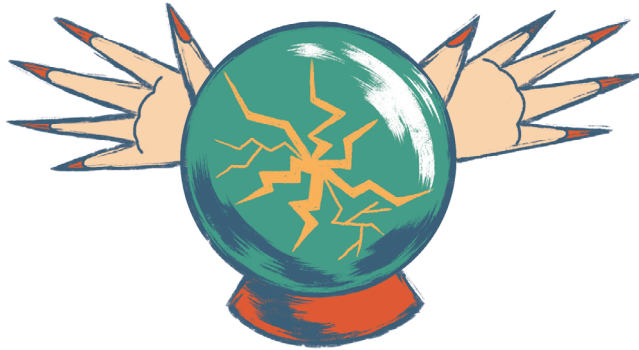


Figura 113 - I modelli predittivi, come dei moderni oracoli, cercano di decifrare il linguaggio nascosto dei dati.

Le *previsioni puntuali* possono essere ottenute come al solito

$$\hat{y}_{t+h|T} = \sum_{j=1}^J \hat{v}_j D_{T+h,j}$$

Anche se questa rappresentazione può apparire complicata, la previsione sarà pari al coefficiente $\hat{v}_j$ relativo al mese di interesse. Infatti, tutte le variabili *dummy* coinvolte nella sommatoria, tranne una, saranno pari a zero.

Possiamo ottenere previsioni puntuali e intervalli di previsione in *R*. Nel nostro esempio, siamo interessati ai prossimi due mesi (novembre e dicembre). Possiamo codificare numericamente questi mesi con i numeri 11 e 12, e utilizzare il comando

```
# Previsioni
mesi = as.factor(11:12)
predict(fitS, newdata=data.frame(month=mesi), interval='prediction')
##      fit      lwr      upr
## 1 20.34735 19.51330 21.18139
## 2 17.76853 16.93449 18.60258
```

La figura seguente mostra le osservazioni degli ultimi tre anni e gli intervalli di previsione per novembre e dicembre 2020. Si nota come le previsioni puntuali sono adesso più coerenti con il *pattern* stagionale della serie storica, e come gli intervalli abbiano adesso un’ampiezza molto più limitata.

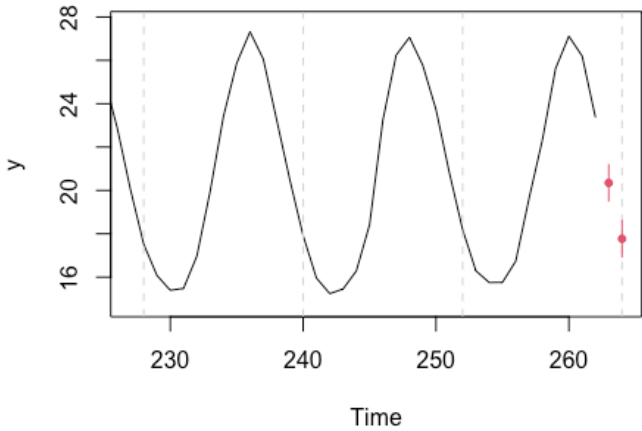


Figura 114 - Osservazioni degli ultimi tre anni ed intervalli di previsione dei mesi novembre e dicembre 2020.

Aggiungendo, all’ultimo grafico, le osservazioni del periodo di validazione, possiamo notare come entrambe le osservazioni “future” siano comprese all’interno dei rispettivi intervalli di previsione:

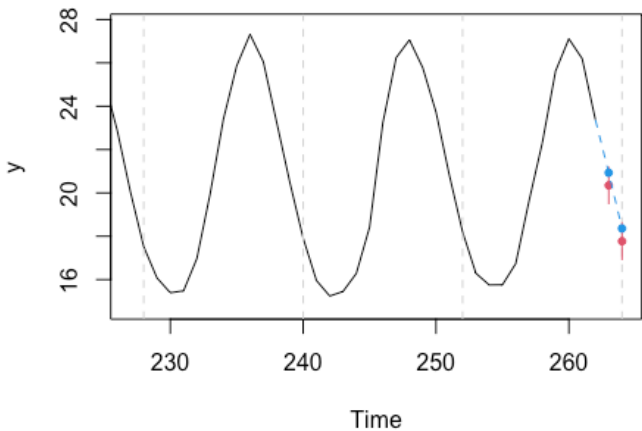


Figura 115 - Aggiunta delle osservazioni future.

Pertanto, possiamo aggiornare le nostre risposte rispetto agli *obiettivi della ricerca* che ci eravamo posti

- la temperatura prevista per novembre 2020 è di 20.347 gradi,
- la temperatura prevista per dicembre 2020 è di 17.769 gradi,
- il modello non prevede una componente di lungo periodo, e quindi esclude *per assunto* che la temperatura salga nel tempo,
- anche in questo modello non è incluso il livello della clorofilla, e quindi non possiamo dare una risposta all'ultima domanda.

Diagnostiche del modello stagionale

Abbiamo già notato che il modello mostra un indice di determinazione elevato. Per dare un'impressione grafica dell'adattamento del modello ai dati, è possibile confrontare le osservazioni con i valori predetti.

I valori predetti, che salveremo nell'oggetto `yHat` possono essere ottenuti con il comando

```
yHat = fitS$fitted.values
```

Ora possiamo tracciare la serie temporale originale e quella dei valori predetti

```
plot.ts(y)
lines(yHat, col=2)
```

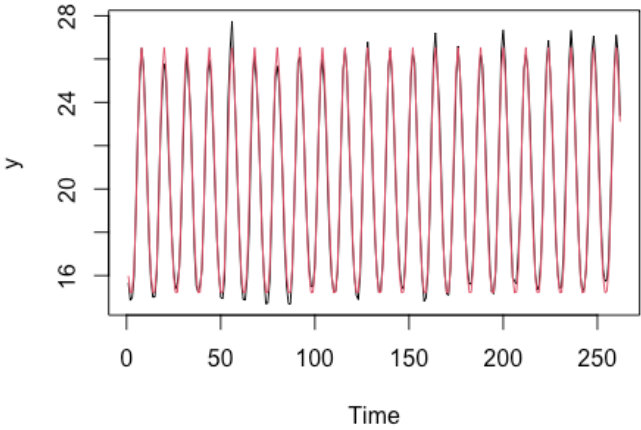


Figura 116 - Serie temporale originale (nero) e serie temporale dei valori predetti (rosso).

Il grafico mostra un adattamento apparentemente perfetto del modello ai dati. La serie di `boxplot` mensili dei residui, evidenzia come il modello sia riuscito a catturare il comportamento stagionale

```
# Residui
e = fit$residuals
boxplot(e ~ month)
```

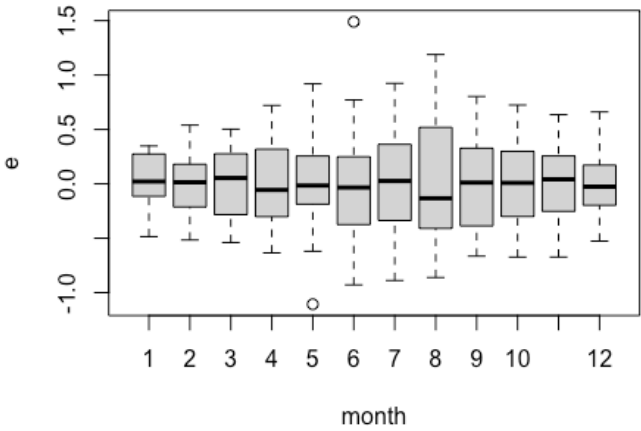


Figura 117 - Boxplots dei residui per mese.

È troppo presto per cantare vittoria! Uno sguardo più approfondito mostra che il nostro modello ha ancora dei problemi di specificazione: se si disegna il vettore dei residui, si può notare come il livello di questa serie abbia un *trend* crescente

```
# Grafico dei residui
plot.ts(e)
abline(h=0, col=2)
```

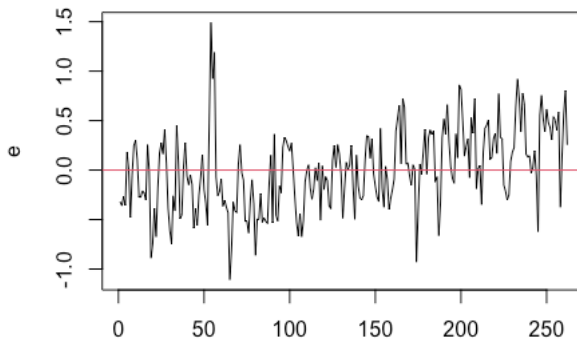


Figura 118 - Vettore dei residui.

Probabilmente, è stato un errore aver escluso il *trend* dal nostro modello: la componente di lungo periodo e la componente stagionale non sono alternative tra loro. Piuttosto, riflettono aspetti diversi dello stesso fenomeno. Per questo motivo, può essere importante che un modello preveda entrambe le componenti.

Un modello più complesso

Stimeremo quindi il modello

$$y_t = T_t + S_t + \varepsilon_t$$

dove

- T_t rappresenta il *trend*,
- S_t rappresenta la stagionalità,
- ε_t è il termine di errore.

Più nel dettaglio, il nostro modello *con tendenza lineare e variabilità stagionale* avrà la forma

$$y_t = \beta t + \sum_{j=1}^J \gamma_j D_{tj} + \varepsilon_t$$

Possiamo stimare anche questo modello utilizzando *OLS*

```
# Modello con trend e stagionalita'
fitTS = lm(y ~ tt + month - 1)
summary(fitTS)
##
## Call:
## lm(formula = y ~ tt + month - 1)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.02837 -0.24207 -0.02209  0.24639  1.67526
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## tt          2.377e-03  3.035e-04   7.833 1.38e-13 ***
## month1      1.569e+01  8.806e-02 178.128 < 2e-16 ***
## month2      1.491e+01  8.819e-02 169.042 < 2e-16 ***
## month3      1.495e+01  8.832e-02 169.261 < 2e-16 ***
```

```
## month4 1.603e+01 8.846e-02 181.161 < 2e-16 ***
## month5 1.873e+01 8.859e-02 211.391 < 2e-16 ***
## month6 2.237e+01 8.873e-02 252.113 < 2e-16 ***
## month7 2.517e+01 8.887e-02 283.212 < 2e-16 ***
## month8 2.622e+01 8.901e-02 294.602 < 2e-16 ***
## month9 2.507e+01 8.915e-02 281.241 < 2e-16 ***
## month10 2.279e+01 8.929e-02 255.233 < 2e-16 ***
## month11 2.004e+01 9.026e-02 221.973 < 2e-16 ***
## month12 1.745e+01 9.040e-02 193.090 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3713 on 249 degrees of freedom
## Multiple R-squared:  0.9997, Adjusted R-squared:  0.9997
## F-statistic: 6.251e+04 on 13 and 249 DF,  p-value: < 2.2e-16
```

Le stime del modello possono essere interpretate come di consueto. Si noti che

- il modello non include l'intercetta,
- a differenza del modello “solo *trend*”, in questo modello, il coefficiente associato al tempo è significativo, anche se con un valore assoluto praticamente dimezzato. Secondo questo modello, la temperatura cresce di 2.4 millesimi di grado, piuttosto che di 4 millesimi di grado, al mese.

Le previsioni per questo modello sono date da

$$\hat{y}_{T+h|T} = \hat{\beta} (T + h) + \sum_{j=1}^J \hat{\gamma}_j D_{T+h,j}$$

```
# Previsioni
predict(fitTS,
  newdata=data.frame(tt=(Ti+1):(Ti+2), month=mesi),
  interval='prediction')
##          fit          lwr          upr
## 1 20.66116 19.90843 21.41389
## 2 18.08234 17.32961 18.83507
```


Rispetto agli *obiettivi della ricerca*,

- le previsioni sono, tutto sommato, simili a quelle già ottenute con il modello stagionale,
- quello che cambia drasticamente è la conclusione rispetto all'ipotesi del riscaldamento di lungo periodo. L'ipotesi, non supportata dai modelli precedenti, riceve qui un deciso supporto. Infatti, il valore-p per il test di significatività del coefficiente associato al *trend* è prossimo allo zero. Se, quindi, crediamo che la specificazione di questo modello sia corretta, concluderemo che la temperatura si sta innalzando nel tempo.

Diagnostiche

Arrivati a questo punto, torniamo a dare un'occhiata all'aspetto dei residui del modello stimato:

```
# Residui  
e = fitTS$residuals  
boxplot(e ~ month)
```

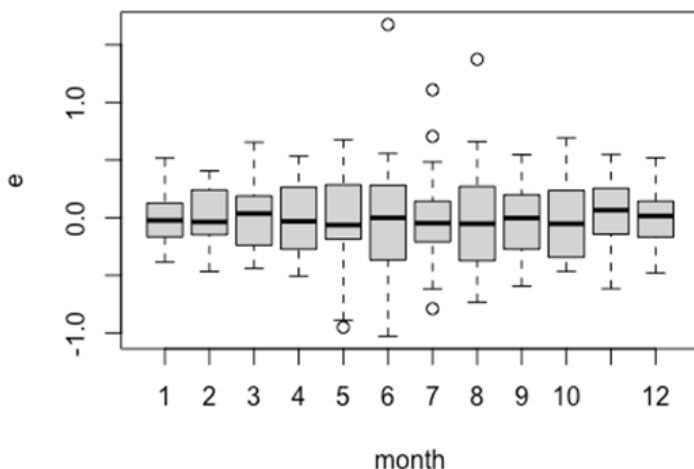


Figura 119 - Residui del modello stimato.

Sembra che i residui siano stati ripuliti dalla stagionalità e, a parte qualche valore anomalo, non mostrano *pattern* particolari nel tempo.

```
# Grafico dei residui
plot.ts(e)
abline(h=0, col = 2)
```

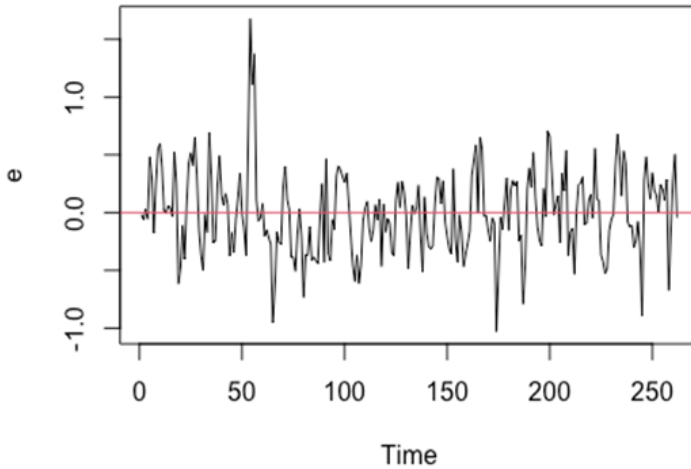


Figura 120 - Vettore dei residui del modello stimato.

Potremmo essere tentati di dire che questo modello si adatta ragionevolmente ai dati e che quindi possiamo ritenere affidabili le risposte che abbiamo ottenuto. Purtroppo, un'analisi più approfondita rivela che esiste ancora una dinamica nascosta nei residui.

Modelli con cicli

La dinamica nascosta può essere mostrata graficamente tracciando ogni residuo e_t , relativo ad una generica osservazione, rispetto al residuo e_{t-1} , relativo all'osservazione precedente

```
# Autocorrelazione
plot(e[1:(Ti-1)], e[2:Ti])
```

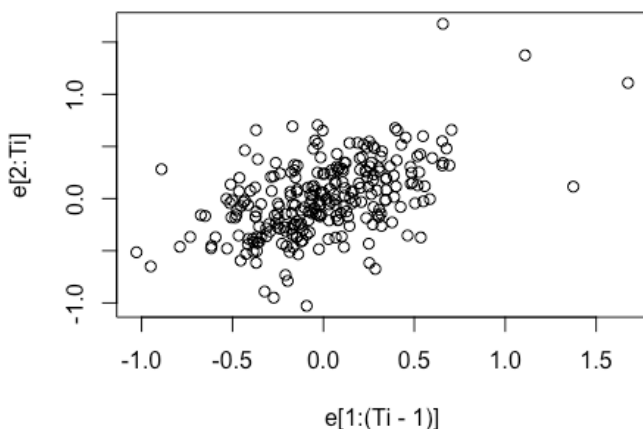


Figura 121 - Autocorrelazione dei residui.

In ascissa troviamo la serie $e_{[1:(T-1)]}$, in ordinata la serie $e_{[2:T]}$. In questo modo, ciascun punto di questo grafico ha coordinate (e_{t-1}, e_t) .

Tra gli assunti alla base dei nostri modelli, abbiamo che gli errori sono *indipendenti* tra loro. Se il modello fosse correttamente specificato, questo assunto implicherebbe che i residui dovrebbero essere incorrelati. Nel grafico, dovremmo quindi osservare una nuvola casuale. Nel nostro caso, al contrario, la presenza di un certo grado di correlazione è evidente anche “ad occhio”.

Come succede molto spesso, trattando serie temporali, le osservazioni sono *serialmente dipendenti*: ogni osservazione dipende dal valore di quelle precedenti.

Dato che il modello che abbiamo stimato non riesce a tenere in considerazione questa dipendenza seriale, possiamo vedere questa dinamica ancora presente nei residui del modello.

(Amara) conclusione: Nessuno dei modelli precedenti è specificato correttamente, poiché l'ipotesi di indipendenza tra le osservazioni, necessaria per garantire le proprietà degli stimatori, è violata!

Per poter ottenere delle inferenze affidabili, dovremo quindi sostituire l'assunto di indipendenza con una coppia di assunti riguardanti la *stazionarietà* e l'*ergodicità* del meccanismo che ha generato la serie storica.

Questi argomenti meriterebbero un'analisi approfondita, ma questa ci porterebbe lontano dai nostri scopi. In maniera molto informale, la

stazionarietà implica che il fenomeno oggetto di studio abbia caratteristiche che non variano nel tempo e che quindi, ad esempio, posso utilizzare le osservazioni del passato per prevedere il futuro. L'ergodicità, invece, ci permette di ritenere che valori della serie storica che siano temporalmente “molto distanti” possano essere considerati tra loro indipendenti.

I modelli per serie storiche sono tipicamente più complessi di quelli visti finora, e spesso non possono essere stimati con l'approccio dei minimi quadrati ordinari. Per rimanere in ambito *OLS*, noi considereremo il modello autoregressivo, ma questo modello può essere generalizzato in molte direzioni.

Correlogramma

Introduciamo, innanzi tutto, il concetto di *correlogramma*. Cerchiamo di calcolare l'entità della correlazione tra residui successivi

```
# Autocorrelazione a lag 1
cor(e[1:(Ti-1)], e[2:Ti])
## [1] 0.5167122
```

Chiameremo questa quantità *autocorrelazione a lag 1* in quanto la correlazione viene calcolata tra la serie dei residui e la serie del ritardo (o *lag*) primo degli stessi residui. Il valore di questa autocorrelazione è coerente con le considerazioni svolte sulla base della figura: la correlazione sembra positiva, anche se non prossima ad uno.

In maniera analoga, è possibile calcolare l'autocorrelazione a *lag 2*

```
# Autocorrelazione a lag 2
cor(e[1:(Ti-2)], e[3:Ti])
## [1] 0.2215574
```

o per qualsiasi altro valore del *lag*. L'insieme dei coefficienti di autocorrelazione ai ritardi $\{1, 2, \dots\}$ è riassunto nella *funzione di autocorrelazione* (*AutoCorrelation Function*, *ACF*).

La funzione `acf` calcola i valori dei coefficienti della *ACF* e li disegna automaticamente

```
# Funzione di autocorrelazione
acf(e)
```

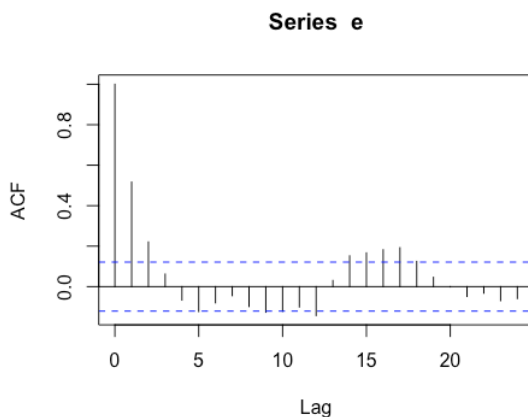


Figura 122 - Coefficienti di correlazione calcolati dalla funzione `acf`.

Nel grafico precedente, ogni barra rappresenta un coefficiente di correlazione. È normale che la prima barra (relativa all'autocorrelazione al ritardo 0) sia pari a 1, in quanto stiamo calcolando la correlazione tra la serie e_t e sé stessa. Le due barre successive riportano i valori appena calcolati (0.517 a *lag* 1 e 0.221 a *lag* 2). In più, abbiamo i valori fino al ritardo 24.

Nel grafico si hanno anche due righe orizzontali, che delimitano la c.d. "banda di Bartlett": ciascun coefficiente che ecceda la banda può essere ritenuto significativamente diverso da zero ad un livello di significatività del 5%. In questo modo, la presenza di coefficienti all'esterno della banda evidenzia la presenza di correlazione (e, quindi, dipendenza) seriale tra le osservazioni.

In questo caso, lasciando da parte il coefficiente a *lag* 0, possiamo vedere che ci sono altri coefficienti che sono significativamente diversi da zero. Abbiamo quindi un'evidenza contro l'assunto di indipendenza che abbiamo introdotto nella sezione relativa allo *stimatore dei minimi quadrati*, e quindi contro la specificazione del modello.

N.B. Il correlogramma è in realtà composto da due grafici. Formalmente, dovremmo ottenere anche il grafico della c.d. *funzione di autocorrelazione parziale*. Il significato e l'utilizzo di questo secondo grafico, tuttavia, eccede lo scopo di questo capitolo.

Il modello autoregressivo

A questo punto, quindi, non possiamo fare altro che rimboccarci le maniche e riprendere l'ultima caratteristica che considereremo: i cicli. Perché? La deduzione dovrebbe essere chiara: con i modelli precedenti abbiamo provato a catturare la tendenza di lungo periodo e la stagionalità, ma non è bastato. I dati (anzi, i residui) parlano chiaro: ogni osservazione dipende dalle osservazioni precedenti, e senza considerare questo aspetto non riusciremo a ottenere un modello correttamente specificato.

La presenza di dipendenza seriale comporta la presenza di un andamento *ciclico* della serie storica, dove fasi di espansione si alternano a fasi di recessione. Le fasi non hanno in generale una periodicità predefinita, né una durata costante.

I cicli rappresentano le deviazioni temporanee dal trend che la serie storica presenta.

È chiaro che questa definizione informale è molto ampia e generica. Ma perché i cicli? Torniamo all'esempio delle variazioni climatiche su scale temporali ampie. Sappiamo per certo che la temperatura varia su base stagionale e che ha avuto una tendenza all'aumento, almeno a partire dalla rivoluzione industriale. A questo, però, si aggiungono ad altri fenomeni ciclici di ampia scala, come ad esempio i cicli di Milanković. Per ottenere una corretta specificazione del nostro modello, è quindi necessario tenere in considerazione queste dinamiche aggiuntive.

Possiamo tenere conto dei cicli in un modello di regressione

$$y_t = C_t + \varepsilon_t, \quad \varepsilon_t \stackrel{iid}{\sim} D(0, \sigma^2)$$

Il modello più semplice che possiamo utilizzare è un modello di regressione in cui le osservazioni passate vengono utilizzate per prevedere quella attuale.

$$y_t : \Phi_0 + \Phi_1 y_{t-1} + \varepsilon_t \quad \varepsilon_t \stackrel{iid}{\sim} D(0, \sigma^2)$$

Questo è un modello autoregressivo di ordine 1, o $AR(1)$. È qui che ci ha condotto la necessità di considerare il fatto che i dati non sono indipendenti e identicamente distribuiti.

È ancora possibile, anche se non efficiente, stimare questo modello utilizzando OLS. Questo metodo di stima ci ha permesso, finora, di focalizzarci sulla sequenza di passaggi logici che ci ha portato a questo tipo di modelli piuttosto che sui dettagli relativi allo stimatore. Per i modelli che andremo ora a considerare questo rimarrà valido, anche se negli script che useremo sarà necessario inserire qualche comando aggiuntivo, per il corretto allineamento delle serie storiche.

Ogni osservazione y_t andrà infatti regredita sull'osservazione precedente y_{t-1} . Così, ad esempio, metteremo in relazione l'osservazione y_2 con y_1 , l'osservazione y_3 con y_2 e così via.

Tuttavia, l'osservazione y_1 sarà messa in relazione con y_0 , che è un valore al di fuori dell'intervallo di osservazione della nostra serie storica. Saremo quindi costretti a non utilizzare una osservazione.

Riassumendo, potremo regredire la serie storica $y_{[2:T]}$ sul suo ritardo primo $y_{[1:(T-1)]}$. Per comodità, salveremo

- la serie storica $y_{[2:T]}$, ovvero senza la prima osservazione nell'oggetto y_0 ,
- la serie storica $y_{[1:(T-1)]}$ dei ritardi primi nell'oggetto Ly .

```
# Modello AR(1)
y0 = y[2:Ti]
Ly = y[1:(Ti-1)]
```

Entrambi questi oggetti contengono $T-1$ elementi, in quanto abbiamo perso una osservazione. Possiamo adesso utilizzare questi due oggetti per stimare il modello $AR(1)$

```
fitC = lm(y0 ~ Ly)
summary(fitC)
##
## Call:
```

```
## lm(formula = y0 ~ Ly)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3.1643 -1.8739 -0.4939  1.9829  4.5280
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.88465     0.65416   4.41 1.52e-05 ***
## Ly           0.85908     0.03164  27.15 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.1 on 259 degrees of freedom
## Multiple R-squared:  0.74, Adjusted R-squared:  0.739
## F-statistic: 737.1 on 1 and 259 DF, p-value: < 2.2e-16
```

Il modello stimato risulta essere

$$\hat{y}_T = \underset{(0.654)}{2.558} + \underset{(0.032)}{0.859} y_{t-1}, \quad \hat{\sigma}^2 = 2.1^2.$$

Previsioni

La previsione puntuale per y_{T+1} risulta

$$\hat{y}_{T+1|T} = \hat{\phi}_0 + \hat{\phi}_1 Y_T$$

dove y_T è l'ultimo valore che osserviamo nella nostra serie storica. Nel nostro caso, otteniamo

$$\hat{y}_{T+1|T} = 2.885 + 0.859 \cdot 23.367 = 22.958$$

```
# Previsioni
yHat = numeric(2)
yHat[1] = as.numeric(predict(fitC, newdata=data.frame(Ly=y[Ti])))
```


Per i modelli autoregressivi, la previsione per $h > 1$ è più complicata. Ad esempio, per il caso $h = 2$, abbiamo

$$\hat{y}_{T+2|T} = \hat{\phi}_0 + \hat{\phi}_1 y_{T+1}$$

L'osservazione y_{T+1} non è nella serie storica, che si ferma al tempo T . Non possiamo, quindi, ottenere la previsione per il tempo $T + 2$. Esistono diversi metodi per superare questo problema. Uno di questi metodi segue una procedura iterata, che consiste nel sostituire i valori previsti ai valori non osservabili.

Nel nostro caso, possiamo sostituire la previsione appena ottenuta all'osservazione mancante y_{T+1} .

```
yHat[2] = as.numeric(predict(fitC, newdata = data.frame(Ly =
yHat[1])))
yHat
## [1] 22.95846 22.60775
```

Iterando questa procedura, è possibile ottenere previsioni a qualunque orizzonte h . Se effettuassimo le diagnostiche per questo modello, troveremmo che, nonostante il buon adattamento del modello ai dati, nei residui c'è evidenza di un *trend* e di un comportamento stagionale. Per questo motivo, otterremo un singolo modello che possa rappresentare adeguatamente tutte le caratteristiche finora considerate rispetto al fenomeno oggetto di studio.

Un modello con *trend*, stagionalità e cicli

Possiamo quindi fornire un modello finale per la serie temporale y_t .

$$Y_t = T_t + S_t + C_t + \varepsilon_t.$$

dove

- T_t è il *trend* o andamento di lungo termine,
- S_t è la componente stagionale,
- C_t è la componente ciclica,
- ε_t è il termine di errore.

Abbiamo diversi gradi di libertà nella scelta di ciascuna componente del modello. Consideriamo un modello molto stilizzato, che comprenda un *trend* lineare, una stagionalità distinta per mesi, e un ciclo. Per stimare questo modello, possiamo utilizzare i comandi

```
# Modello con trend, stagionalità e cicli
tt0 = tt[2:Ti]
month0 = month[2:Ti]
fitTSC = lm(y0 ~ tt0 + month0 + Ly - 1)
summary(fitTSC)
##
## Call:
## lm(formula = y0 ~ tt0 + month0 + Ly - 1)
##
## Residuals:
```

	Min	1Q	Median	3Q	Max
	-0.98065	-0.18641	-0.01358	0.19457	1.33540

```
##
## Coefficients:
```

	Estimate	Std. Error	t value	Pr(> t)
tt0	1.145e-03	2.927e-04	3.913	0.000118 ***
month01	6.669e+00	9.539e-01	6.991	2.54e-11 ***
month02	6.804e+00	8.577e-01	7.933	7.42e-14 ***
month03	7.248e+00	8.155e-01	8.888	< 2e-16 ***
month04	8.302e+00	8.178e-01	10.152	< 2e-16 ***

```
## month05 1.045e+01 8.762e-01 11.926 < 2e-16 ***
## month06 1.269e+01 1.023e+00 12.410 < 2e-16 ***
## month07 1.361e+01 1.221e+00 11.148 < 2e-16 ***
## month08 1.322e+01 1.373e+00 9.626 < 2e-16 ***
## month09 1.152e+01 1.430e+00 8.056 3.35e-14 ***
## month10 9.835e+00 1.368e+00 7.190 7.66e-12 ***
## month11 8.261e+00 1.244e+00 6.642 1.96e-10 ***
## month12 7.103e+00 1.094e+00 6.492 4.60e-10 ***
## Ly      5.167e-01 5.448e-02 9.485 < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3192 on 247 degrees of freedom
## Multiple R-squared:  0.9998, Adjusted R-squared:  0.9998
## F-statistic: 7.838e+04 on 14 and 247 DF,  p-value: < 2.2e-16
```

Dato che questo modello contiene una componente autoregressiva di ordine 1, è stato necessario eliminare la prima osservazione della serie storica. In maniera analoga, abbiamo dovuto eliminare la prima osservazione di tutti i regressori (a parte *Ly* che per costruzione già prevede il giusto numero di elementi). È interessante notare che, per la stima di questo modello, il coefficiente associato al tempo risulta ancora significativamente diverso da zero, ma il suo valore viene ulteriormente dimezzato rispetto al modello con *trend* e stagionalità: per questo modello, la temperatura cresce di 1.1 millesimi di grado al mese.

Previsioni

Per ottenere le previsioni da questo modello, è necessario utilizzare la procedura iterata appena considerata

```
# Previsione
yHat = numeric(2)
yHat[1] = as.numeric(predict(fitTSC, newdata = data.frame(tt0 = Ti+1,
                                                             month0 = as.factor(11), Ly = y[Ti])))
yHat[2] = as.numeric(predict(fitTSC, newdata = data.frame(tt0 = Ti+2,
                                                             month0 = as.factor(12), Ly = yHat[1])))
yHat
## [1] 20.63643 18.06907
```

Queste previsioni sono, tutto sommato, simili a quelle fornite dal modello con *trend* e stagionalità. Tuttavia, le diagnostiche del modello mostrano un comportamento migliore dei residui.

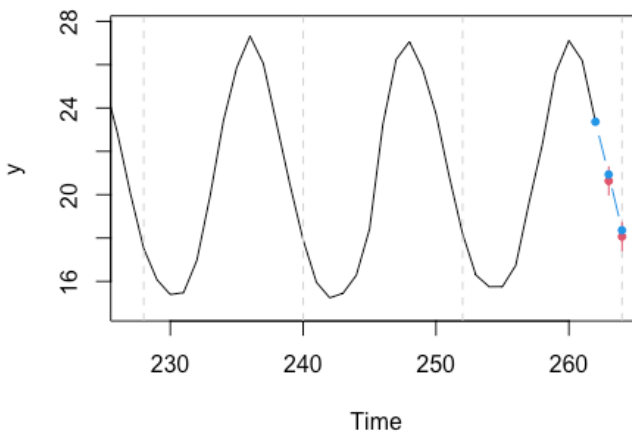


Figura 123 - Previsioni puntuali ed intervalli di previsione del modello con trend, stagionalità e cicli.

Diagnostiche del modello

Semplici controlli sui residui non mostrano alcuna prova contro l'errata specificazione del modello:

```
# Residui  
e = fitTSC$residuals  
plot.ts(e)  
abline(h=0, col = 2)
```

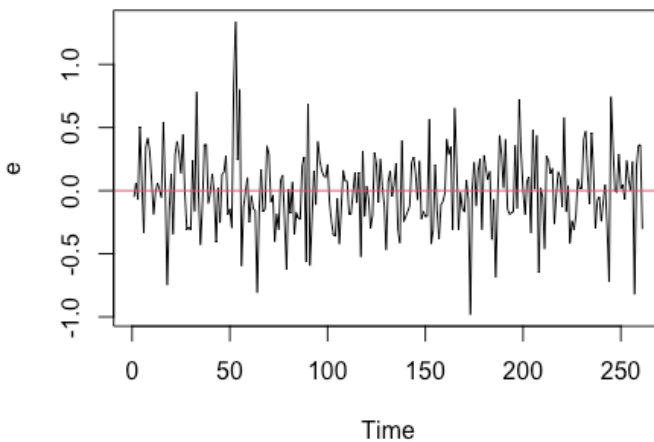


Figura 124 - Serie storica dei residui del nuovo modello.

La serie storica dei residui non mostra *pattern* particolari e, a parte qualche *outlier*, non suggerisce che la specificazione del modello sia errata. Anche la funzione di autocorrelazione non evidenzia problemi con la specificazione del modello.

```
# Coefficienti di autocorrelazione  
acf(e)
```

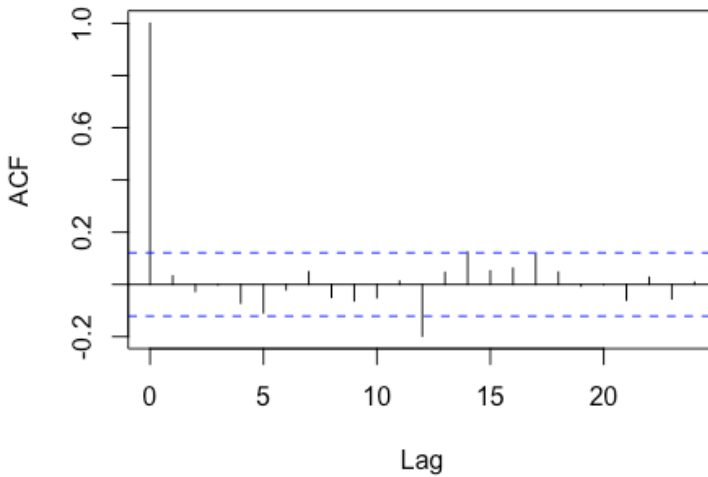
Series e

Figura 125 - Coefficienti di autocorrelazione del nuovo modello.

Qualcuno potrebbe obiettare che ci sono alcuni coefficienti che “toccano” il bordo della banda o che addirittura fuoriescono. Per rispondere a questa osservazione, si noti che la banda di Bartlett è costruita per effettuare verifiche di significatività con un livello di confidenza del 95%. Non solo ci aspettiamo che qualche coefficiente fuoriesca dalla banda, ma questo è anche un indice che la verifica funziona correttamente.

Selezione del modello

L'ultimo modello che abbiamo sviluppato prevede:

- un *trend* lineare,
- una stagionalità mensile,
- un struttura autoregressiva $AR(1)$.

Questo modello ci ha permesso di illustrare tutte le caratteristiche salienti del fenomeno che abbiamo studiato, e non lo complicheremo ulteriormente. Tuttavia, esiste un'ampia gamma di alternative che non sono state esplorate. Ad esempio:

- il *trend* può essere polinomiale,
- la stagionalità può essere definita in base alle 4 stagioni astronomiche, piuttosto che ai 12 mesi,
- l'ordine autoregressivo può essere superiore ad uno.

Prima di trarre le nostre conclusioni, dovremmo quindi considerare una rosa di modelli candidati, e tra questi scegliere il "migliore", ad esempio utilizzando un *criterio di informazione* come il *BIC* (*Bayesian Information Criterion*).

Per utilizzare questo criterio, bisogna stimare tutti i modelli candidati e calcolare, per ognuno di essi, il valore del criterio. Selezioneremo il modello che *minimizzi* il valore del criterio.

Lasciamo al lettore interessato l'ingrato compito di esplorare tutte queste possibilità. Se rimaniamo nell'ambito di modelli che possono venire stimati tramite *OLS*, possiamo comunque contare sulla funzione *BIC*. Ad esempio, possiamo calcolare facilmente il valore del criterio per l'ultimo modello stimato:

```
# Selezione del modello
BIC(fitTSC)
## [1] 213.6913
```

e paragonarlo con il valore dello stesso criterio per qualsiasi altro modello alternativo.

Conclusioni

Anche se ci sarebbe ancora un lungo lavoro di selezione del modello da svolgere, traiamo le nostre conclusioni rispetto agli *obiettivi della ricerca*, sulla base del modello finale:

- la temperatura prevista per novembre 2020 è pari a 20.64 gradi centigradi,
- per dicembre, si prevede una temperatura pari a 18.07 gradi centigradi,
- il coefficiente associato al *trend* è significativamente diverso da zero, la temperatura cresce di circa 1.1 millesimi di grado al mese,
- questo modello non ci può essere di aiuto a rispondere all'ultima domanda, in quanto il livello della clorofilla non è incluso nel modello.

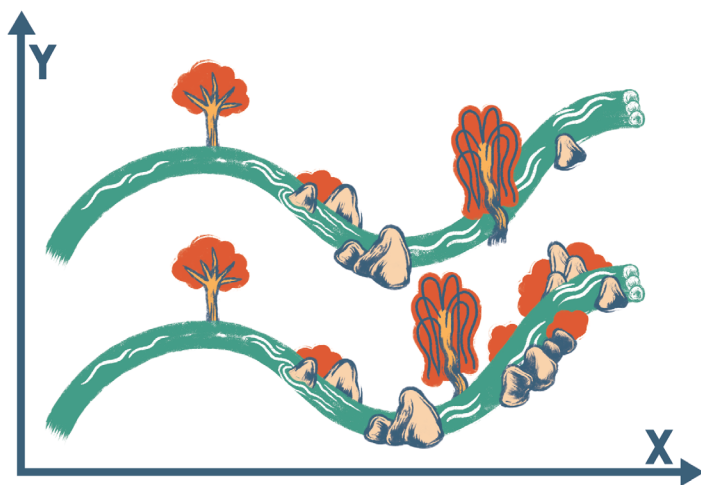
In particolare, ripercorrendo le tappe di questo capitolo, possiamo notare come abbiamo modificato più volte le nostre conclusioni rispetto all'ipotesi del riscaldamento delle acque dell'Adriatico:

- sulla base del modello con *trend* lineare, abbiamo concluso che non c'erano prove a favore dell'ipotesi del riscaldamento globale;
- d'altronde, il modello era troppo poco flessibile, e mostrava un cattivo adattamento ai dati. Il modello stagionale puro forniva un miglior adattamento. Questo secondo modello escludeva l'ipotesi del riscaldamento per assunzione;
- consentendo la presenza di un *trend* all'interno di un modello stagionale abbiamo migliorato ulteriormente l'adattamento del modello ai dati. La stima di questo modello supporta la tesi del riscaldamento, pur dimezzando l'intensità dell'aumento mensile rispetto al modello lineare;
- data la presenza di una struttura di correlazione seriale significativa, abbiamo infine consentito anche un comportamento autoregressivo nel nostro modello, ottenendo il nostro modello finale. Questo modello supporta ancora l'ipotesi del riscaldamento, anche se diminuisce ulteriormente la stima del coefficiente.

Queste conclusioni tra loro discordanti non sono un indice di inaffidabilità del metodo che stiamo utilizzando. Piuttosto, questa discordanza rivela la necessità di una ricerca accurata di una adeguata specificazione del modello.

Cerchiamo adesso di dare una risposta all'ultimo quesito.

Il modello ADL



L'ultima domanda ci pone davanti ad un problema di non facile soluzione. Innanzitutto, ci dovremo chiedere se, sul piano logico, esiste una relazione tra le due variabili, e quale forma possa assumere tale relazione. I dati non ci vengono in aiuto in questo compito: trovare una correlazione campionaria tra due variabili non equivale ad aver trovato un nesso causale tra le stesse.

Se è logicamente plausibile che la relazione esista, allora dovremo confermare l'esistenza di tale relazione tramite l'evidenza campionaria. Nel nostro caso, sembra plausibile che, al crescere della temperatura, il livello di clorofilla diminuisca. Dovremo quindi specificare un modello per il livello della clorofilla, che contenga, tra i regressori, anche la temperatura.

È possibile includere questo regressore in uno qualunque dei modelli appena visti. Tuttavia, ripetendo i passi dell'analisi, troviamo che il modello con stagionalità e ciclo, e senza *trend*, possa essere ritenuto "accettabile" per la clorofilla. Ovviamente, questo modello dovrà considerare anche il regressore di interesse

$$z_t = \sum_{j=1}^J \gamma_j D_{tj} + \phi_1 z_{t-1} + \beta y_{t-1} + \epsilon_t$$

Otterremo, in questo modo, un modello *ADL* (*Autoregressive Distributed Lag*). Questo tipo di modelli sfrutta uno o più *lag* di una variabile esogena per spiegare la variabilità della serie storica di interesse.

Per stimare il modello, possiamo utilizzare i seguenti comandi

```
# Modello ADL
z0 = z[2:Ti]
Lz = z[1:(Ti-1)]
fitADL = lm(z0 ~ Lz + Ly + month0 - 1)
summary(fitADL)
##
## Call:
## lm(formula = z0 ~ Lz + Ly + month0 - 1)
##
## Residuals:
```

	Min	1Q	Median	3Q	Max
	-0.0105121	-0.0025647	0.0000808	0.0022478	0.0147237

```
##
## Coefficients:
```

	Estimate	Std. Error	t value	Pr(> t)	
Lz	0.4946234	0.0537852	9.196	< 2e-16	***
Ly	-0.0020631	0.0006331	-3.259	0.00128	**
month01	0.0789592	0.0128722	6.134	3.38e-09	***
month02	0.0931993	0.0117149	7.956	6.43e-14	***
month03	0.0909854	0.0117341	7.754	2.33e-13	***
month04	0.0988762	0.0119844	8.250	9.48e-15	***
month05	0.1349636	0.0129401	10.430	< 2e-16	***
month06	0.1134436	0.0157218	7.216	6.57e-12	***
month07	0.1042145	0.0174541	5.971	8.17e-09	***
month08	0.1134493	0.0184635	6.145	3.19e-09	***
month09	0.1175312	0.0189095	6.215	2.16e-09	***
month010	0.1080213	0.0181978	5.936	9.84e-09	***
month011	0.0943700	0.0166485	5.668	4.01e-08	***
month012	0.0826178	0.0147042	5.619	5.18e-08	***

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##  
## Residual standard error: 0.004108 on 247 degrees of freedom  
## Multiple R-squared:  0.9989, Adjusted R-squared:  0.9989  
## F-statistic: 1.655e+04 on 14 and 247 DF, p-value: < 2.2e-16
```

Sempre assumendo che la specificazione di questo modello sia corretta, possiamo quindi rispondere all'ultimo obiettivo della ricerca, concludendo che può essere utile considerare la temperatura come un predittore per il livello di clorofilla: a parità di altre condizioni, se in questo mese la temperatura aumenta, nel prossimo mese ci aspettiamo una diminuzione del livello della clorofilla.



Il mondo di *ggplot*

In genere, gli ambienti di calcolo che si basano su un'interfaccia a riga di comando (come *R*) non sono la categoria di strumenti che sceglieremmo per fare “un bel grafico” da mettere in un report, in una presentazione o in una pubblicazione scientifica. Molto di noi sono abituati ad applicativi come Microsoft Excel, Strata, Statistica, e tanti altri che ci offrono un'ampia varietà di grafici “preconfezionati” che possiamo facilmente (?) usare per plottare i nostri dati, divertendoci a cambiare dettagli come i colori e le scritte.

Ovviamente, essendo *R* un ambiente di elaborazione e programmazione, le nostre possibilità sono molto più ampie di quelle mediamente fornite da software tradizionali per la statistica. Abbiamo già visto nella sezione *Le basi* di questo testo quali sono le principali tipologie di *plot* che è possibile fare con una installazione *base* di *R*. Il mondo di *ggplot2* rappresenta la più significativa integrazione delle funzioni grafiche di base ed è diventato uno strumento di riferimento per tantissimi utenti, tanto che ormai dire “lo faccio con *ggplot*” indica una scelta ben precisa e facilmente riconoscibile.

Vale la pena spendere due parole sull'autore e sviluppatore del pacchetto `ggplot2`, che si chiama *Hadley Wickham*. Hadley è *Chief Scientist* di *RStudio* e professore aggiunto di statistica presso l'Università di Auckland, la Stanford University e la Rice University... ma soprattutto è diventato uno dei *guru* di *R* poichè, come autore di pacchetti quali `ggplot2`, `plyr`, `dplyr`, `tidr` e tanti altri, ha impresso una direzione ben precisa a una parte del mondo *R*. Secondo Hadley Wickham, rappresentare i dati e l'informazione contenuta al loro interno è una vera e propria **arte**... e comporre un grafico equivale a scrivere una poesia.

Come sintetizzare cosa è e a cosa serve l'approccio di `ggplot2`? Nel 2012 lo stesso Wickam affermava: «*L'enfasi di ggplot2 è sulla riduzione del tempo di riflessione, rendendo più facile il passaggio dal grafico nel cervello al grafico sulla pagina*». Questa frase si concentra sull'aspetto immediatezza/semplicità del flusso di creazione che da una serie di dati ci porta a un grafico informativo e chiaro. D'altra parte, la stessa frase non rende giustizia alla incredibile molteplicità di soluzioni grafiche che sono nate da e intorno a `ggplot2`.

Una panoramica dei principali tipi di grafico che è possibile ottenere è riportata nella figura seguente, ottenuta da <https://r-graph-gallery.com/>.



Figura 126 - Le famiglie di grafici che si possono (facilmente) realizzare con *ggplot2*.

Per comprendere a fondo la filosofia di *ggplot2*, comunque, bisogna fare un passo indietro e parlare brevemente di **Grammar of Graphics** o *Grammatica dei grafici*. **Una Grammatica dei grafici è uno strumento linguistico che ci permette di descrivere in modo conciso i componenti di un grafico che abbiamo nella testa e vorremmo realizzare a partire dai nostri dati.** La *Grammar of Graphics* è stata sviluppata, prima di Wickham, da Wilkinson, Anand e Grossman (2005), che descrivono anche le sue estensioni e i perfezionamenti. La Grammatica dei grafici

può essere pensata come una lingua per comunicare più facilmente con R (un dialetto, in un certo senso) per ottenere i nostri grafici. Ma in che senso possiamo comunicare più facilmente? Nei comandi di base per i plot di R abbiamo visto che le funzioni principali (es. `plot`, `lines`, `points`, `par`) sono dei mattoni grossolani che ci permettono di arrivare lontano ma con una certa fatica. Nella Grammatica dei grafici, invece, abbiamo a disposizione funzioni più evolute e simili al linguaggio umano. Il risultato, in poche parole, è che usando `ggplot2` serve molto meno tempo e codice per ottenere (più o meno) lo stesso grafico che otterremo con R base.

Per convincere il lettore, abbiamo adattato un esempio disponibile *qui*. I dati che useremo sono presi da *qui* e rappresentano un sottoinsieme (anni 2013-2021) dei dati spaziali forniti dagli Stati membri dell'UE nell'ambito della richiesta di dati DCF (*Data Collection Framework*) per la raccolta di informazioni dipendenti dalla pesca (IDE). In particolare, si tratta dell'origine (in mare) delle biomasse sbarcate per le 5 specie più importanti nell'area FAO 37, cioè il Mar Mediterraneo e il Mar Nero. Si tratta di dati pubblici e ufficiali che documentano un'attività che ha una grande importanza per l'alimentazione umana ma anche per le condizioni dei nostri mari, visto che la pesca è lo sfruttamento di risorse naturali a fini commerciali.

Le specie in questione sono (in ordine decrescente di quantità sbarcate):

- La sardina (*Sardina pilchardus*), il cui codice FAO è *ANE*
- L'acciuga (*Engraulis encrasicolus*), il cui codice FAO è *PIL*
- Il gambero rosa (*Parapenaeus longirostris*), il cui codice FAO è *DPS*
- Il gambero rosso (*Aristaeomorpha foliacea*), il cui codice FAO è *ARS*
- Il nasello (*Merluccius merluccius*), il cui codice FAO è *HKE*

Le biomasse (in Kg) sbarcate ogni anno sono ripostate per ciascuna delle aree (denominate *Geographical Sub Areas*) in cui la FAO divide **il Mar Mediterraneo e il Mar Nero**

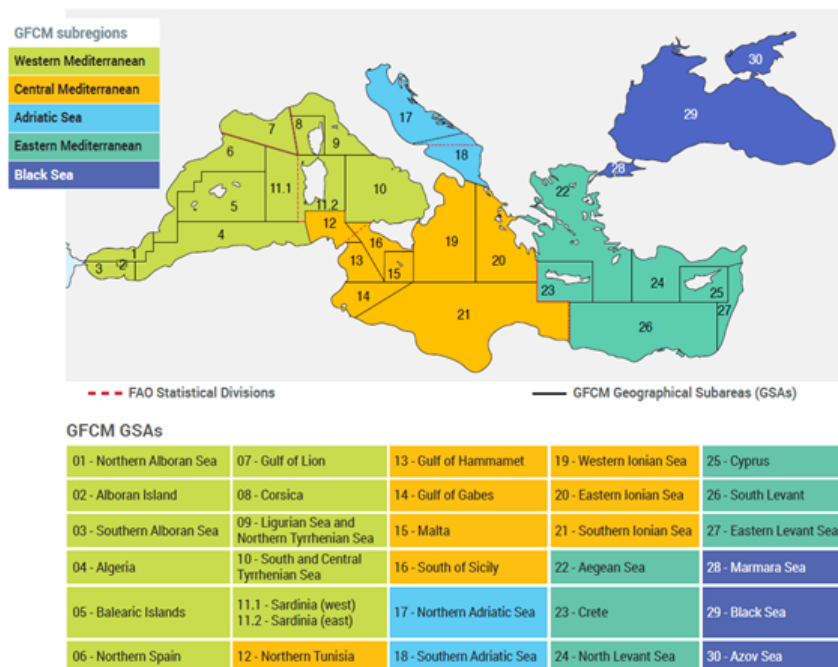


Figura 127 - La divisione FAO del Mar Mediterraneo e Mar Nero in Geographical Sub Areas (GSA).

```
# Load the FDI data
FDI =
read.table("https://raw.githubusercontent.com/tommaso-russo/LibroR2/
main/FDI/Sample_FDI.txt",
          sep = ";", dec = ".", header = T)
dim(FDI)
## [1] 251 5
head(FDI)
##   year          region species fishing_technique totwghtlandg
## 1 2016      Adriatic Sea    ANE                DTS      0.21200
## 2 2017      Adriatic Sea    ANE                DTS      2.20300
## 3 2018      Adriatic Sea    ANE                DTS      0.27800
## 4 2019      Adriatic Sea    ANE                DTS     24.19000
## 5 2020      Adriatic Sea    ANE                DTS      0.02500
## 6 2014 Central Mediterranean ANE                DTS      5.78273
```

Il data frame contiene le seguenti colonne (variabili):

- **year**: l'anno di rilevamento;

- **region**: la regione di origine degli sbarcati;
- **species**: la specie, identificata mediante il codice FAO di tre lettere;
- **fishing_technique**: il metodo di pesca utilizzato;
- **totwghtlandg**: il valore degli sbarcati in Tonnellate.

Adesso lanciamo la nostra sfida tra R base e ggplot2. Vogliamo un grafico in cui:

- ci devono essere tanti pannelli (di dimensioni identiche) quante sono le aree (in righe) e le tecniche di pesca (in colonne);
- i titoli e i sottotitoli dei pannelli devono corrispondere alle aree e alle tecniche di pesca, rispettivamente;
- ogni pannello deve mostrare i valori di **totwghtlandg** contro quelli di **year** (in pratica l'andamento nel tempo dello sbarcato in peso) mediante righe che abbiano un colore diverso (ma coerente tra i grafici) a seconda della specie considerata;
- la scala dell'asse y deve essere in $\log_{10}$;
- le scale dei due grafici devono essere identiche per consentire un confronto tra i diversi casi;
- ci deve essere una legenda per le specie.

Adesso diamo un'occhiata al codice qui sotto, che ci consente di ottenere, con i comandi base di R, il grafico desiderato:

```
par(mar = c(1.5, 5, 3, 1.5),
    mfrow = c(length(unique(FDI$region)),
               length(unique(FDI$fishing_technique))))
ylim = ceiling(log10(range(FDI$totwghtlandg)))
species = unique(as.character(FDI$species))
colors = 1:5
for(region in unique(FDI$region)) {
  for(fishing_technique in unique(FDI$fishing_technique)){
    sub_data = FDI[which((FDI$region == region) &
                        (FDI$fishing_technique ==
                         fishing_technique)),]
    plot(x = sub_data$year[which(sub_data$species == species[1])],
         y = log10(sub_data$totwghtlandg)[which(sub_data$species == species[1])],
         type = "l",
         col = 1,
```

```

    main = paste0(as.character(region), "\n",
                  as.character(fishing_technique)),
    ylim = ylim,
    ylab = "totwghtlandg",
    xlab = "year",
    yaxt='n')
  for(ispecies in 2:5){
    lines(x = sub_data$year[which(sub_data$species == species[ispecies])],
          y = log10(sub_data$totwghtlandg[which(sub_data$species ==
species[ispecies])]),
          type = "l",
          col = ispecies)
  }
  legend("topleft",
        species,
        col = colors,
        horiz = F,
        lwd = 2,
        cex = 0.7,
        y.intersp = 0.4)
  axis(2,
        at = seq(min(ylim),
                  max(ylim), 1)[seq(2, 8, 2)]),
        labels = paste(10^(seq(min(ylim),
                              max(ylim), 1)[seq(2, 8, 2)])))
  title(ylab = "totwghtlandg")

  for(z in 1:length(unique(FDI$species))){
    sub_data_species = subset(sub_data,
                              species == unique(FDI$species)[z])
    lines(x = sub_data_species$year,
          y = log10(sub_data_species$totwghtlandg),
          col = as.numeric(sub_data_species$species), lwd = 2)
  }
}
}

```

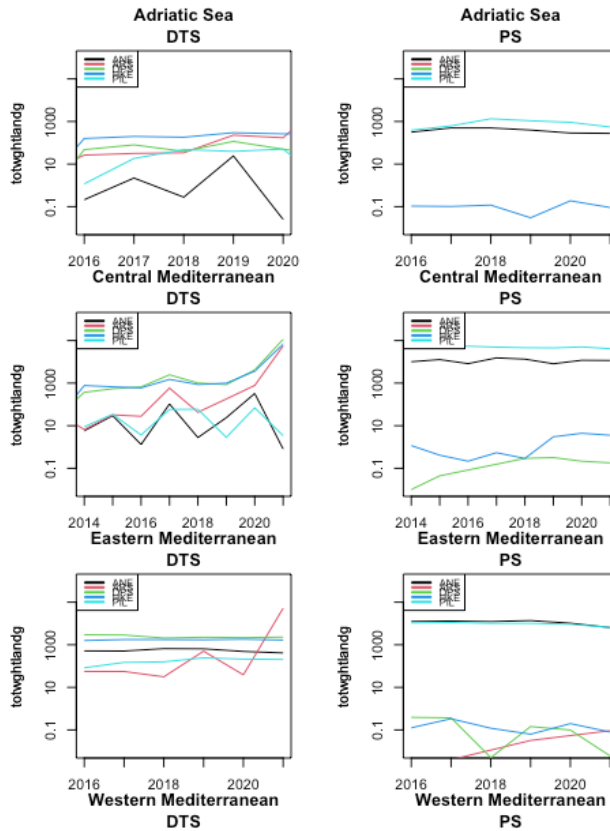


Figura 128 - Grafico ottenuto utilizzando i comandi base di R.

Il codice qui sopra comprende 51 righe. Abbiamo usato 3 cicli `for` annidati (si poteva fare in altri modi, ma questo è uno dei più “veloci”) in modo da essere sicuri che alcune caratteristiche dei pannelli fossero identiche (scale e limiti degli assi, colori, stili), e abbiamo ottenuto un risultato a malapena presentabile, ma certo non proprio bello. Per esempio la stessa legenda ripetuta per ogni pannello non è il massimo dell’eleganza, e l’aspetto complessivo lascia a desiderare.

Adesso proviamo con `ggplot2`:

```
library(ggplot2)
ggplot(FDI, aes(x = year,
                y = totwghtlandg,
                color = species)) +
```

```
geom_line(size = 1.2) +
scale_x_continuous(breaks = seq(min(FDI$year),
                                max(FDI$year), 1)) +
scale_y_continuous(trans = "log10") +
facet_wrap(~ region + fishing_technique,
           ncol = 2) +
theme(legend.position = "bottom")
```

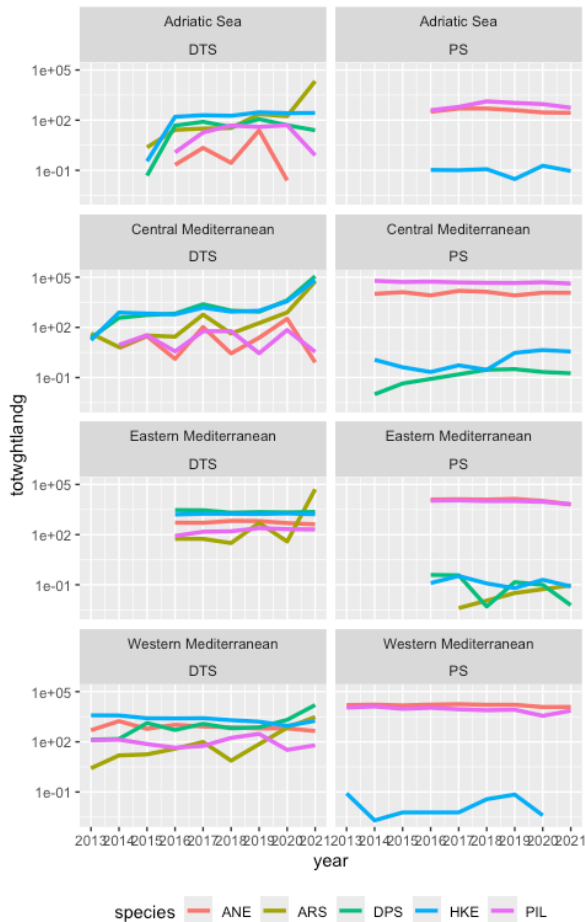


Figura 129 - Grafico ottenuto utilizzando ggplot2.

In questo secondo caso sono bastate 6 righe di codice (più o meno $\frac{1}{10}$ della versione base). L'aspetto del secondo grafico è, però, molto più gradevole del primo: la legenda non è replicata, le etichette e lo stile dei

pannelli sono più eleganti, e ci sono le griglie che sono comode per fare i confronti. Ma l'aspetto più interessante emerge se ci mettiamo a guardare i due codici: il primo è contorto, poco trasparente e intuitivo, e ci vuole un po' per capire come funziona. Nel secondo caso, anche senza sapere nulla di `ggplot2`, ogni riga corrisponde a una funzione ed è intuitivo capire i vari passaggi. E questo è esattamente quello che ci accingiamo a fare adesso.

Un grafico `ggplot2` è dunque costituito da alcuni elementi di base:

- **data**: i dati che si desidera utilizzare. È bene che questi siano un data frame che contiene tutti i campi (spesso in formato **long**) ai quali si farà riferimento per tracciare il grafico. Quasi sempre **data** viene specificato come argomento della funzione `ggplot` (generica) che inizializza qualsiasi grafico in `ggplot2`;
- **geom_**: le forme geometriche che rappresenteranno i dati nel grafico che abbiamo in mente (es. punti, linee, barre, torte, ecc.). Questo è un aspetto cruciale in cui la *Grammatica dei Grafici* trova la sua applicazione: mediante funzioni dal nome molto intuitivo (es. `geom_point()`, `geom_line()`, `geom_bar()` o `geom_histogram()`) dovremo dire ad R come vogliamo che sia fatto il grafico;
- **aes()**: l'estetica degli oggetti geometrici e statistici, come colore, dimensione, forma e posizione;
- **scale_**: le trasformazioni che occorre applicare per passare dai dati grezzi ai dati del grafico, come ad esempio una conversione logaritmica per le coordinate o le scale di colori degli oggetti.

Questi quattro ingredienti sono tutto quello che ci occorre per la nostra ricetta grafica. Impareremo semplicemente che ci sono tante funzioni **geom_** e **scale_** quanti sono gli aspetti da personalizzare. L'uso di **data** e **aes()** è, invece, abbastanza standard: una volta definiti per il nostro grafico possiamo non toccarli più.

Coerentemente con la **Grammatica dei Grafici**, gli ingredienti che abbiamo appena elencato corrispondono ad aspetti concettualmente (e praticamente) diversi di un grafico. Per illustrare questi aspetti usiamo i dati FDI introdotti pocanzi.

```
str(FDI)
## 'data.frame':   251 obs. of  5 variables:
##  $ year          : int  2016 2017 2018 2019 2020 2014 2015
```

```

2016 2017 2018 ...
## $ region      : chr  "Adriatic Sea" "Adriatic Sea"
"Adriatic Sea" "Adriatic Sea" ...
## $ species     : chr  "ANE" "ANE" "ANE" "ANE" ...
## $ fishing_technique: chr  "DTS" "DTS" "DTS" "DTS" ...
## $ totwghtlandg : num  0.212 2.203 0.278 24.19 0.025 ...

```

L'analisi della struttura del data frame mostra che il campo *year* è un intero, mentre *region*, *species* sono fattori ordinati, ognuno avente 5 livelli, e *fishing_technique* è un *character*. Infine, *totwghtlandg* contiene valori numerici continui.

Quanti "tipi" di grafico si possono concettualmente definire da questo data frame, considerando anche quelli un po' assurdi? L'elenco che segue riguarda *SOLO* i grafici che si potrebbero ottenere da ognuna delle colonne del data frame presa singolarmente:

- un istogramma dei valori nella colonna *year*;
- un istogramma dei valori nella colonna *totwghtlandg*;
- un grafico a barre dei valori nella colonna *region*;
- un grafico a barre dei valori nella colonna *species*;
- un grafico a barre dei valori nella colonna *fishing_technique*;

Questo tanto per cominciare. Poi ci sono i grafici che potremmo ottenere combinando le colonne a coppie:

- un istogramma dei valori di *totwghtlandg* in base a *year*;
- un istogramma dei valori *totwghtlandg* in base a *region*;
- un grafico a barre dei valori *totwghtlandg* in base a *species*;
- un grafico a barre dei valori *totwghtlandg* in base a *fishing_technique*.

Concentriamoci sul primo elenco (colonne singole come istogrammi). La sintassi minima di *ggplot* per ottenere ognuno dei grafici è:

```

library(ggplot2)
ggplot(data = FDI,
       aes(x = year)) +
  geom_histogram() +
  theme_test() +
  scale_x_continuous(breaks = as.numeric(levels(factor(FDI$year))))

```

```
## `stat_bin()` using `bins = 30`. Pick better value with  
`binwidth`.
```

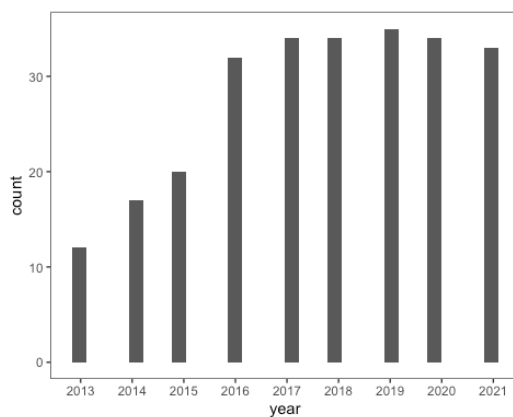


Figura 130 - Istogramma dei valori nella colonna year.

```
ggplot(data = FDI,  
       aes(x = totwghtlandg)) +  
  geom_histogram() +  
  theme_test()  
## `stat_bin()` using `bins = 30`. Pick better value with  
`binwidth`.
```

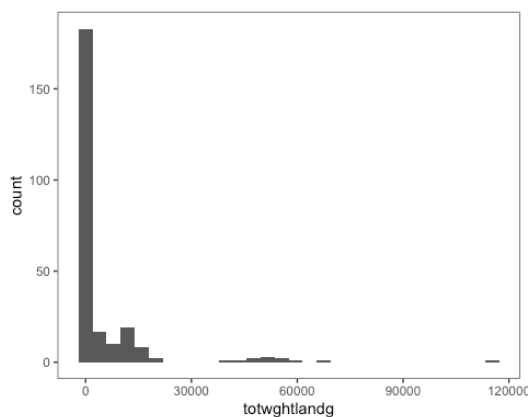


Figura 131 - Istogramma dei valori nella colonna totwghtlandg.

```
ggplot(data = FDI,
       aes(x = region)) +
  geom_bar() +
  theme_test() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

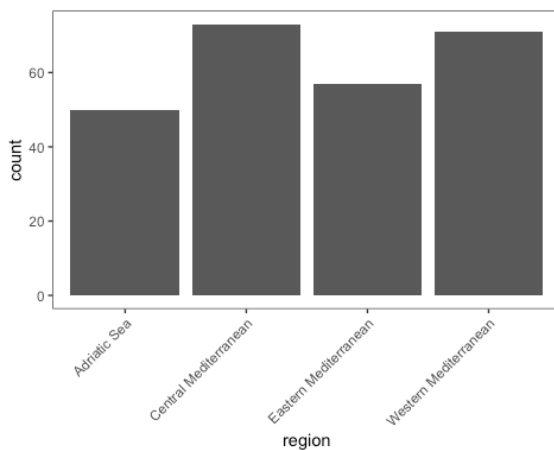


Figura 132 - Grafico a barre dei valori nella colonna region.

```
ggplot(data = FDI,
       aes(x = species)) +
  geom_bar() +
  theme_test()
```

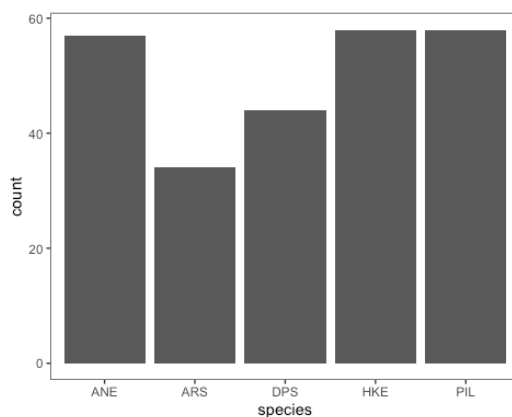


Figura 133 - Grafico a barre dei valori nella colonna species.

Figura 135 - Grafici che è possibile realizzare con ggplot. Fonte: <https://stulp.gmw.rug.nl/ggplotworkshop/ggplotgeoms.html>.

Graphical Primitives

- a** <- ggplot(economics, aes(date, unemploy))
b <- ggplot(seals, aes(x = long, y = lat))
- a** + geom_blank()
 (Useful for expanding limits)
- b** + geom_curve(aes(yend = lat + 1, xend = long + 1, curvature = z)) - x, xend, y, yend, alpha, angle, color, curvature, linetype, size
- a** + geom_path(lineend = "butt", linejoin = "round", linemitre = 1)
 x, y, alpha, color, group, linetype, size
- a** + geom_polygon(aes(group = group))
 x, y, alpha, color, fill, group, linetype, size
- b** + geom_rect(aes(xmin = long, ymin = lat, xmax = long + 1, ymax = lat + 1)) - xmax, xmin, ymax, ymin, alpha, color, fill, linetype, size
- a** + geom_ribbon(aes(ymin = unemploy - 900, ymax = unemploy + 900)) - x, ymax, ymin, alpha, color, fill, group, linetype, size

Line Segments

common aesthetics: x, y, alpha, color, linetype, size

- b** + geom_abline(aes(intercept = 0, slope = 1))
b + geom_hline(aes(yintercept = lat))
b + geom_vline(aes(xintercept = long))
b + geom_segment(aes(yend = lat + 1, xend = long + 1))
b + geom_spoke(aes(angle = 1:1155, radius = 1))

One Variable

Continuous

c <- ggplot(mpg, aes(hwy)); **c2** <- ggplot(mpg)

- c** + geom_area(stat = "bin")
 x, y, alpha, color, fill, linetype, size
- c** + geom_density(kernel = "gaussian")
 x, y, alpha, color, fill, group, linetype, size, weight
- c** + geom_dotplot()
 x, y, alpha, color, fill
- c** + geom_freqpoly()
 x, y, alpha, color, group, linetype, size
- c** + geom_histogram(binwidth = 5)
 x, y, alpha, color, fill, linetype, size, weight
- c2** + geom_qq(aes(sample = hwy))
 x, y, alpha, color, fill, linetype, size, weight

Discrete

d <- ggplot(mpg, aes(fl))

- d** + geom_bar()
 x, alpha, color, fill, linetype, size, weight

Two Variables

Continuous X, Continuous Y
e <- ggplot(mpg, aes(cty, hwy))

e + geom_label(aes(label = cty), nudge_x = 1, nudge_y = 1, check_overlap = TRUE)

x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust



e + geom_jitter(height = 2, width = 2)

x, y, alpha, color, fill, shape, size



e + geom_point()

x, y, alpha, color, fill, shape, size, stroke



e + geom_quantile()

x, y, alpha, color, group, linetype, size, weight



e + geom_rug(sides = "bl")

x, y, alpha, color, linetype, size



e + geom_smooth(method = lm)

x, y, alpha, color, fill, group, linetype, size, weight



e + geom_text(aes(label = cty), nudge_x = 1, nudge_y = 1, check_overlap = TRUE)

x, y, label, alpha, angle, color, family, fontface, hjust, lineheight, size, vjust

Discrete X, Continuous Y

f <- ggplot(mpg, aes(class, hwy))



f + geom_col()

x, y, alpha, color, fill, group, linetype, size



f + geom_boxplot()

x, y, lower, middle, upper, ymax, ymin, alpha, color, fill, group, linetype, shape, size, weight



f + geom_dotplot(binaxis = "y", stackdir = "center")

x, y, alpha, color, fill, group



f + geom_violin(scale = "area")

x, y, alpha, color, fill, group, linetype, size, weight

Discrete X, Discrete Y

g <- ggplot(diamonds, aes(cut, color))



g + geom_count()

x, y, alpha, color, fill, shape, size, stroke

Continuous Bivariate Distribution
h <- ggplot(diamonds, aes(carat, price))

h + geom_bin2d(binwidth = c(0.25, 500))

x, y, alpha, color, fill, linetype, size, weight



h + geom_density2d()

x, y, alpha, colour, group, linetype, size



h + geom_hex()

x, y, alpha, colour, fill, size

Continuous Function

i <- ggplot(economics, aes(date, unemployment))



i + geom_area()

x, y, alpha, color, fill, linetype, size



i + geom_line()

x, y, alpha, color, group, linetype, size



i + geom_step(direction = "hv")

x, y, alpha, color, group, linetype, size

Visualizing error

df <- data.frame(grp = c("A", "B"), fit = 4:5, se = 1:2)

j <- ggplot(df, aes(grp, fit, ymin = fit-se, ymax = fit+se))



j + geom_crossbar(fatten = 2)

x, y, ymax, ymin, alpha, color, fill, group, linetype, size



j + geom_errorbar()

x, ymax, ymin, alpha, color, group, linetype, size, width (also **geom_errorbarh**())



j + geom_linerange()

x, ymin, ymax, alpha, color, group, linetype, size



j + geom_pointrange()

x, y, ymin, ymax, alpha, color, fill, group, linetype, shape, size

Maps

data <- data.frame(murder = USArrests\$Murder, state = tolower(rownames(USArrests)))

map <- map_data("state")

k <- ggplot(data, aes(fill = murder))



k + geom_map(aes(map_id = state), map = map) + **expand_limits**(x = map\$long, y = map\$lat)

map_id, alpha, color, fill, linetype, size

Three Variables

seals\$z <- with(seals, sqrt(delta_long^2 + delta_lat^2))

l <- ggplot(seals, aes(long, lat))



l + geom_contour(aes(z = z))

x, y, z, alpha, colour, group, linetype, size, weight



l + geom_raster(aes(fill = z), hjust=0.5, vjust=0.5, interpolate=FALSE)

x, y, alpha, fill



l + geom_tile(aes(fill = z))

x, y, alpha, color, fill, linetype, size, width

```
ggplot(data = FDI,  
       aes(x = fishing_technique)) +  
geom_bar()
```

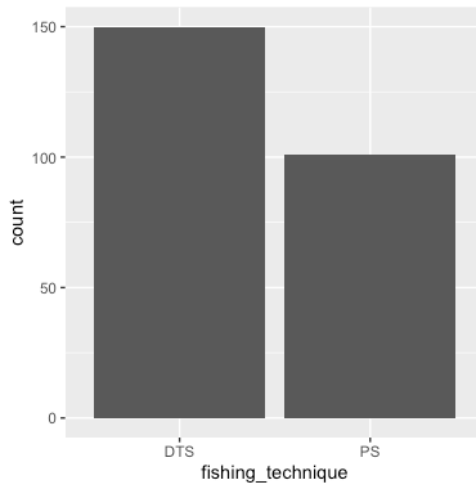


Figura 134 - Grafico a barre dei valori nella colonna fishing_technique.

Notiamo che:

- in tutti i casi, si comincia con la funzione **ggplot** che specifica due argomenti: il primo (**data**) è il data frame su cui lavorare, il secondo è la funzione **aes** che contiene, tutte le volte, l'argomento **x** (l'ascissa del grafico) con indicazioni diverse a seconda dalle variabili a cui siamo interessati;
- la funzione **ggplot** è seguita da un **+** e poi da un'altra funzione dal nome molto intuitivo: **geom_histogram()** nel caso degli istogrammi e **geom_bar()** nel caso delle barre (la differenza tra i due tipi di grafico è che l'istogramma è definito per dati numerici e il grafico a barre si usa per variabili discrete o fattori).

In sostanza abbiamo usato due comandi e tre funzioni:

- il primo comando (**ggplot**) introduce il grafico, specifica i dati (**data**) da usare (cioè il data frame **FDI**) ed indica quale variabile mappare e su quale asse (**aes(x = ...)**)
- il secondo comando (**geom_histogram()** o **geom_bar()** o qualcosa) serve a dire che tipo di grafico vogliamo (usando parole colloquiali dal significato molto intuitivo: **point**, **line**, **polygon**, ecc.)

Se abbiamo capito questo, possiamo immaginare di usare i dati FDI per applicare tante tra le **decine** di geometrie che sono definite in *ggplot* (si chiamano **primitive geometriche** per l'esattezza). Possiamo trovare una lista di primitive geometriche e di altri tipi di grafici che è possibile realizzare con *ggplot*.

Vediamone alcune per capire quanto è semplice applicarle, combinando insieme diverse variabili del data frame FDI.

```
library(ggplot2)
ggplot(data = subset(FDI,
                     species == "PIL" & fishing_technique == "PS" &
                     region == "Adriatic Sea"),
       aes(x = year,
           y = totwghtlandg)) +
  geom_line() +
  theme_test()
```

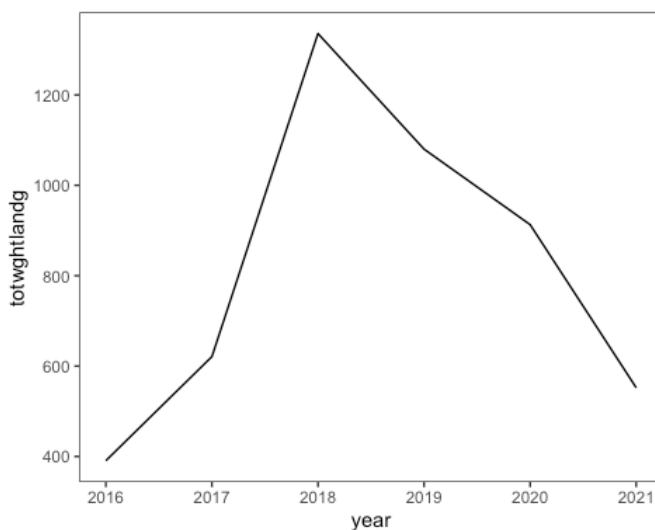


Figura 136 - Andamento annuale del peso delle catture di sardina (PIL) effettuate nell'Adriatico con reti a circuizione (PS).

Nel grafico abbiamo rappresentato l'andamento della variabile *totwghtlandg* (peso totale degli sbarcati) di una specie (la Sardina,

Sardina pilchardus) pescata con la circuizione (PS) nel Mar Adriatico. Dopo aver specificato il data frame di riferimento (`data = subset(FDI, ....)`) è bastato dire che sulle x volevamo gli anni e sulle y gli sbarcati. Infine, abbiamo usato la funzione `geom_line()` senza bisogno di aggiungere altro.

Adesso, a partire da questo grafico, facciamo conoscenza con un'altra geometria: `geom_smooth()`, che ci permette di stimare la relazione tra i dati rappresentati nel grafico (ad esempio `year` e `totwghtlandg`):

```
library(ggplot2)
ggplot(data = subset(FDI,
                      species == "PIL" &
                      fishing_technique == "PS" &
                      region == "Adriatic Sea"),
       aes(x = year,
           y = totwghtlandg)) +
  geom_line() +
  geom_smooth(method = "lm")
## `geom_smooth()` using formula = 'y ~ x'
```

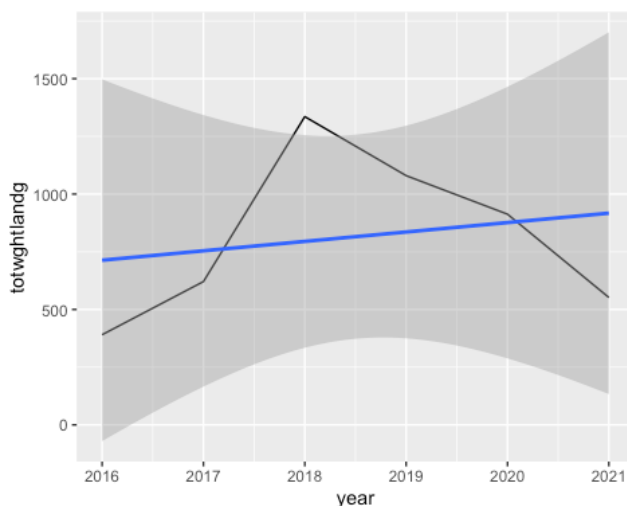


Figura 137 - Andamento annuale del peso delle catture di sardina (PIL) effettuate nell'Adriatico con reti a circuizione (PS) e sovrapposizione con modello lineare fittato sui dati.

In questo caso abbiamo specificato che il metodo deve essere `lm`, che come sappiamo già sta per *linear model*, ma avremo potuto scegliere tra: `lm`, `glm`, `gam`, `loess` e anche funzioni definite dall'utente (ovviamente ammesso che valga la pena farlo e che i nostri dati siano adeguati).

Una cosa notevole, e che ancora non abbiamo sottolineato, è che tutto ruota intorno alla funzione `aes` (aesthetics), che ci permette di legare ogni elemento del grafico (forme, colori, dimensioni) a una o più variabili. Vediamo un esempio sulla base del grafico precedente, che era limitato alla *species* PIL:

```
library(ggplot2)
ggplot(data = subset(FDI,
                     fishing_technique == "PS" & region == "Adriatic
Sea"),
       aes(x = year,
           y = totwghtlandg,
           colour = species)) +
  geom_line() +
  theme_test()
```

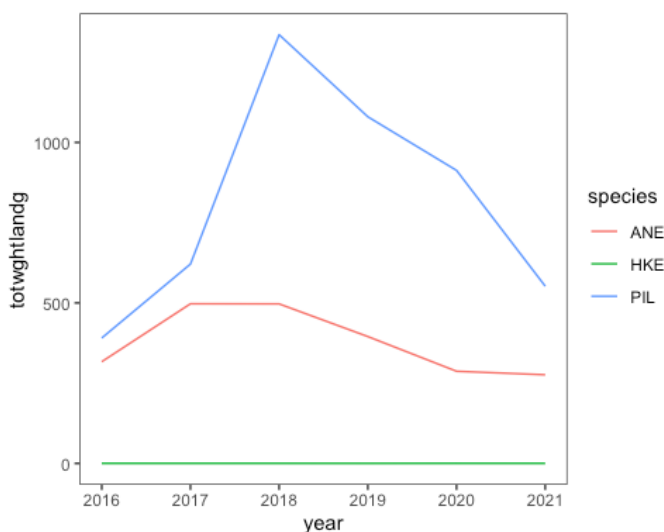


Figura 138 - Andamento annuale del peso delle catture effettuate nell'Adriatico con reti a circuizione (PS) per ogni specie.

Facciamo caso al fatto che, specificando `colour = species` tra gli argomenti di `aes`, abbiamo ottenuto tre linee distinte, una per ogni specie. Inoltre `ggplot` ha inserito una legenda. Questa volta, infatti, non abbiamo usato come argomento di `ggplot` il `subset` del data frame FDI ristretto a PIL.

Oltre al colore, comunque, possiamo lavorare sulla forma delle linee per le diverse specie o, meglio ancora, usare i colori per le specie e linee diverse per le tecniche di pesca.

```
library(ggplot2)
ggplot(data = subset(FDI,
                     region == "Adriatic Sea"),
       aes(x = year,
           y = totwghtlandg,
           colour = species,
           linetype = fishing_technique)) +
  geom_line() +
  theme_test()
```

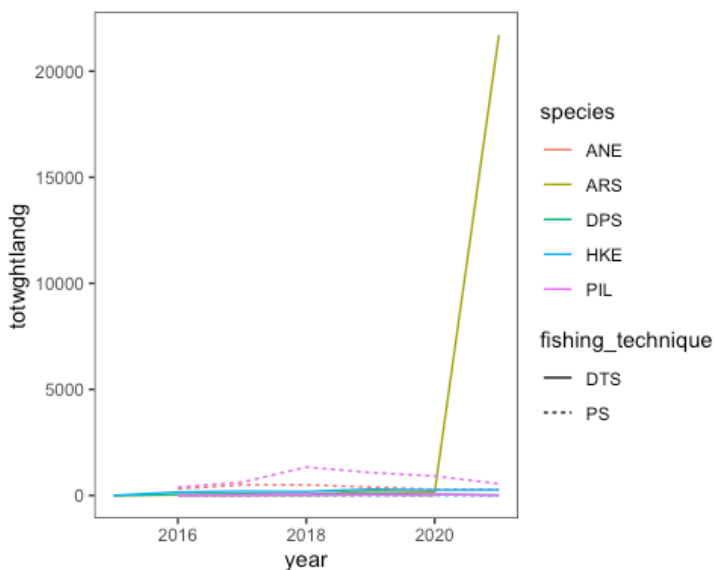


Figura 139 - Andamento annuale del peso delle catture nell'Adriatico per specie e tecnica di pesca.

Dalla legenda si nota che è comparsa una seconda “chiave”, quella relativo appunto al tipo di linea, che varia in base alla tecnica di pesca. Il grafico ottenuto non è molto chiaro, perché i valori di una specie sono chiaramente fuori scala rispetto agli altri. Una delle soluzioni per correggere questo genere di problemi ci fornisce lo spunto per aggiungere un nuovo ingrediente: *scale*. Attraverso questa funzione possiamo fare diverse cose, come ad esempio applicare delle trasformazioni matematiche agli assi dei dati senza toccare il data frame di partenza.

```
library(ggplot2)
ggplot(data = subset(FDI,
                     region == "Adriatic Sea"),
       aes(x = year,
           y = totwghtlandg,
           colour = species,
           linetype = fishing_technique)) +
  geom_line() +
  scale_y_continuous(trans = "log10") +
  theme_test()
```

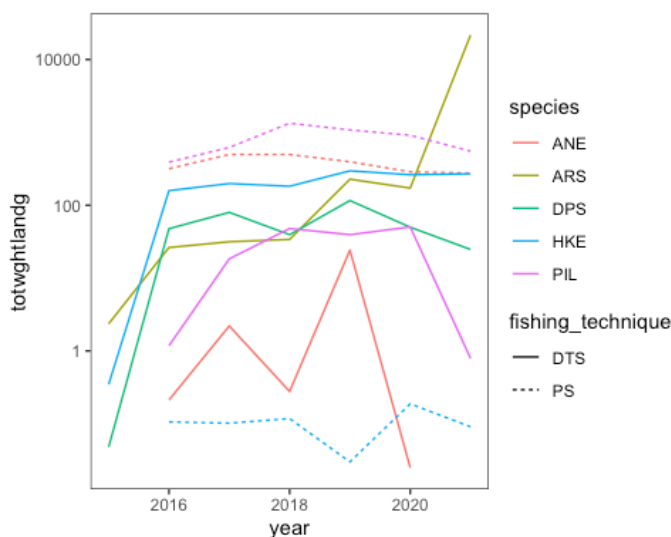


Figura 140 - Andamento annuale in scala logaritmica del peso delle catture nell'Adriatico per specie e tecnica di pesca.

Ecco quindi che, usando la funzione `scale_y_continuous` e specificando che la trasformazione che vogliamo è il logaritmo in base 10, abbiamo ottenuto il risultato desiderato: un grafico molto più chiaro. Quella delle scale, comunque, è una vera e propria famiglia. A seconda dell'asse a cui ci riferiamo e del tipo di dato (discreto o continuo) rappresentato potremmo usare:

- `scale_x...`
- `scale_y...`
- `scale_fill...`
- `scale_colour...`

Ho usato i puntini perché le funzioni in questione sono circa **100!!!** E se qualcuno potrebbe obiettare che è impossibile memorizzarle tutte, la risposta sarebbe che non serve: a seconda di quello che ci serve e di cosa vogliamo fare, il nome della funzione che ci serve (e che magari non abbiamo mai usato) lo possiamo “intuire” senza grandi difficoltà. Provare per credere.

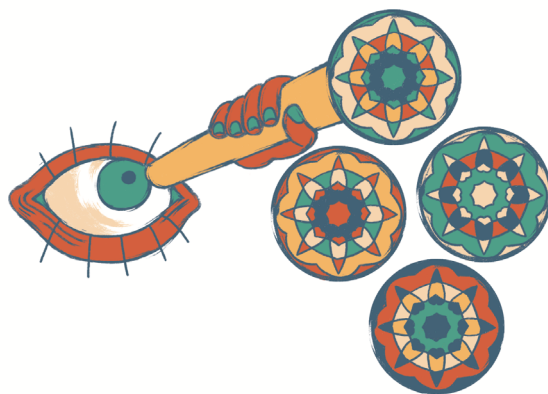


Figura 141 - Il metodo faceting è una specie di caleidoscopio per i nostri grafici.

Ma torniamo per un attimo all'ultimo grafico e al data frame FDI (con il quale ormai abbiamo una certa familiarità). Per introdurre l'ultimo ingrediente di cui parleremo, ritorniamo alla comparazione tra R base e `ggplot`, che abbiamo introdotto all'inizio del capitolo. Siamo in grado ora di decodificare (quasi) ogni singola istruzione:

```
library(ggplot2)
ggplot(FDI, aes(x = year,
                y = totwghtlandg,
                color = species)) +
  geom_path(size = 1.2) +
  scale_x_continuous(breaks = seq(min(FDI$year),
                                   max(FDI$year), 1)) +
  scale_y_continuous(trans = "log10") +
  facet_wrap(~ region + fishing_technique,
            ncol = 2) +
  theme_test() +
  theme(legend.position = "bottom")
```

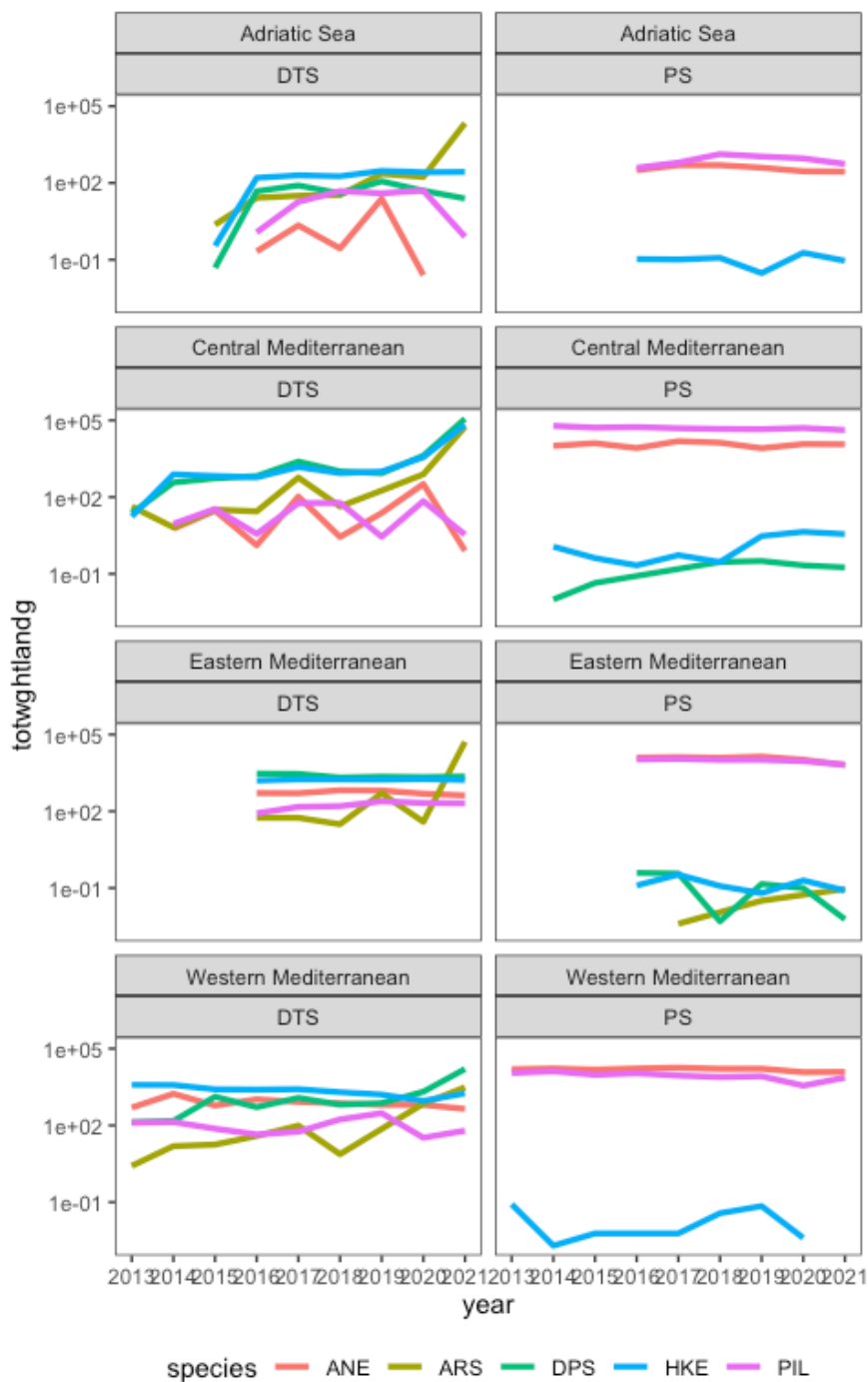
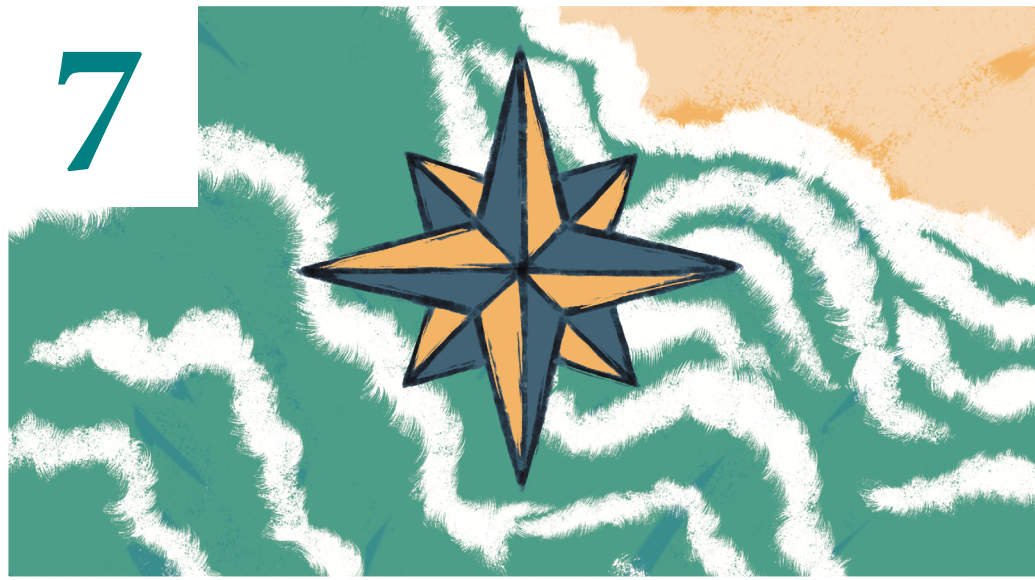


Figura 142 - Grafico realizzato utilizzando ggplot2 e la funzione facet_wrap().

In particolare:

- `ggplot(FDI, aes(x = year, y = totwghtlandg, color = species))` ci serve per inizializzare il grafico, definire i dati di *input* e le variabili coinvolte (collegandole agli assi e al colore degli elementi grafici);
- `geom_line(size = 1.2)` disegna gli andamenti (non ci vuole molto a capire che `size` indica lo spessore del tratto);
- `scale_x_continuous` viene usato per definire meglio il passo e gli estremi dei valori di riferimento usati per l'asse *x*;
- `scale_y_continuous` viene usato per applicare una scala logaritmica (in base 10) ai valori sull'asse delle *y*;
- `facet_wrap(~ region + fishing_technique, ncol = 2)` è una *new entry*, che non abbiamo ancora spiegato. In pratica, definito il grafico coi comandi precedenti, questa funzione effettua un "*faceting*", cioè genera un grafico a pannelli in cui ogni pannello è perfettamente coerente e omogeneo agli altri. Ogni pannello visualizza un sottoinsieme del data frame di partenza, definito (in questo caso) da tutte le possibili combinazioni di `region` e `fishing_technique`;
- `theme` è semplicemente una funzione "contenitore" che ci permette di personalizzare alcuni aspetti come la posizione della legenda.



Analisi di dati spaziali e cenni di geostatistica

Introduzione e considerazioni generali

«*Spazio, ultima frontiera!*», così diceva la sigla di Star Trek, ed in effetti l'analisi dei dati spaziali è l'ultimo capitolo di questo libro. Un tema affascinante ma complesso, che può avere tante declinazioni e che, però, può capitare di incontrare in tanti contesti della biologia e delle scienze ambientali. In questo capitolo impareremo a maneggiare i dati spaziali con *R* e faremo una piccola passeggiata nel mondo della geostatistica.

La **Geostatistica** è la scienza che studia i fenomeni (naturali e non) che si sviluppano su base spaziale a partire dalle informazioni derivanti da un loro campionamento. In particolare, la Geostatistica:

- studia la variabilità spaziale dei parametri che descrivono i suddetti fenomeni,
- estrae le “regole” (funzioni) in un quadro modellistico di riferimento,
- usa queste regole per dare soluzione a specifiche problematiche riguardanti la caratterizzazione e la stima dei fenomeni stessi.

I metodi della Geostatistica sono applicabili in tutti quei settori delle scienze applicate in cui i fenomeni di studio hanno carattere spaziale. In relazione alle applicazioni registrate negli ultimi tre decenni, tra i settori applicativi si possono citare:

- le scienze geologiche e minerarie, l'idrologia e l'idrogeologia,
- la scienza dei suoli e l'agronomia,
- la geotecnica e la geofisica
- il telerilevamento,
- la climatologia e la meteorologia,
- l'oceanografia, le scienze forestali, la zoologia e l'ecologia,
- l'epidemiologia e l'igiene ambientale.

La Geostatistica inizialmente è stata sviluppata per esigenze pratiche, lontano dall'ambito puramente statistico. Progressivamente è diventata una branca della statistica spaziale (spazio-temporale). Molti dei concetti geostatistici sono funzionali nell'ambito delle altre tipologie di analisi spaziale (e non solo... vedi serie temporali...). Oggetto delle analisi spaziali (in *R*) e della geostatistica sono, dunque, tutte le tipologie di dati e informazioni riferite a località definite nello spazio e nel tempo. Lo spazio è, quasi sempre, una porzione del nostro pianeta (e ancora più spesso una qualche località sulla sua superficie).

Dati geospaziali vettoriali

Questa prima tipologia è rappresentata da punti che possono rappresentare entità ben distinte ed indipendenti (ad esempio, le stazioni di campionamento nell'ambito di un'attività di raccolta dati sul campo) o possono essere collegati tra loro per formare geometrie più complesse come linee e poligoni. Nella maggiore parte dei casi si ha a che fare con punti individuati da solo due dimensioni (spesso denominate *x* e *y*), ma non è raro maneggiare dati in 3 dimensioni (nei quali si aggiunge una dimensione generalmente denominata *z* che corrisponde all'altezza sul livello del mare o la profondità).

I dati vettoriali geospaziali consistono in geometrie definite sulla base di coordinate geografiche. Ecco alcuni esempi di diversi tipi di geometrie:

- **geometria puntuale:** rappresenta un singolo punto sulla superficie terrestre. Ad esempio, la posizione di stazioni di campionamento, o individui di una determinata popolazione animale o vegetale, assumono solitamente questa forma;
- **geometria lineare:** linee, cioè punti collegati tra loro. Fiumi, batimetriche e isobate, linee di costa e percorsi di organismi possono essere rappresentati in questo modo;
- **poligoni:** insiemi di punti collegati tra loro per formare una forma chiusa, si tratta di un poligono. Qualsiasi area può essere rappresentata da un poligono spaziale: un lago, un'isola, un territorio, le celle di una griglia regolare.

Nella sua grande flessibilità, comunque, *R* consente di avere anche geometrie multiple costituite da oggetti multipli, anche appartenenti a diversi tipi di geometria. Ad esempio, un poligono multiplo può essere costituito da più di un poligono separato, utile quando si vuole rappresentare un'area non contigua come ad esempio i sub-areali di una metapopolazione.

Dati geospaziali raster

I dati raster hanno una struttura completamente differente rispetto ai dati vettoriali: essi sono “discreti”, nel senso che sono costituiti da unità fondamentali (generalmente denominati *pixel*), solitamente organizzati come una matrice di celle.

È utile spiegare che i dati raster hanno questa struttura perché, tipicamente, sono generati da sensori discreti. Cosa vuol dire? Pensate alla struttura del sensore di una fotocamera digitale oppure (è la stessa cosa) alla struttura della nostra retina: in entrambi i casi, ci sono un certo numero di unità discrete che vengono eccitate dalla radiazione luminosa, e traducono un'immagine in una serie di segnali discreti. Dunque, un ottimo esempio di dati geografici raster sono le immagini di telerilevamento. Naturalmente il dato raster si porta dietro un aspetto importante, cioè la risoluzione. Tutti sappiamo che uno schermo (del pc, della SmartTV o altro) è tanto migliore quanti più *pixel* (e più piccoli) contiene. Se invece i *pixel* sono pochi e grandi avremo immagini sgranate e poco definite.

A questo punto possiamo chiederci quando e perché potremmo trovarci a generare e usare uno tra queste due tipologie di dati spaziali. La risposta dipende probabilmente dall'ambito di applicazione:

- i dati vettoriali tendono a dominare le scienze sociali, perché gli insediamenti umani tendono ad avere confini discreti. Nella progettazione di infrastrutture (strade, edifici, ma anche parchi terrestri o marini) è utile e comodo usare poligoni regolari e confini netti;
- i dati raster dominano molte scienze ambientali, in parte a causa della dipendenza dai dati di telerilevamento. Se studiamo la produzione primaria in mare o la morfologia di un suolo lo faremo, molto probabilmente, partendo dai dati raccolti da un sensore ottico.

Da questo discende che, nel contesto dell'ecologia e delle scienze ambientali, spesso queste tipologie di dati sono utilizzate insieme (e per questo le abbiamo introdotte subito). Nell'ambiente *R* esiste un'ampia disponibilità di pacchetti per l'analisi dei dati spaziali, ma recentemente c'è stato un importante aggiornamento e cambio di paradigma che ha "mandato in soffitta" alcuni approcci e pacchetti per consolidare le analisi spaziali intorno a nuove (e più funzionali) procedure. Coghieremo questo momento di transizione per concentrare il nostro studio su due principali pacchetti: il pacchetto *sf* per l'analisi dei dati vettoriali e il pacchetto *terra* per quelli raster.

Il concetto di *Coordinates Reference System (CRS)*



Figura 143 - Il CRS definisce il nostro sistema di riferimento per definire le coordinate utilizzate.

Prima di affrontare gli argomenti di questo capitolo è necessaria una piccola digressione di geografia: dobbiamo rinfrescarci la memoria sul concetto di Sistema di Riferimento delle Coordinate (*Coordinates Reference System* o *CRS*). In geografia, un sistema di coordinate è un sistema di riferimento che consente di specificare ogni luogo sulla Terra mediante un insieme di numeri, lettere o simboli.

Ogni *CRS* è definito da:

- il suo quadro di misura, che può essere geografico (in cui le coordinate sferiche sono misurate dal centro della Terra) o proiettato, ad esempio planimetrico (in cui le coordinate terrestri sono proiettate su una superficie planare bidimensionale);
- le unità di misura (piedi/metri per i sistemi proiettati, gradi lon/lat per i sistemi geografici);
- per i sistemi di coordinate proiettate, una definizione della proiezione cartografica;
- altre proprietà del sistema di misura, come uno sferoide di riferimento, un *datum*, uno o più paralleli standard, un meridiano centrale ed eventuali spostamenti nelle direzioni *x* ed *y*.

In *R*, ci sono due modi per descrivere un *CRS*. Utilizzando il codice `epsg` o la definizione `proj4string`.

EPSG(SRID)

Lo *European Petroleum Survey Group Geodesy (EPSG)* è noto soprattutto per il suo sistema di ID di riferimento spaziale per i sistemi di coordinate. Un codice *EPSG* è un ID univoco che può essere un modo semplice per identificare un *CRS*. Un elenco dei codici *EPSG* è disponibile qui. Il codice *EPSG* potrebbe non essere sempre disponibile per un particolare sistema di coordinate, ma se un oggetto spaziale ha un sistema di coordinate definito, avrà sempre una stringa di proiezione *PROJ4*. La sua sintassi multiparametrica è discussa brevemente qui di seguito.

proj4string

PROJ è un software generico di trasformazione delle coordinate che trasforma le coordinate geospaziali da un sistema di riferimento di coordinate (CRS) a un altro. La sintassi di *PROJ4* consiste in un elenco di parametri utilizzati per definire un sistema di coordinate. I parametri vengono combinati in un'unica stringa dal carattere +, ed i più comuni sono:

- **+proj**: nome della proiezione;
- **+ellps**: nome dell'ellissoide;
- **+datum**: nome del *Datum*;
- **+units**: unità di misura (metri, piedi, ecc);
- **+x_0**: falso orientamento;
- **+lat_0**: latitudine di origine.

La libreria sf per i dati vettoriali

Questa libreria, il cui nome è l'acronimo di “*simple features*”, è stata sviluppata da Edzer Pebesma (un professore di geoinformatica tedesco che è uno dei guru del settore). Le “*simple features*” non sono altro che i tipi di geometrie che abbiamo elencato precedentemente per i dati vettoriali, e sono uno standard aperto sviluppato e approvato dall'*Open Geospatial Consortium* (OGC), un'organizzazione senza scopo di lucro. *Simple features* è un modello di dati gerarchico che rappresenta un'ampia gamma di tipi di geometria. Dei 18 tipi di geometria supportati dalla specifica, solo 7 sono utilizzati nella stragrande maggioranza delle ricerche geografiche.

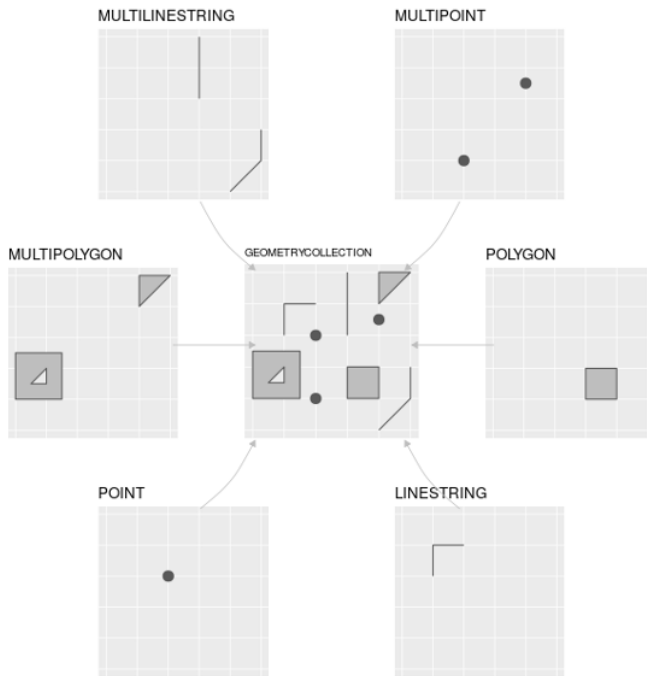


Figura 144 - Diverse classi di geometrie di elementi semplici. Fonte: Lovelace et al. (2020) (CC BY-NC-ND 4.0).

Le *simple features* sono uno standard formale (ISO 19125-1:2004) che descrive come gli oggetti del mondo reale possono essere rappresentati nei computer, con particolare attenzione alla geometria spaziale di questi oggetti. Inoltre, aspetto fondamentale, questo standard descrive come tali oggetti possano essere memorizzati e recuperati dai database e quali operazioni geometriche debbano essere definite per essi. Lo standard è ampiamente implementato nei database spaziali (come *PostGIS*), nei GIS commerciali (ad esempio, *ESRI ArcGIS*) e costituisce la base dei dati vettoriali per librerie come *GDAL*. Un sottoinsieme di caratteristiche semplici costituisce lo standard *GeoJSON*.

R dispone di classi ben supportate per la memorizzazione dei dati spaziali (*sp*) e per l'interfacciamento con gli ambienti sopra citati (*rgdal*, *rgeos*), ma finora mancava un'implementazione completa delle caratteristiche semplici, rendendo le conversioni a volte contorte, inefficienti o incomplete. Il pacchetto *sf* cerca di colmare questa lacuna e mira a superare *sp* nel lungo periodo.

`sf` può rappresentare tutti i tipi di geometria vettoriale più comuni (le classi di dati raster non sono supportate da `sf`): punti, linee, poligoni e le rispettive versioni “multi” (che raggruppano elementi dello stesso tipo in un unico elemento). `sf` supporta anche le collezioni di geometrie, che possono contenere più tipi di geometria in un singolo oggetto. `sf` fornisce le stesse funzionalità (e altre ancora) precedentemente fornite in tre pacchetti: `sp` per le classi di dati, `rgdal` per la lettura/scrittura dei dati tramite un'interfaccia con `GDAL` e `PROJ`, e `rgeos` per le operazioni spaziali tramite un'interfaccia con `GEOS`. La capacità di `sf` di integrare più librerie potenti per il *geocomputing* in un unico *framework* è un risultato notevole che riduce le “barriere all'ingresso” nel mondo dell'analisi dei dati geografici riproducibile con librerie ad alte prestazioni.

Vediamo dunque come sono fatte le *simple features*. Una *feature* è un oggetto del mondo reale, come un edificio o un albero. Come nel caso degli oggetti, che spesso sono costituiti da altri oggetti, questo vale anche per le caratteristiche: un insieme di caratteristiche può formare una singola caratteristica. Un popolamento forestale può essere una *feature*, una foresta può essere una *feature*, una città può essere una *feature*. Un *pixel* di un'immagine satellitare può essere un elemento, ma anche un'immagine completa può essere un elemento. Le *features* hanno una geometria che descrive la loro posizione sulla Terra e hanno attributi che descrivono altre proprietà. La geometria di un albero può essere la delimitazione della sua chioma, del suo fusto o del punto che indica il suo centro. Altre proprietà possono essere l'altezza, il colore, il diametro al seno in una determinata data e così via.



Figura 145 - Concettualmente (e praticamente) esistono diverse geometrie spaziali.

Come avevamo anticipato, tutte le geometrie sono composte da punti. I punti sono definiti da coordinate in uno spazio a 2 o 3 dimensioni (ne potrebbe esistere una quarta che ignoreremo). Tutti i punti di una geometria hanno la stessa dimensione. I 7 principali tipi di geometria sono quindi definiti come segue:

- **POINT**: geometria zero-dimensionale contenente un singolo punto.
- **LINESTRING**: sequenza di punti collegati da tratti di retta che non si intersecano tra loro, con geometria unidimensionale.
- **POLYGON**: geometria con area positiva (bidimensionale). Definisce una sequenza di punti formante un anello chiuso e non intersecante, dove il primo anello denota l'anello esterno, zero o più anelli successivi denotano buchi in questo anello esterno.
- **MULTIPOINT**: insieme di **POINT** in cui nessun punto è uguale all'altro.
- **MULTILINESTRING**: insieme di **LINESTRING**.
- **MULTIPOLYGON**: insieme di **POLYGON**.
- **GEOMETRYCOLLECTION**: insieme di geometrie di qualsiasi tipo eccetto **GEOMETRYCOLLECTION**.

Il pacchetto *sf* rappresenta le *simple features* come oggetti *R* nativi (classi *S3*, liste, matrici, vettori) e li manipola con una serie di funzioni dal nome molto intuitivo. Analogamente ad altri software, come *PostGIS*, tutte le funzioni e i metodi di *sf* che operano sui dati spaziali sono

preceduti da `st_`, che si riferisce al tipo spaziale (*spatial type*); ciò li rende facilmente reperibili tramite il completamento della riga di comando. L'uso tipico prevede la lettura, la manipolazione e la scrittura di insiemi di caratteristiche, con attributi e geometrie.

La cosa “bella” di `sf` è che, quasi sempre, le geometrie e i loro attributi sono memorizzati e gestiti come `data.frame`, in cui una colonna contiene la specifica della geometria utilizzata. Poiché, però, spesso le geometrie non sono a entità singola (ad esempio sono costituite da insiemi di punti), le geometrie vengono inserite in una colonna-elenco, ovvero un elenco di lunghezza pari al numero di record nel `data.frame`, con ogni elemento dell'elenco che contiene la geometria semplice di quella caratteristica. Certo questo approccio implica una certa ridondanza, ma ha il vantaggio di essere molto chiaro. Poiché di solito non si lavora con singole geometrie, ma con insiemi di dati costituiti da insiemi di caratteristiche con attributi, le due cose vengono riunite in oggetti `sf`.

Ricapitolando, se qualcuno ci chiedesse «ma perché dobbiamo usare le *Simple Features*?», potremmo rispondere che:

- lettura e scrittura veloce dei dati;
- migliori prestazioni di plottaggio;
- gli oggetti `sf` possono essere trattati come *frame* di dati nella maggior parte delle operazioni;
- i nomi delle funzioni `sf` sono relativamente coerenti e intuitivi (iniziano tutti con `st_`);
- le funzioni `sf` possono essere combinate con l'operatore `%>%` e funzionano bene con la collezione di pacchetti *R tidyverse*.

Per cominciare a prendere la mano, cominciamo con l'esplorare la struttura di un oggetto `sf` integrato nel pacchetto stesso. È possibile leggere un oggetto `sf` con la funzione `st_read()` di `sf`. Ad esempio, qui leggiamo lo *shapefile* `nc` di `sf` che contiene le contee della Carolina del Nord, USA, nonché il loro nome, il numero di nascite e il numero di morti improvvise infantili nel 1974 e nel 1979.

```
library(sf)
## Linking to GEOS 3.11.0, GDAL 3.5.3, PROJ 9.1.0; sf_use_s2() is TRUE
pathshp <- system.file("shape/nc.shp", package = "sf")
nc <- st_read(pathshp, quiet = TRUE)
```

```
class(nc)
## [1] "sf"          "data.frame"
```

L'oggetto `sfnc` è un `data.frame` contenente una collezione di 100 caratteristiche semplici (righe) e 6 attributi (colonne) più una colonna-elenco con la geometria di ciascuna caratteristica. Un oggetto `sf` contiene i seguenti oggetti di classe `sf`, `sfc` e `sfg`:

- * **`sf` (elemento semplice)**: ogni riga del `data.frame` è un singolo elemento semplice composto da attributi e geometria
- * **`sfc` (simple feature geometry list-column)**: la colonna geometria del `data.frame` è un elenco-colonna di classe `sfc` con la geometria di ogni elemento semplice
- * **`sfg` (geometria dell'elemento semplice)**: ciascuna delle righe della colonna-elenco `sfc` corrisponde alla geometria dell'elemento semplice (`sfg`) di un singolo elemento semplice.

Possiamo vedere e tracciare le informazioni dell'oggetto `sf` come segue:

```
print(nc)
## Simple feature collection with 100 features and 14 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:   xmin: -84.32385 ymin: 33.88199 xmax: -75.45698
##                 ymax: 36.58965
## Geodetic CRS:   NAD27
## First 10 features:
##      AREA PERIMETER CNTY_ CNTY_ID      NAME  FIPS  FIPSNO  CRESS_
## ID BIR74  SID74
## 1  0.114    1.442  1825    1825      Ashe  37009  37009
## 5  1091      1
## 2  0.061    1.231  1827    1827  Alleghany 37005  37005
## 3   487      0
## 3  0.143    1.630  1828    1828      Surry  37171  37171
## 86 3188      5
## 4  0.070    2.968  1831    1831  Currituck 37053  37053
## 27  508      1
## 5  0.153    2.206  1832    1832 Northampton 37131  37131
## 66 1421      9
## 6  0.097    1.670  1833    1833   Hertford 37091  37091
```



```
46 1452 7
## 7 0.062 1.547 1834 1834 Camden 37029 37029
15 286 0
## 8 0.091 1.284 1835 1835 Gates 37073 37073
37 420 0
## 9 0.118 1.421 1836 1836 Warren 37185 37185
93 968 4
## 10 0.124 1.428 1837 1837 Stokes 37169 37169
85 1612 1
## NWBIR74 BIR79 SID79 NWBIR79 geometry
## 1 10 1364 0 19 MULTIPOLYGON (((-81.47276 3...
## 2 10 542 3 12 MULTIPOLYGON (((-81.23989 3...
## 3 208 3616 6 260 MULTIPOLYGON (((-80.45634 3...
## 4 123 830 2 145 MULTIPOLYGON (((-76.00897 3...
## 5 1066 1606 3 1197 MULTIPOLYGON (((-77.21767 3...
## 6 954 1838 5 1237 MULTIPOLYGON (((-76.74506 3...
## 7 115 350 2 139 MULTIPOLYGON (((-76.00897 3...
## 8 254 594 2 371 MULTIPOLYGON (((-76.56251 3...
## 9 748 1190 2 844 MULTIPOLYGON (((-78.30876 3...
## 10 160 2038 5 176 MULTIPOLYGON (((-80.02567 3...
plot(nc)
```

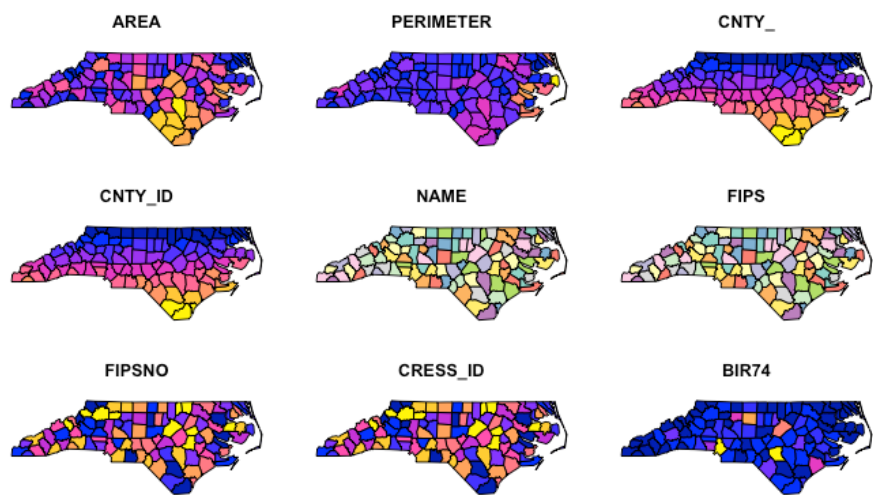


Figura 146 - Un esempio: il plot del North Carolina SIDS data frame.

È possibile suddividere gli insiemi di caratteristiche utilizzando la notazione delle parentesi quadre e utilizzare l'argomento drop per eliminare le geometrie.

```
nc[1, ] # prima riga
## Simple feature collection with 1 feature and 14 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:   xmin: -81.74107 ymin: 36.23436 xmax: -81.23989
ymax: 36.58965
## Geodetic CRS:   NAD27
##   AREA PERIMETER CNTY_ CNTY_ID NAME  FIPS FIPSNO CRESS_ID BIR74
SID74 NWBIR74
## 1 0.114      1.442  1825      1825 Ashe 37009  37009          5  1091
1      10
##   BIR79 SID79 NWBIR79                                geometry
## 1  1364      0      19 MULTIPOLYGON (((-81.47276 3...
nc[nc$NAME == "Ashe", ] # riga con NAME = "Ashe"
## Simple feature collection with 1 feature and 14 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:   xmin: -81.74107 ymin: 36.23436 xmax: -81.23989
ymax: 36.58965
## Geodetic CRS:   NAD27
##   AREA PERIMETER CNTY_ CNTY_ID NAME  FIPS FIPSNO CRESS_ID BIR74
SID74 NWBIR74
## 1 0.114      1.442  1825      1825 Ashe 37009  37009          5  1091
1      10
##   BIR79 SID79 NWBIR79                                geometry
## 1  1364      0      19 MULTIPOLYGON (((-81.47276 3...
nc[1, "NWBIR74"] # prima riga, colonna con NAME = NWBIR74
## Simple feature collection with 1 feature and 1 field
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:   xmin: -81.74107 ymin: 36.23436 xmax: -81.23989
ymax: 36.58965
```

```
## Geodetic CRS:  NAD27
##   NWBIR74                      geometry
## 1      10 MULTIPOLYGON (((-81.47276 3...
nc[1, "NWBIR74", drop = TRUE] # elimina le geometrie
## [1] 10
## attr(,"class")
## [1] "numeric"
```

La funzione `st_geometry()` può essere utilizzata per recuperare l'elenco-colonna della geometria delle *simple features* (*sfc*).

```
# Geometrie stampate in forma abbreviata
st_geometry(nc)
## Geometry set for 100 features
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:   xmin: -84.32385 ymin: 33.88199 xmax: -75.45698
ymax: 36.58965
## Geodetic CRS:  NAD27
## First 5 geometries:
## MULTIPOLYGON (((-81.47276 36.23436, -81.54084 3...
## MULTIPOLYGON (((-81.23989 36.36536, -81.24069 3...
## MULTIPOLYGON (((-80.45634 36.24256, -80.47639 3...
## MULTIPOLYGON (((-76.00897 36.3196, -76.01735 36...
## MULTIPOLYGON (((-77.21767 36.24098, -77.23461 3...
```

Adesso proviamo a creare noi stessi un oggetto *sf*. È possibile utilizzare la funzione `st_sf()` per creare un oggetto *sf* fornendo due elementi: (i) un *data.frame* con gli attributi di ogni caratteristica; (ii) una colonna-elenco di geometrie semplici *sfc* contenente geometrie semplici *sfg*. In dettaglio, si creano geometrie di elementi semplici *sfg* e si usa la funzione `st_sfc()` per creare un elenco-colonna di geometrie semplici *sfc* con tali elementi. Quindi, si usa `st_sf()` per mettere insieme il *data.frame* con gli attributi e l'elenco-colonna di geometrie semplici *sfc*.

Gli oggetti *sfg* possono essere, ad esempio, di tipo `POINT` (punto

singolo), `MULTIPOINT` (insieme di punti) o `POLYGON` (poligono) e possono essere creati rispettivamente con `st_point()`, `st_multipoint()` e `st_polygon()`. Un esempio banalissimo e completamente astratto, utile anche per collegare questo discorso a quello fatto per `ggplot2`, è il seguente:

```
# Punto singolo (punto come un vettore)
p1_sfg <- st_point(c(2, 2))
p2_sfg <- st_point(c(2.5, 3))

# Set di punti (punti come una matrice)
p <- rbind(c(6, 2), c(6.1, 2.6), c(6.8, 2.5),
           c(6.2, 1.5), c(6.8, 1.8))
mp_sfg <- st_multipoint(p)

# Polygon. Sequenza di punti che formano un anello chiuso,
# non intersecante.
# Il primo anello denota l'anello esterno,
# zero o più anelli successivi denotano buchi in questo anello
# esterno.
p1 <- rbind(c(10, 0), c(11, 0), c(13, 2),
           c(12, 4), c(11, 4), c(10, 0))
p2 <- rbind(c(11, 1), c(11, 2), c(12, 2), c(11, 1))
pol_sfg <- st_polygon(list(p1, p2))

# Crea un oggetto sf
p_sfc <- st_sfc(p1_sfg, p2_sfg, mp_sfg, pol_sfg)
df <- data.frame(v1 = c("A", "B", "C", "D"))
p_sf <- st_sf(df, geometry = p_sfc)

# Plot di punti singoli, set di punti e un poligono
library(ggplot2)
ggplot(p_sf) + geom_sf(aes(col = v1), size = 3) + theme_bw()
```

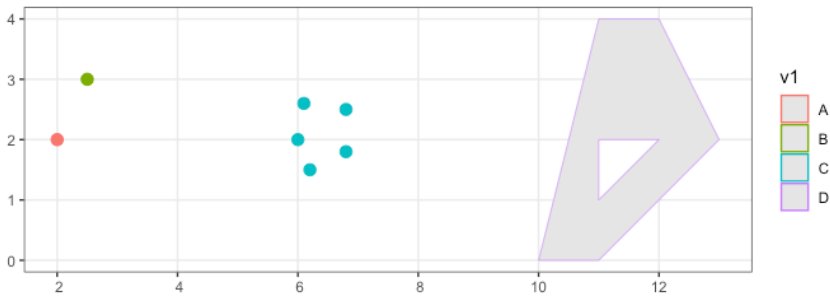


Figura 147 - I diversi tipi di geometria gestiti da sf in una sola lista di oggetti.

In questo esempio abbiamo creato e visualizzato un oggetto `sf` contenente due punti singoli, un insieme di punti e un poligono (con un attributo). Per prima cosa, si creano gli oggetti geometrici semplici (`sfg`) di tipo `POINT`, `MULTIPOINT` e `POLYGON`. Quindi si utilizza `st_sfc()` per creare un elenco-colonna di geometria semplice `sfc` con gli oggetti `sfg`. Infine, si usa `st_sf()` per mettere insieme il `data.frame` con l'attributo e l'elenco-colonna `sfc` della geometria semplice delle caratteristiche.

È importante notare alcune cose:

- la funzione `st_point()` vuole come *input* un vettore numerico;
- la funzione `st_multipoint()` vuole come *input* una matrice di numeri;
- la funzione `st_polygon()` vuole come *input* una lista formata da matrici di numeri corrispondenti alle coordinate dei punti (ordinati come righe);
- la funzione `st_sfc()` si mangia tutto: accetta come *input* zero o più geometrie di elementi semplici (oggetti di classe `sfg`), o un singolo elenco di tali oggetti; i valori `NULL` sono sostituiti da geometrie vuote.

Siamo pronti per giocare con un caso di studio ecologico e reale. Carichiamo dal web un `data.frame` contenente le informazioni georeferenziate (Longitudine e Latitudine) di una serie di eventi di pesca (a strascico) che hanno determinato la cattura di 6 specie dal grande valore commerciale.

```
catch =  
  read.csv("https://raw.githubusercontent.com/tommaso-russo/LibroR2/  
main/Spatial/SpeciesCatch.csv")
```

Ognuna delle sei specie è identificata, nel `data.frame`, dal suo codice FAO a 3 lettere:

- **ARA** – *Aristeus antennatus* (il gambero viola).
- **ARS** – *Aristaeomorpha foliacea* (il gambero rosso).
- **DPS** – *Parapenaeus longirostris* (il gamberetto rosa).
- **HKE** – *Merluccius merluccius* (il nasello europeo).
- **MUT** – *Mullus barbatus* (la triglia di fango).
- **NEP** – *Nephrops norvegicus* (lo scampo).

Il data frame contiene, oltre alle coordinate geografiche e la specie, la colonna **KG** che riporta il peso complessivo delle catture, e la profondità del fondale marino in corrispondenza di ciascuna posizione di pesca. Il data frame ha 6096 righe corrispondenti ad altrettante posizioni di pesca. Dunque, creiamo un oggetto **sf** contenente tanti punti quante sono le posizioni di pesca, usando la funzione **st_multipoint()**, e lo rappresentiamo graficamente con **ggplot**.

```
fishing_positions <- st_multipoint(data.matrix(catch[, c("LON",
"LAT")]))
ggplot(data = fishing_positions) +
  geom_sf() +
  theme_test() +
  xlab("Longitude") +
  ylab("Latitude")
```

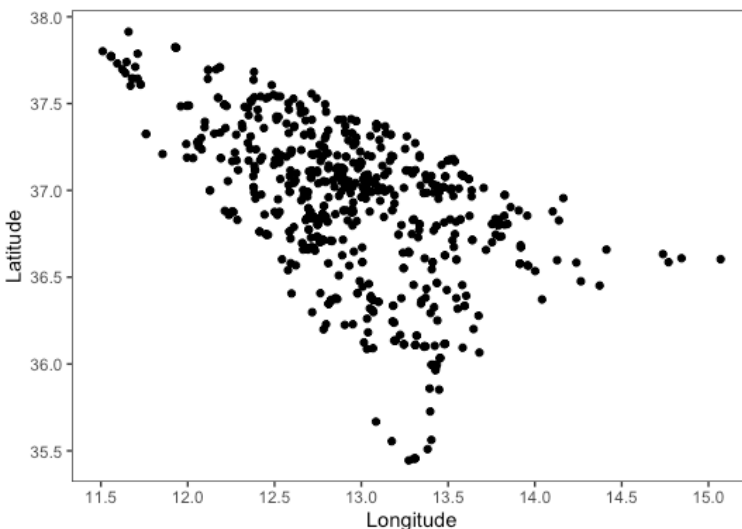


Figura 148 - Le posizioni di pesca nel nostro data frame.

Se invece vogliamo sfruttare al massimo l'informazione presente in un determinato data frame, possiamo usare direttamente la funzione `st_as_sf`. Si tratta di una funzione che converte direttamente un oggetto appartenente a diverse classi (`data.frame`, `sfc`, `Spatial`, `map`, ecc.) in un oggetto `sf`. Il vantaggio di questa funzione è che non dobbiamo trasformare i dati, ma solo specificare quali sono le coordinate (mediante l'argomento `coords` della funzione stessa). Vediamo un esempio:

```
catch_sf <- st_as_sf(catch,
                     coords = c("LON", "LAT"),
                     remove = F)
st_crs(catch_sf) = 4326
head(catch_sf)
## Simple feature collection with 6 features and 6 fields
## Geometry type: POINT
## Dimension:      XY
## Bounding box:   xmin: 12.2135 ymin: 36.10117 xmax: 14.0015 ymax:
36.882
## Geodetic CRS:   WGS 84
##   HAUL SPECIES      LON      LAT KG  DEPTH      geometry
## 1     6      ARA 12.21350 36.88200 804 -538.00 POINT (12.2135 36.882)
## 2     7      ARA 12.41383 36.76233  76 -561.50 POINT (12.41383 36.76233)
## 3    14      ARA 13.36367 36.10117 243 -680.00 POINT (13.36367 36.10117)
## 4    15      ARA 14.00150 36.53517 103 -675.50 POINT (14.0015 36.53517)
## 5    18      ARA 13.60408 36.39292  37 -735.75 POINT (13.60408 36.39292)
## 6    19      ARA 13.80733 36.73483  48 -784.00 POINT (13.80733 36.73483)
```

Come vediamo, la funzione ha assegnato alle diverse posizioni di pesca la geometria `point`, ed ha conservato le altre colonne del data frame, che adesso possiamo sfruttare per plottare i dati.

Questo esempio è utile anche per riprendere il concetto di *CRS*. Infatti, dopo aver creato l'oggetto `catch_sf`, abbiamo usato la funzione `st_set_crs` per attribuirgli il codice *CRS* 4326, che corrisponde al *Geodetic coordinate system for World*, cioè il famoso e diffuso sistema di coordinate (Longitude e Latitudine) universali in gradi, minuti e secondi.

Nel pacchetto `sf`, è possibile utilizzare la funzione `st_crs` per estrarre le informazioni sul sistema di coordinate (CS) da un oggetto. In questo modo si ottiene il codice *EPSG* e la stringa di proiezione *PROJ4*. Infatti:

```
st_crs(catch_sf)
## Coordinate Reference System:
##   User input: EPSG:4326
##   wkt:
##   GEOGCRS["WGS 84",
##     ENSEMBLE["World Geodetic System 1984 ensemble",
##       MEMBER["World Geodetic System 1984 (Transit)"],
##       MEMBER["World Geodetic System 1984 (G730)"],
##       MEMBER["World Geodetic System 1984 (G873)"],
##       MEMBER["World Geodetic System 1984 (G1150)"],
##       MEMBER["World Geodetic System 1984 (G1674)"],
##       MEMBER["World Geodetic System 1984 (G1762)"],
##       MEMBER["World Geodetic System 1984 (G2139)"],
##       ELLIPSOID["WGS 84",6378137,298.257223563,
##         LENGTHUNIT["metre",1]],
##       ENSEMBLEACCURACY[2.0]],
##     PRIMEM["Greenwich",0,
##       ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##       AXIS["geodetic latitude (Lat)",north,
##         ORDER[1],
##         ANGLEUNIT["degree",0.0174532925199433]],
##       AXIS["geodetic longitude (Lon)",east,
##         ORDER[2],
##         ANGLEUNIT["degree",0.0174532925199433]],
##     USAGE[
##       SCOPE["Horizontal component of 3D system."],
##       AREA["World."],
##       BBOX[-90,-180,90,180]],
##     ID["EPSG",4326]]
```


L'esame attento della risposta del terminale ci permette di riconoscere le componenti che avevamo precedentemente introdotto. Il pacchetto `sf` ci consente anche di trasformare (di fatto ri-proiettare) un qualsiasi oggetto da un CRS ad un altro. La funzione che ci serve è `st_transform`. Il suo utilizzo è semplicissimo, vediamo un esempio e facciamo un confronto grafico:

```
catch_sf_transform = st_transform(catch_sf, 3347)

par(mfrow=c(1,2)) # plottiamo i grafici insieme
plot(st_geometry(catch_sf), main = "EPSG: 4326", pch = 19, cex =
0.2)
plot(st_geometry(catch_sf_transform), main = "EPSG: 3347", pch = 19,
cex = 0.2)
```

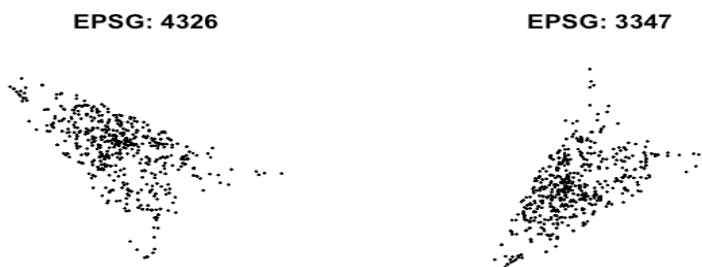


Figura 149 - Un esempio di trasformazione del CRS.

Adesso plottiamo nuovamente le coordinate delle posizioni di pesca, ma questa volta evidenziando la profondità del fondale marino in corrispondenza di ognuna di esse (le specie sfruttate, come abbiamo detto, sono tutte *demersali*, cioè vivono in prossimità del fondale e sono, quindi, preda della pesca a strascico).

```
ggplot(data = catch_sf) +
  geom_sf(aes(colour = DEPTH)) +
  scale_fill_viridis_c(option = "mako", direction = -1) +
```

```
theme_test() +
xlab("Longitude") +
ylab("Latitude")
```

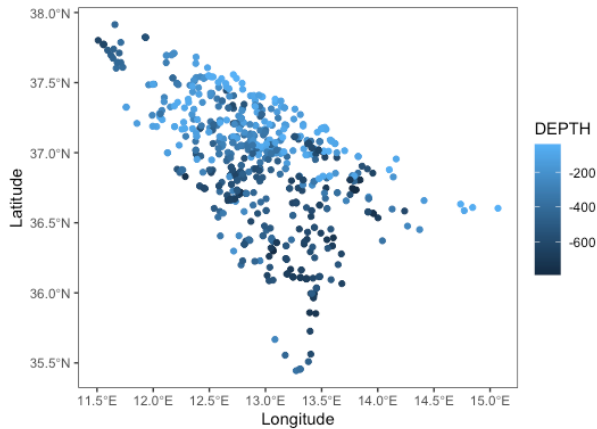


Figura 150 - Profondità delle posizioni di pesca.

Ehi, la distribuzione dei punti comincia a sembrare vagamente familiare (almeno per chi ha presente la morfologia della costa meridionale della Sicilia), ma una mappa di dati spaziali che si rispetti ha bisogno di avere dei riferimenti. Siccome stiamo analizzando dati relativi all’ambiente marino, la scelta più logica è usare le terre emerse circostanti. Un primo approccio potente e facile è fornito dal pacchetto `mapview`. Questo strumento ci permette di plottare in pochi istanti **mappe interattive** (in cui, ad esempio, possiamo fare uno zoom manuale), variando facilmente il tema di fondo e tante opzioni grafiche. Di seguito un esempio di mappa interattiva realizzata con `mapview`:

```
library(mapview)
mapview(catch_sf, cex = 1)
```

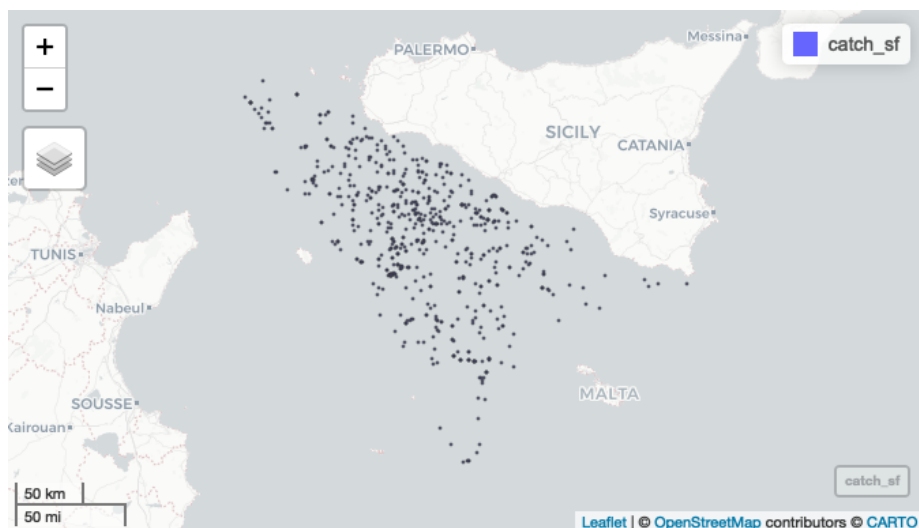


Figura 151 - Un esempio del potente pacchetto mapview.

`mapview` usa le mappe di OpenStreetMap, il più grande database geografico libero e modificabile di tutto il mondo, costruito dal lavoro di volontari e rilasciato con una licenza libera. Si tratta di un gigantesco progetto collaborativo, con milioni di utenti registrati in tutto il mondo, il cui scopo è creare e fornire dati geografici liberi a chiunque li voglia utilizzare.

Se invece quello che ci occorre è una **mappa statica** (meno flessibile ma più utile quando, ad esempio, vogliamo produrre figure in serie che abbiano esattamente la stessa ampiezza e visuale), possiamo usare `ggmap`, che è una estensione di `ggplot2`. `ggmap` fornisce raccolta di funzioni per visualizzare dati e modelli spaziali su mappe statiche provenienti da varie fonti *online* (ad esempio *Google Maps* e *Stamen Maps*). Include strumenti comuni a questi compiti, tra cui funzioni per la geolocalizzazione ed il *routing*. Qui ci limiteremo a mostrare le funzionalità di *Stadia Maps*. *Stadia Maps* offre piastrelle di mappe in diversi stili, comprese quelle aggiornate di *Stamen Design*. È richiesta una chiave API (privata e unica per ogni utente – le istruzioni per crearla si possono trovare qui, ma non è necessaria una carta di credito per iscriversi e c'è un livello gratuito per l'uso non commerciale).

Dopo aver caricato la libreria `ggmap`, useremo la nostra chiave personale per registrarci prima di generare la mappa:

```

library(ggmap)
## i Google's Terms of Service: <https://mapsplatform.google.com>
##   Stadia Maps' Terms of Service: <https://stadiamaps.com/terms-
of-service/>
##   OpenStreetMap's Tile Usage Policy: <https://operations.
osmfoundation.org/policies/tiles/>
## i Please cite ggmap if you use it! Use `citation("ggmap")` for
details.
key = read.table("Key.txt")
ggmap::register_stadiamaps(key)

map10 = get_stadiamap(bbox = as.numeric(st_bbox(catch_sf)) *
                      c(0.95, 0.99, 1.05, 1.01),
                      zoom = 8,
                      maptype = "alidade_smooth",
                      messaging = FALSE)
## i © Stadia Maps © Stamen Design © OpenMapTiles © OpenStreetMap
contributors.
ggmap(map10) +
  geom_sf(data = catch_sf, aes(colour = DEPTH), inherit.aes = F) +
  scale_fill_viridis_c(option = "mako", direction = -1) +
  theme_test() +
  xlab("Longitude") +
  ylab("Latitude")
## Coordinate system already present. Adding new coordinate system,
which will
## replace the existing one.

```

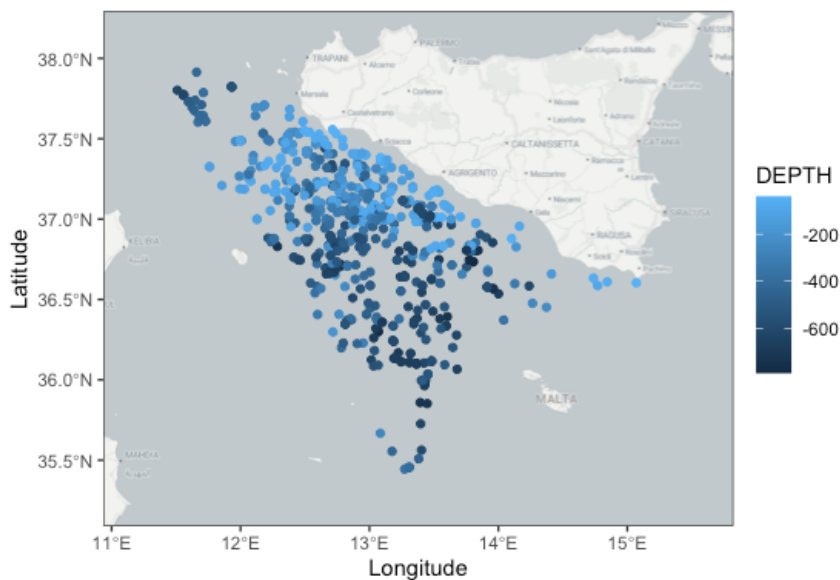


Figura 152 - Mappa ottenuta mediante il pacchetto ggmap.

Notare che, in maniera un po' furba, abbiamo allargato il campo definito dalla funzione `st_bbox`, in maniera da avere un'immagine più chiara e senza punti che cadano sui suoi bordi.

Adesso torniamo all'esplorazione dei nostri dati e usiamo il campo `KG` (quantità delle catture):

```
ggmap(map10) +
  geom_sf(data = catch_sf, aes(size = KG, colour = SPECIES),
inherit.aes = F) +
  theme_test() +
  facet_wrap(~ SPECIES, nrow = 3) +
  xlab("Longitude") +
  ylab("Latitude")
## Coordinate system already present. Adding new coordinate system,
## which will
## replace the existing one.
```

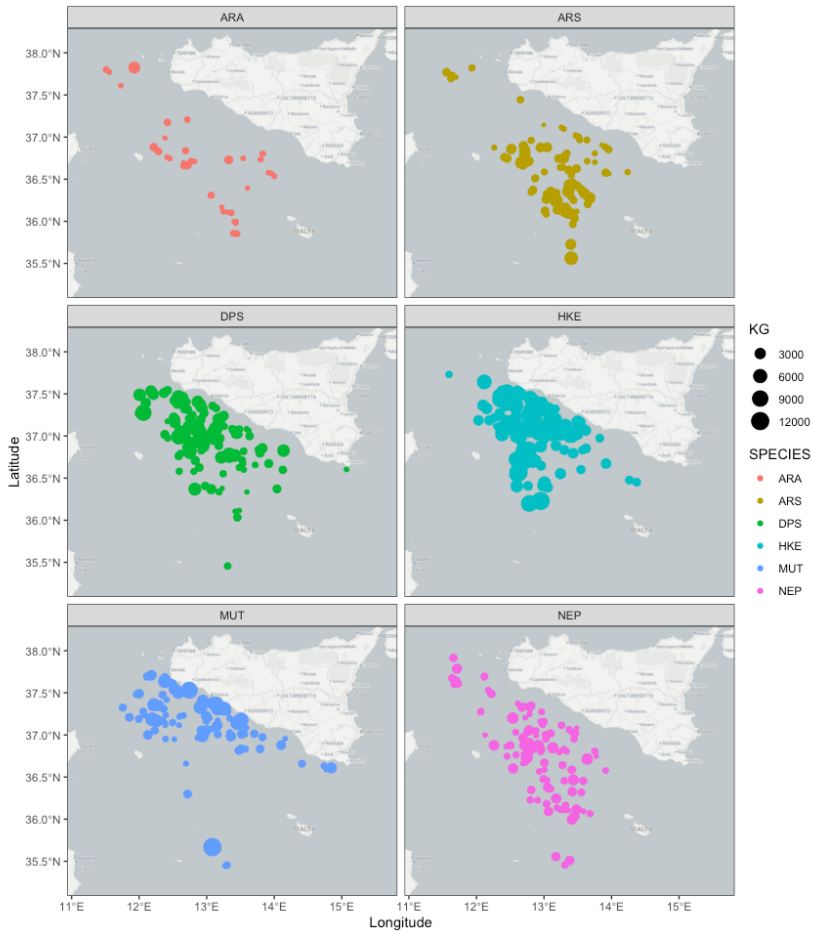


Figura 153 - Mappa delle posizioni di pesca e delle quantità per specie.

Cominciano a notarsi delle cose interessanti: le catture non sono distribuite in maniera omogenea per le varie specie. Questo aspetto, in realtà, potremmo approfondirlo senza analisi spaziali, per esempio aggregando i dati rispetto ad intervalli batimetrici di ampiezza 100 metri:

```
catch$stratum = c("800-700", "700-600",
                  "600-500", "500-400",
                  "400-300", "300-200",
                  "200-100", "100-0") [findInterval(catch$DEPTH,
                                                    seq(-800, 0, by = 100))]
```

```

catch$stratum = factor(catch$stratum,
                        levels = c("800-700", "700-600",
                                   "600-500", "500-400",
                                   "400-300", "300-200",
                                   "200-100", "100-0"))

ggplot(data = catch, aes(x = stratum, y = KG)) +
  geom_boxplot(aes(fill = SPECIES)) +
  facet_wrap(~ SPECIES, nrow = 3, scales = "free_y") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5,
                                     hjust=1))

```

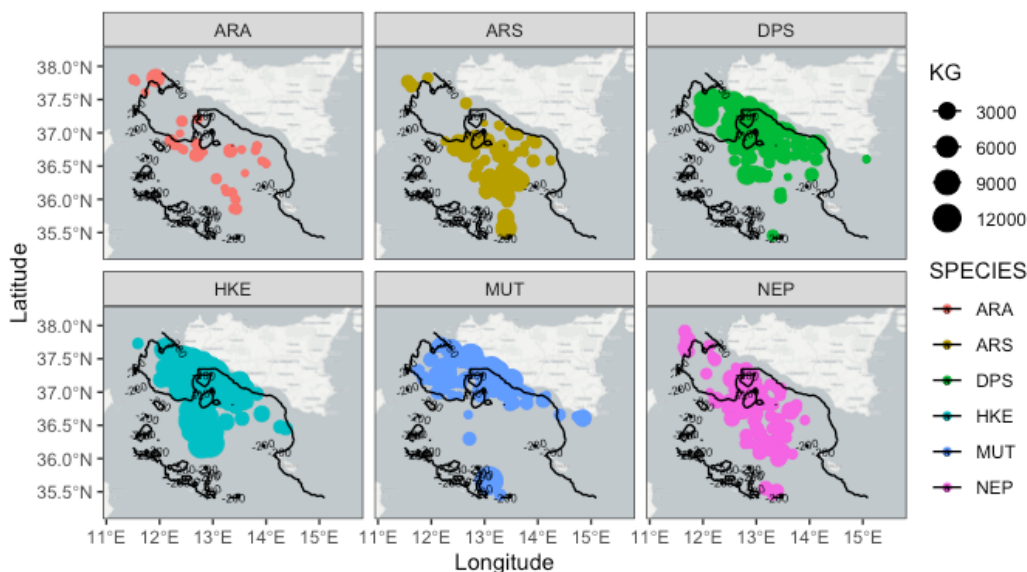


Figura 154 - Boxplot delle quantità di catture per strato batimetrico.

Come possiamo fare per evidenziare questa distribuzione delle catture, in funzione della profondità, sulla nostra mappa? La risposta è: usando i pacchetti *marmap* e *metR*. Il pacchetto *marmap* importa dati xyz dal database del NOAA (*National Oceanic and Atmospheric Administration*), dalla GEBCO (*General Bathymetric Chart of the Oceans*) e da altre fonti, consentendo di ottenere profondità/altitudine in

base alle coordinate geografiche e di preparare figure pronte per la pubblicazione. Per usarlo, ci conviene definire gli estremi dell'area di nostro interesse e poi scaricare i dati batimetrici, mediante la funzione `getNOAA.bathy`. Questa funzione ci consente anche di specificare la risoluzione desiderata (in minuti). Per estrarre gli estremi useremo la funzione `st_bbox` di `sf`.

```
library(marmap)
area_extremes = st_bbox(catch_sf)
area_extremes
##      xmin      ymin      xmax      ymax
## 11.51033 35.44467 15.07033 37.91383
bat = getNOAA.bathy(lon1 = area_extremes[1],
                    lon2 = area_extremes[3],
                    lat1 = area_extremes[2],
                    lat2 = area_extremes[4],
                    resolution = 1)
```

Poi usiamo `metR` per plottare (ed etichettare) le isobate desiderate...

```
library(metR)

ggmap(map10) +
  geom_sf(data = catch_sf, aes(size = KG, colour = SPECIES),
inherit.aes = F) +
  theme_test() +
  xlab("Longitude") +
  ylab("Latitude") +
  geom_contour(data = bat,
              aes(x=x, y=y, z=z),
              breaks = -200,
              linewidth = 0.5,
              colour = "black",
              show.legend = TRUE) +
  geom_text_contour(data = bat,
                  aes(x=x, y=y, z=z),
```



```

stat = "text_contour",
breaks = -200,
size = 2,
nudge_x = 0.005,
skip = 0,
colour = "black",
min.size = 1,
rotate = TRUE,
show.legend = TRUE) +
facet_wrap(~ SPECIES, nrow = 2)

```

Figura 155 - Un'altra mappa delle posizioni di pesca e delle quantità per specie.

In questo esempio abbiamo usato solo l'isobata dei -400 metri, ma si vede chiaramente che le catture di **ARA** e **ARS** (le due specie di aristeidi) sono sempre oltre i -400 metri (sono specie profonde), mentre **MUT** (la triglia) è una specie superficiale, pescata esclusivamente sulla piattaforma continentale.

A questo punto, una possibile strada da imboccare per proseguire l'analisi potrebbe essere quella di suddividere l'area di studio in una serie di blocchi, come per esempio poligoni, che abbiano un significato gestionale (ad esempio aree che si vogliono proteggere dalla pesca) o, più, semplicemente, le celle di una griglia regolare. Con **sf** è molto semplice costruire una griglia a partire da un altro oggetto già presente in memoria. Qui di seguito usiamo la funzione **st_make_grid** per generare una griglia avente celle quadrate di 5 Km di lato. Poi la plottiamo per osservare il risultato:

```

grid_sf = st_as_sf(st_make_grid(catch_sf,
                                cellsz = 10/89.5695,
                                what = "polygons",
                                square = TRUE,
                                crs = 4326,
                                flat_topped = FALSE))

ggmap(map10) +

```

```

geom_sf(data = catch_sf, inherit.aes = F) +
geom_sf(data = grid_sf, colour = "darkgrey",
        fill = NA,
        inherit.aes = F) +
theme_test() +
xlab("Longitude") +
ylab("Latitude")

```

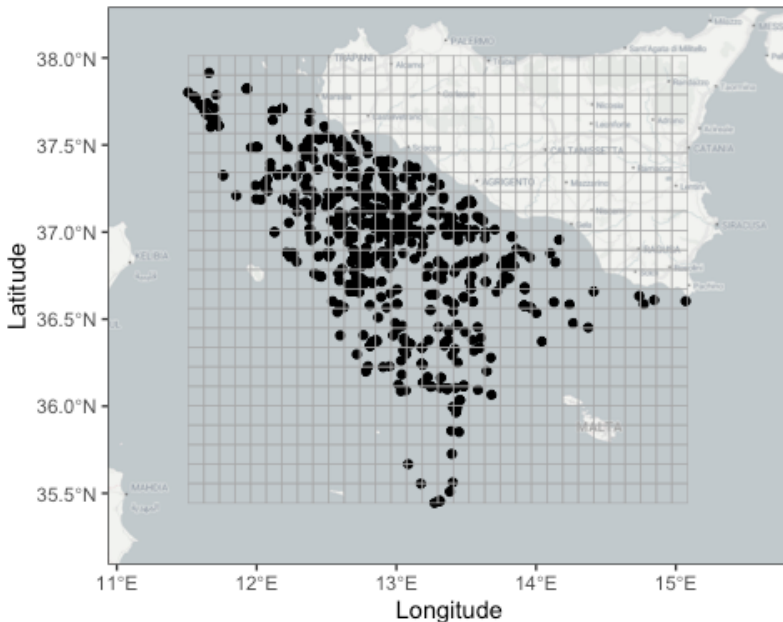


Figura 156 - Mappa delle posizioni di pesca con sovrapposta la nostra griglia.

Il risultato è quello desiderato? Forse non ci serve che la griglia copra le terre emerse, e anzi sarebbe meglio ritagliarla in modo che copra solo la porzione propriamente marina dell'area di studio. Per farlo, abbiamo bisogno di avere un oggetto `sf` che rappresenti le terre emerse. A tal proposito, possiamo usare l'oggetto `world` che è fornito dal pacchetto `spData`. Questa è una visione complessiva dell'oggetto in cui abbiamo scelto di colorare i poligoni in base al continente a cui appartengono (sono presenti anche altre informazioni nel data frame dell'oggetto).

```
library(spData)
## To access larger datasets in this package, install the
spDataLarge
## package with: `install.packages('spDataLarge',
## repos='https://nowosad.github.io/drat/', type='source')`
ggplot(data = world) +
  geom_sf(aes(fill = continent))
```

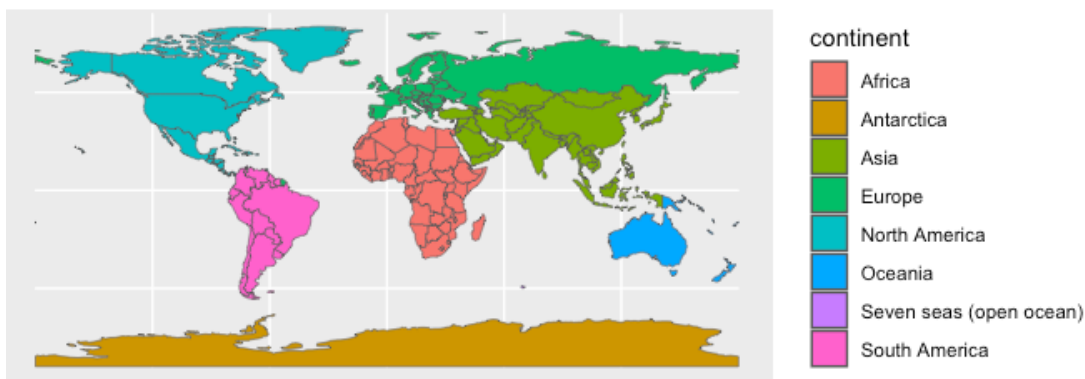


Figura 157 - Mappa dei continenti del dataframe world.

Uno zoom sulla nostra regione di interesse ...

```
ggplot(data = world) +
  geom_sf(fill = "brown") +
  geom_sf(data = grid_sf, colour = "darkgrey",
    fill = NA, inherit.aes = F) +
  coord_sf(xlim = st_bbox(catch_sf)[c(1, 3)] * c(0.95, 1.05),
    ylim = st_bbox(catch_sf)[c(2, 4)] * c(0.99, 1.01)) +
  theme_test()
```

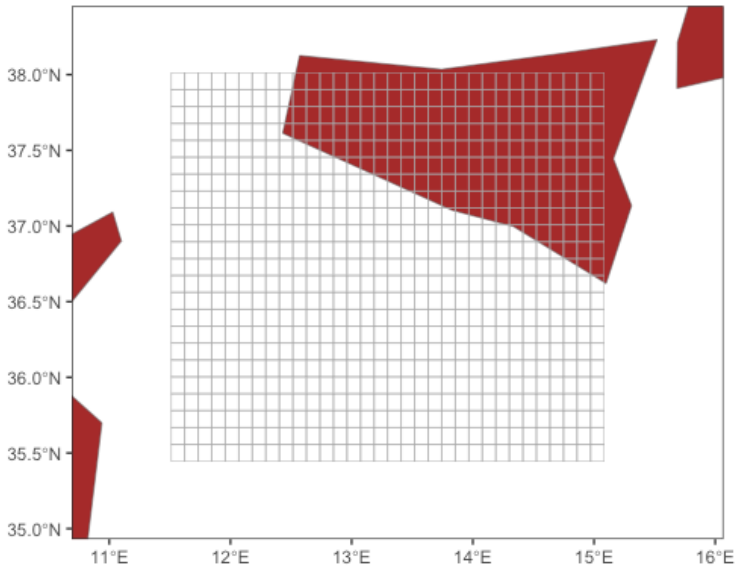


Figura 158 - Zoom della mappa sulla regione di interesse.

A questo punto possiamo usare le terre emerse per ritagliare la griglia ed escludere le celle che non ci servono. Per farlo, possiamo usare la funzione `st_intersects()`. Questa funzione fa parte di una famiglia molto più ampia che comprende:

- `st_disjoint()`
- `st_touches()`
- `st_crosses()`
- `st_within()`
- `st_contains()`
- `st_contains_properly()`
- `st_overlaps()`
- `st_equals()`
- `st_covers()`
- `st_covered_by()`
- `st_equals_exact()`
- `st_is_within_distance()`

Quello che fa ciascuna di queste funzioni è abbastanza intuitivo (si capisce dal nome), e non ha senso esaminarle tutte. In sostanza si tratta di una serie di funzioni che permettono di applicare l'algebra di

Boole e suoi derivati (es. selezione in base alla distanza) a qualsivoglia coppia di oggetti spaziali.

Introduciamo quindi l'utilizzo di una funzione MOLTO potente e versatile: `st_intersects()`. Questa funzione ci permette di cercare le intersezioni di qualsivoglia coppia di oggetti `sf`, ma dobbiamo prestare attenzione all'ordine in cui forniamo gli argomenti e capire la struttura dell'oggetto che questa funzione ci restituisce. Vedremo infatti diversi esempi. Tanto per cominciare, usiamo la funzione `st_intersects` dandole come argomenti `grid_sf` e `world`.

```
inter = st_intersects(grid_sf, world)
inter
## Sparse geometry binary predicate list of length 736, where the
## predicate was `intersects'
## first 10 elements:
## 1: (empty)
## 2: (empty)
## 3: (empty)
## 4: (empty)
## 5: (empty)
## 6: (empty)
## 7: (empty)
## 8: (empty)
## 9: (empty)
## 10: (empty)
```

Ma di che cosa si tratta? Questa `*Sparse geometry binary predicate list of length 736, where the predicate was intersects*` è una struttura che può essere concettualmente immaginata come un `data.frame` di coppie tra gli elementi `digrid_sfe` quelli di `diworld`. Dal momento che `grid_sf` è un oggetto di classe `POLYGON` contiene una serie di poligoni, mentre `world` è un oggetto di classe `MULTIPOLYGON`, la struttura di interci dice quale elemento `digrid_sf` (cioè quale poligono) corrisponde a quale elemento di `world`. Siccome nessuno dei primi 10 punti `digrid_sfcade` su una parte di `world`, sono tutti `**empty**` in `inter`. Ma non è così per tutti! Infatti:

```
as.numeric(inter)
```

```
## [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [19] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [37] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [55] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [73] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [91] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [109] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [127] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [145] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [163] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [181] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [199] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [217] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [235] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [253] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [271] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [289] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [307] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [325] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [343] NA NA NA NA NA NA NA NA NA NA NA 142 NA NA NA NA NA NA NA NA
## [361] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [379] NA NA NA 142 142 142 NA NA NA NA NA NA NA NA NA NA NA NA NA
## [397] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA 142 142 142
## [415] 142 142 NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [433] NA NA NA NA NA NA NA NA NA 142 142 142 142 142 142 142 142 NA NA
## [451] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [469] 142 142 142 142 142 142 142 142 142 142 142 NA NA NA NA NA NA
## [487] NA NA NA NA NA NA NA NA NA NA NA NA NA 142 142 142 142 142 142 142
## [505] 142 142 142 142 142 142 142 142 NA NA NA NA NA NA NA NA NA NA
## [523] NA NA NA NA 142 142 142 142 142 142 142 142 142 142 142 142 142
## [541] 142 142 142 142 NA NA NA NA NA NA NA NA NA NA NA NA 142 142
## [559] 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142
## [577] NA NA NA NA NA NA NA NA NA NA 142 142 142 142 142 142 142 142
## [595] 142 142 142 142 142 142 142 142 142 142 142 142 142 NA NA NA NA
## [613] NA NA NA NA 142 142 142 142 142 142 142 142 142 142 142 142 142
```

```
## [631] 142 142 142 142 142 142 142 142 142 142 142 NA NA NA NA NA NA NA NA
## [649] 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142
## [667] 142 142 142 142 142 142 NA NA NA NA NA NA NA NA 142 142 142 142
## [685] 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142
## [703] 142 142 NA NA NA NA NA NA NA NA 142 142 142 142 142 142 142 142
## [721] 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142 142
```

Ora vediamo che molti elementi di `grid_sf` sono associati all'elemento 142 di `world`. Chi sarà l'elemento 142 di `world`? La troviamo in figura:

```
world[142,]
## Simple feature collection with 1 feature and 10 fields
## Geometry type: MULTIPOLYGON
## Dimension:      XY
## Bounding box:   xmin: 6.749955 ymin: 36.61999 xmax: 18.48025 ymax:
47.11539
## Geodetic CRS:   WGS 84
## # A tibble: 1 × 11
##   iso_a2 name_long continent region_un subregion   type area_km2
pop lifeExp
##   <chr>  <chr>      <chr>    <chr>    <chr>      <chr>    <dbl>
<dbl>    <dbl>
## 1 IT      Italy      Europe    Europe    Southern E... Sove... 315105.
6.08e7    83.1
## # i 2 more variables: gdpPercap <dbl>, geom <MULTIPOLYGON [°]>
ggplot(data = world[142,]) +
  geom_sf(aes(fill = continent))
```

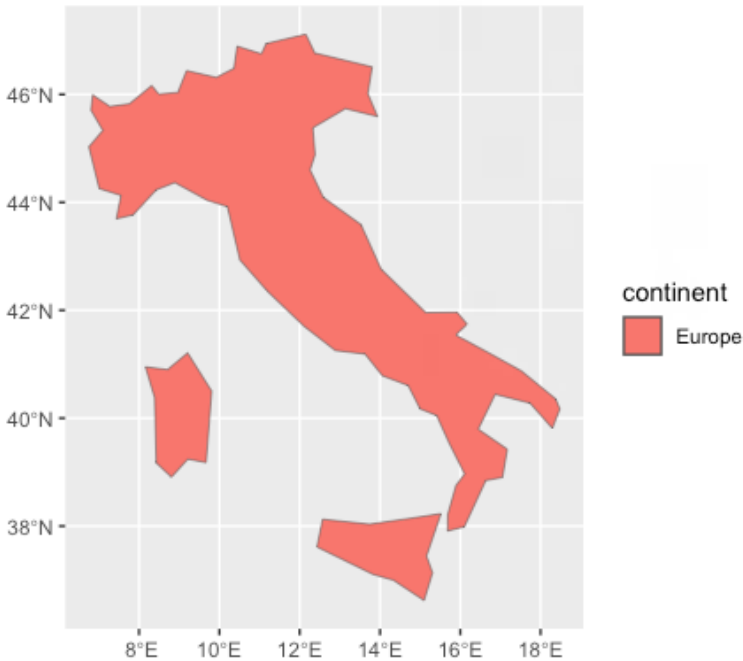


Figura 159 - La sezione del dataframe world relativa all'Italia.

Sorpresa: l'Italia (in `world` tutte le regioni italiane, Sicilia compresa, sono accorpate). Ora possiamo sfruttare questo risultato per ritagliare le celle di `grid_sf` che toccano la terraferma. In pratica eliminiamo da `grid_sf` tutte le celle che non sono NA in `as.numeric(inter)`:

```
grid_sf_cut = grid_sf[-unique(which(!is.na(as.numeric(inter)))),]
ggmap(map10) + geom_sf(data = grid_sf_cut, colour = "blue", inherit.
aes = F) +
  theme_test() + xlab("Longitude") + ylab("Latitude")
```

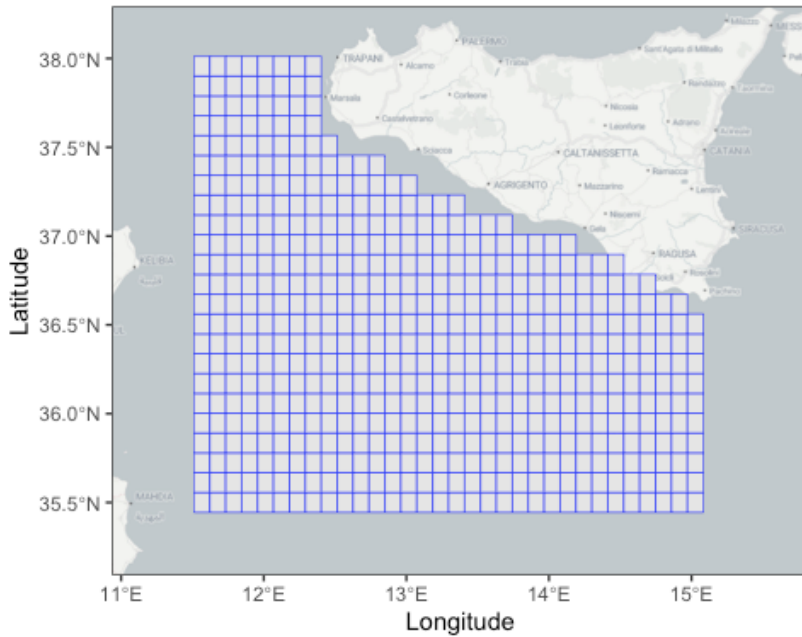



Figura 160 - Mappa con la nostra griglia ritagliata.

Questa operazione di ritaglio della griglia può essere (e in questo caso non è) “indolore”. Per dimostrarlo, vediamo la figura...

```
ggmap(map10) +
  geom_sf(data = grid_sf_cut,
    colour = "darkgrey",
    inherit.aes = F) +
  geom_sf(data = catch_sf,
    inherit.aes = F) +
  theme_test() +
  xlab("Longitude") +
  ylab("Latitude")
```

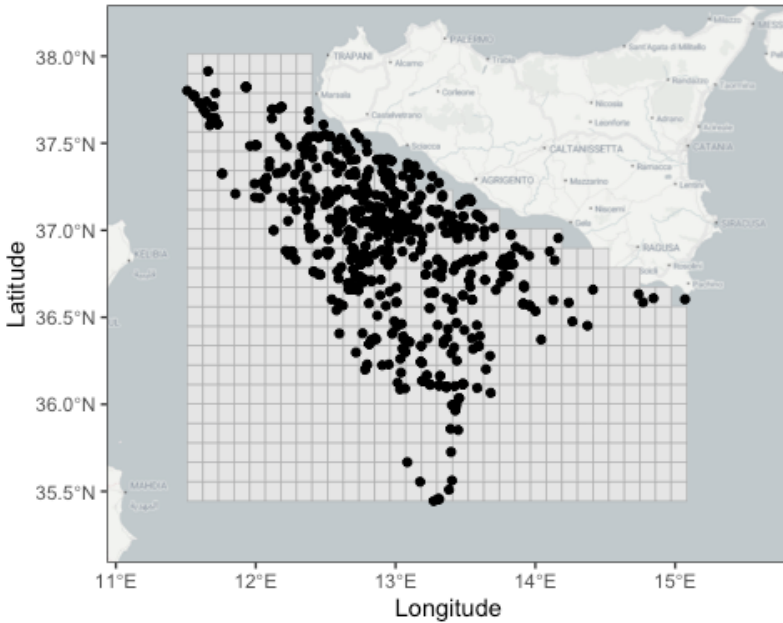


Figura 161 - La nostra griglia ritagliata e i punti delle cale di pesca.

Ad un occhio attento non sfuggirà, adesso alcuni punti cadono fuori dalla griglia! Questa cosa potrebbe compromettere alcune analisi successive, per cui ci conviene rimuovere i punti non sovrapposti con la griglia.

Per farlo, usiamo daccapo la funzione `st_intersects`. Questa funzione può essere usata anche per confrontare un oggetto `POINT` (come nel caso di `catch_sf`) con un oggetto di tipo `POLYGON`, e capire come gli elementi nell'oggetto `POINT` si associano a quelli nel `POLYGON`. La funzione `st_intersects` ci restituirà, anche in questo caso, un oggetto di classe `Sparse geometry binary predicate (sbgp)` e lista con i seguenti attributi: le `row.names` corrispondono agli indici degli elementi dell'oggetto `POINT` che ricadono nei rispettivi elementi specificati dopo il punto (le `ncol`) dell'oggetto `POLYGON`.

Dunque la struttura seguente:

```
points_on_grid <- st_intersects(catch_sf, grid_sf_cut)
points_on_grid
## Sparse geometry binary predicate list of length 618, where the
```

```
## predicate was `intersects'
## first 10 elements:
## 1: 387
## 2: 360
## 3: 177
## 4: 311
## 5: 275
## 6: 372
## 7: 113
## 8: 401
## 9: 362
## 10: 114
```

ci dice che gli elementi da 1 a 10 dell'oggetto `POINT` cadono, rispettivamente, negli elementi 387, 360, 177, 311, 275, 372, 113, 401, 362, 114 dell'oggetto `POLYGON`.

Ora che lo sappiamo, possiamo usare questo risultato per generare un vettore da associare a `catch_sf` in modo che `catch_sf` abbia una colonna denominata **ongrid** che ci dica quali punti in `catch_sf` cadono nella nostra griglia.

```
df_ongrid = as.data.frame(points_on_grid)
df_ongrid[row.names(points_on_grid)]
## data frame with 0 columns and 578 rows
colnames(df_ongrid) = c("POINT", "POLYGON")
head(df_ongrid)
##   POINT POLYGON
## 1     1     387
## 2     2     360
## 3     3     177
## 4     4     311
## 5     5     275
## 6     6     372
```

Se confrontiamo le dimensioni di questo data.frame con quelle di `catch_sf` noteremo una discrepanza:

```
dim(df_ongrid)
## [1] 578  2
dim(catch_sf)
## [1] 618  7
```

`df_ongrid` ha meno righe di `catch_sf`, perché evidentemente alcuni punti di `catch_sf` non cadono nelle celle della nostra griglia (che avevamo ritagliato). Possiamo infatti identificare i punti “fuori griglia”:

```
setdiff(1:nrow(catch_sf), df_ongrid$POINT)
## [1] 35 145 146 147 148 149 154 158 186 210 272 273 275 276 278
280 299 301 302
## [20] 400 401 402 403 404 405 406 407 408 410 416 417 418 423 425
427 428 449 453
## [39] 454 496
```

Eccoli qui!!!

A questo punto possiamo elegantemente trovare un modo per associare a `catch_sf` un campo vettoriale che specifichi i punti su griglia e quelli fuori.

```
on_grid = numeric(nrow(catch_sf))
on_grid[df_ongrid$POINT] = 1
catch_sf$on_grid = on_grid
```

Visualizziamo il risultato:

```
ggmap(map10) +
  geom_sf(data = grid_sf_cut,
    colour = "darkgrey",
    inherit.aes = F) +
  geom_sf(data = catch_sf,
    aes(colour = factor(on_grid)),
    inherit.aes = F) +
  scale_colour_manual(values = c("red", "black")) +
  theme_test() +
```

```
xlab("Longitude") +
ylab("Latitude")
```

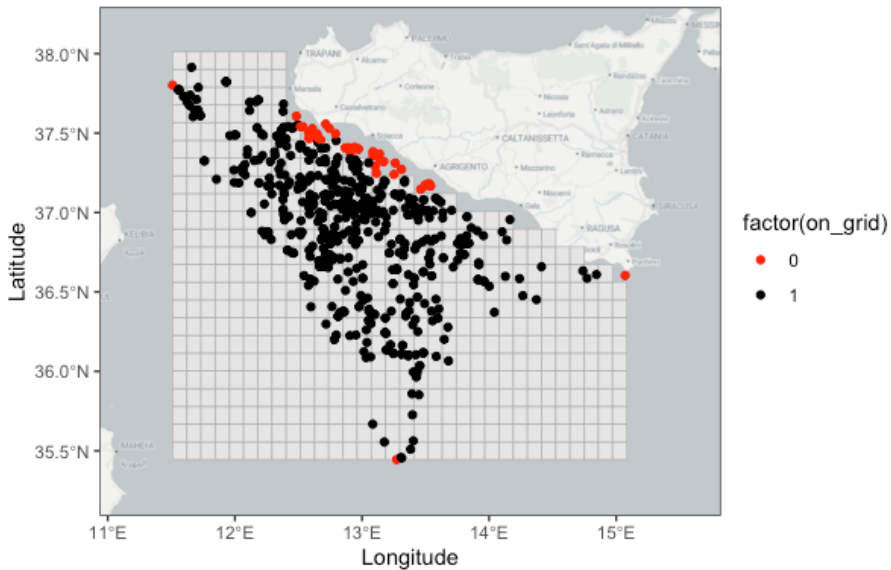


Figura 162 - Mappa con le cale di pesca dentro e fuori la griglia.

Sembra che il *pattern* sia del tutto coerente con le operazioni di “ritaglio” che abbiamo svolto. Quindi, potremmo restringere la nostra analisi ai soli punti che stanno nella griglia.

```
catch_sf = catch_sf[which(on_grid == 1),]
```

Ora chiediamoci, quante osservazioni (punti o posizioni di pesca) abbiamo per ciascuna cella della griglia? Per rispondere, usiamo nuovamente la funzione `st_intersects`, ma questa volta non ci limiteremo a considerare solo quali `POINT` cadono all'interno dei `POLYGON`, bensì stabiliremo un'associazione diretta tra ogni elemento di `POINT` ed un elemento del `POLYGON`. Per farlo, sarà sufficiente coercire l'oggetto `inter` (generato da `st_intersects`) in un data frame.

```
# Intersezione (primo argomento points, poi grid)
inter <- st_intersects(catch_sf, grid_sf_cut)
inter
```

```
## Sparse geometry binary predicate list of length 578, where the
## predicate was `intersects`
## first 10 elements:
## 1: 387
## 2: 360
## 3: 177
## 4: 311
## 5: 275
## 6: 372
## 7: 113
## 8: 401
## 9: 362
## 10: 114
df_inter = as.data.frame(inter)

head(df_inter)
##   row.id col.id
## 1      1    387
## 2      2    360
## 3      3    177
## 4      4    311
## 5      5    275
## 6      6    372
```

Come si può vedere, *R* assegnerà automaticamente i nomi `row.id` e `col.id` al data frame così generato. Ora, ci basta sapere che `row.id` contiene gli indici dei punti `POINT` che corrispondono alle rispettive celle in `col.id`. Dunque:

```
cell = numeric(nrow(catch_sf))
cell[df_inter$row.id] = df_inter$col.id
catch_sf$cell = cell

catch_sf
## Simple feature collection with 578 features and 8 fields
## Geometry type: POINT
## Dimension:      XY
```

```
## Bounding box:  xmin: 11.558 ymin: 35.4545 xmax: 14.84467 ymax:
37.91383
## Geodetic CRS:  WGS 84
## First 10 features:
##   HAUL SPECIES      LON      LAT   KG  DEPTH      geometry
## 1      6      ARA 12.21350 36.88200 804.0 -538.00  POINT (12.2135 36.882)
## 2      7      ARA 12.41383 36.76233  76.0 -561.50  POINT (12.41383 36.76233)
## 3     14      ARA 13.36367 36.10117 243.0 -680.00  POINT (13.36367 36.10117)
## 4     15      ARA 14.00150 36.53517 103.0 -675.50  POINT (14.0015 36.53517)
## 5     18      ARA 13.60408 36.39292  37.0 -735.75  POINT (13.60408 36.39292)
## 6     19      ARA 13.80733 36.73483  48.0 -784.00  POINT (13.80733 36.73483)
## 7     21      ARA 13.39350 35.85867 303.0 -729.00  POINT (13.3935 35.85867)
## 8     25      ARA 13.82550 36.80267 140.0 -746.00  POINT (13.8255 36.80267)
## 9     26      ARA 12.66000 36.69575  60.5 -577.75  POINT (12.66 36.69575)
## 10    27      ARA 13.44817 35.85217 374.0 -718.50  POINT (13.44817 35.85217)
##   on_grid cell
## 1         1  387
## 2         1  360
## 3         1  177
## 4         1  311
## 5         1  275
## 6         1  372
## 7         1  113
## 8         1  401
## 9         1  362
## 10        1  114
```

In questo modo abbiamo aggiunto a `catch_sf` una colonna `cell`. Ma ora possiamo contare le occorrenze di ogni cella e generare un vettore di conte da assegnare alle celle di `grid_sf_cut`:

```
count_cell = table(cell[which(cell != 0)])
count = numeric(nrow(grid_sf_cut))
count[as.numeric(names(count_cell))] = count_cell
grid_sf_cut$count <- count
```

```
# Mappa del numero di punti all'interno delle celle
ggmap(map10) +
  geom_sf(data = grid_sf_cut,
    aes(fill = count),
    colour = "darkgrey",
    inherit.aes = F) +
  scale_fill_viridis_c(option = "cividis",
    trans = "log10",
    na.value = NA) +

  theme_test() +
  xlab("Longitude") +
  ylab("Latitude")

## Coordinate system already present. Adding new coordinate system,
## which will
## replace the existing one.
## Warning in scale_fill_viridis_c(option = "cividis", trans =
## "log10", na.value =
## NA): log-10 transformation introduced infinite values.
```

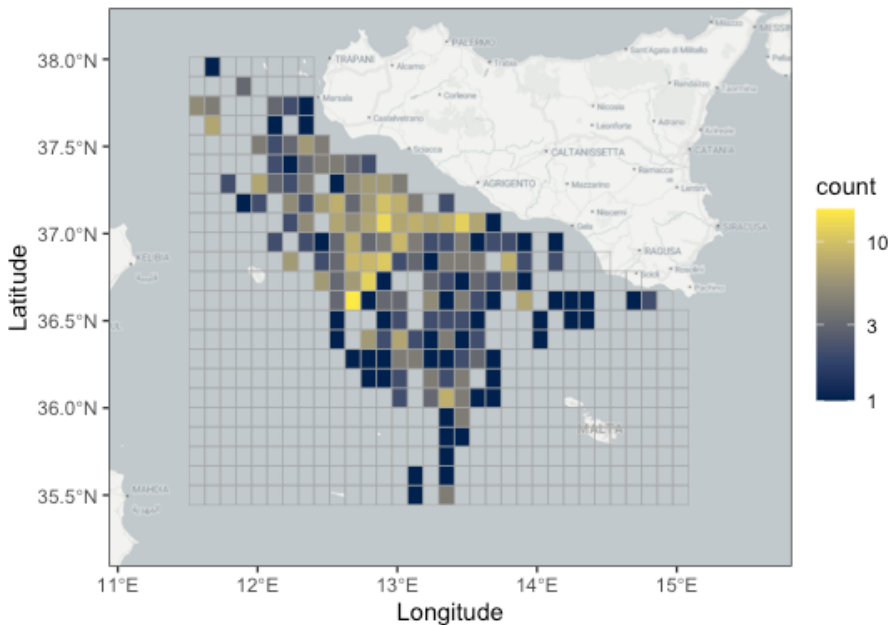


Figura 163 - La nostra griglia con le conte dei punti per cella.

A questo punto immaginiamo di essere dei “manager” e di voler classificare le aree (rappresentate dalle nostre celle) in base alla quantità di pescato che forniscono per le diverse specie (lo faremo solo per una di esse a titolo di esempio).

Uno tra i molti modi per farlo è prendere l'oggetto `catch_sf`, nel quale ogni record è già associato ad una cella, quindi calcolare, per ogni *species*, il valore totale di cattura in ogni cella, ed infine associare questi valori alle celle di `grid_sf_cut`. Per prima cosa, però, dobbiamo prenderci da `catch_sf` le sole colonne della *species*, dei kg di cattura e della cella di occorrenza. Per farlo, ci serve la funzione `st_drop_geometry`:

```
df_cell_catch = st_drop_geometry(catch_sf)[, c("SPECIES", "KG",
"cell")]
df_cell_catch = aggregate(data = df_cell_catch, KG ~ SPECIES + cell,
FUN = "sum", na.rm = TRUE)
df_cell_catch_DPS = df_cell_catch[which(df_cell_catch$SPECIES ==
"DPS"),]
DPS = numeric(nrow(grid_sf_cut))
DPS[df_cell_catch_DPS$cell] = df_cell_catch_DPS$KG
grid_sf_cut$DPS = DPS

# Mappa del numero di catture di DPS
ggmap(map10) +
  geom_sf(data = grid_sf_cut,
    aes(fill = DPS),
    colour = "darkgrey",
    inherit.aes = F) +
  scale_fill_viridis_c(option = "turbo",
    trans = "log10",
    na.value = NA) +
  theme_test() +
  xlab("Longitude") +
  ylab("Latitude")
```

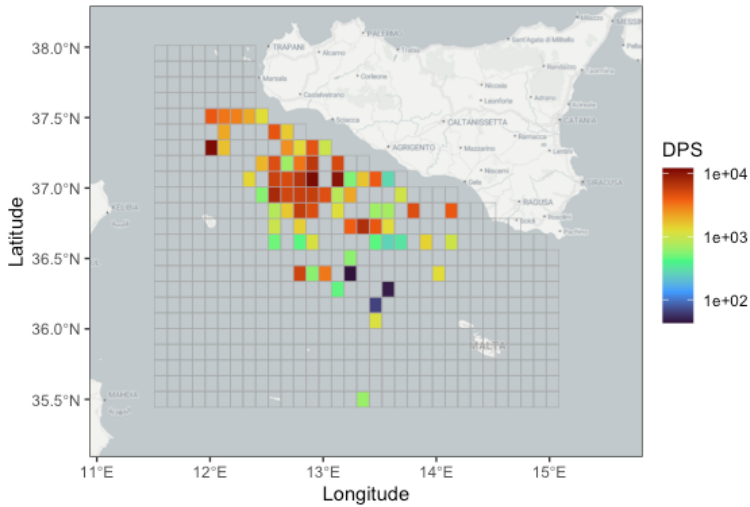


Figura 164 - La nostra griglia con le catture di DPS.

Il comportamento e l'utilizzo della funzione `st_intersects`, che abbiamo appena visto, è generalizzabile a tutta una serie di funzioni della libreria `sf`, che sono utili per svolgere una qualsivoglia operazione di *overlap* spaziale. Di fatto, la libreria `sf` fornisce un approccio ordinato, efficiente e rapido per manipolare dati spaziali in *R*. Una lista completa dei suoi comandi è riportata nelle figure seguenti.

Spatial manipulation with sf: : CHEAT SHEET

The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.



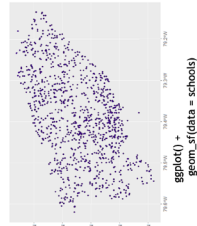
Un viaggio nell'ambiente R

Geometric confirmation

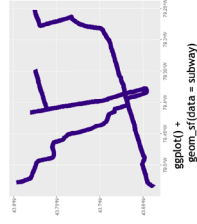
-   `st_contains(x, y, ...)` Identifies if y is within x (i.e. point within polygon)
-   `st_covered_by(x, y, ...)` Identifies if x is completely within y (i.e. polygon completely within polygon)
-   `st_covers(x, y, ...)` Identifies if any point from x is outside of y (i.e. polygon outside polygon)
-   `st_crosses(x, y, ...)` Identifies if any geometry of x have commonalities with y
-   `st_disjoint(x, y, ...)` Identifies when geometries from x do not share space with y
-   `st_equals(x, y, ...)` Identifies if x and y share the same geometry
-   `st_intersects(x, y, ...)` Identifies if x and y geometry share any space
-   `st_overlaps(x, y, ...)` Identifies if geometries of x and y share space, are of the same dimension, but are not completely contained by each other
-   `st_touches(x, y, ...)` Identifies if geometries of x and y share a common point but their interiors do not intersect
-   `st_within(x, y, ...)` Identifies if x is in a specified distance to y

Geometric operations

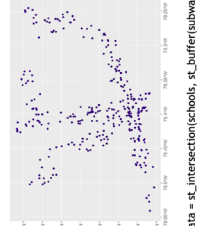
-   `st_boundary(x)` Creates a polygon that encompasses the full extent of the geometry
-   `st_buffer(x, dist, nQuads=5)` Creates a polygon covering all points of the geometry within a given distance
-   `st_centroid(x, ..., of_largest_polygon)` Creates a point at the geometric centre of the geometry
-   `st_convex_hull(x)` Creates geometry that represents the minimum convex geometry of x
-   `st_line_merge(x)` Creates linestring geometry from sewing multi linestring geometry together
-   `st_node(x)` Creates nodes on overlapping geometry where nodes do not exist
-   `st_point_on_surface(x)` Creates a point that is guaranteed to fall on the surface of the geometry
-   `st_polygonize(x)` Creates polygon geometry from linestring geometry
-   `st_segmentize(x, dfMaxLength, ...)` Creates linestring geometry from x based on a specified length
-   `st_simplify(x, preserveTopology, dTolerance)` Creates a simplified version of the geometry based on a specified tolerance



+



=>



This cheatsheet presents the sf package [Edzer Pebesma 2018] in version 0.6.3. See <https://github.com/r-spatial/sf> for more details.

CC BY Ryan Garnett <http://github.com/ryangarnett>
<https://creativecommons.org/licenses/by/4.0/>

Figura 165 - La sheet dei comandi di sf (!)

Spatial manipulation with sf: : CHEAT SHEET

The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.



Geometry operations

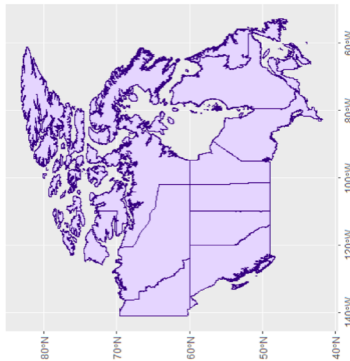
-  `st_contains(x, y, ...)` Identifies if *y* is within *x* (i.e. point within polygon)
-  `st_crop(x, y, ..., xmin, ymin, xmax, ymax)` Creates geometry of *x* that intersects a specified rectangle
-  `st_difference(x, y)` Creates geometry from *x* that does not intersect with *y*
-  `st_intersection(x, y)` Creates geometry of the shared portion of *x* and *y*
-  `st_sym_difference(x, y)` Creates geometry representing portions of *x* and *y* that do not intersect
-  `st_snap(x, y, tolerance)` Snap nodes from geometry *x* to geometry *y*
-  `st_union(x, y, ..., by_feature)` Creates multiple geometries into a single geometry, consisting of all geometry elements

Geometric measurement

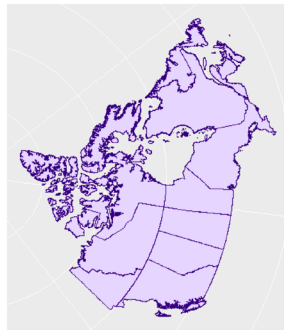
- `st_area(x)` Calculate the surface area of a polygon geometry based on the current coordinate reference system
- `st_distance(x, y, ..., dist_fun, by_element, which)` Calculates the 2D distance between *x* and *y* based on the current coordinate system
- `st_length(x)` Calculates the 2D length of a geometry based on the current coordinate system

Misc operations

- `st_as_sf(x, ...)` Create a sf object from a non-geospatial tabular data frame
- `st_cast(x, to, ...)` Change *x* geometry to a different geometry type
- `st_coordinates(x, ...)` Creates a matrix of coordinate values from *x*
- `st_crs(x, ...)` Identifies the coordinate reference system of *x*
- `st_join(x, y, join, FUN, suffix, ...)` Performs a spatial left or inner join between *x* and *y*
- `st_make_grid(x, cellsize, offset, n, crs, what)` Creates rectangular grid geometry over the bounding box of *x*
- `st_nearest_feature(x, y)` Creates an index of the closest feature between *x* and *y*
- `st_nearest_points(x, y, ...)` Returns the closest point between *x* and *y*
- `st_read(dsn, layer, ...)` Read file or database vector dataset as a sf object
- `st_transform(x, crs, ...)` Convert coordinates of *x* to a different coordinate reference system



```
ggplot() +
  geom_sf(data = cdm) +
  coord_sf(crs = st_crs(4326))
```



```
ggplot() +
  geom_sf(data = cdm) +
  coord_sf(crs = st_crs(3347))
```

This cheatsheet presents the sf package (Edzer Pebesma 2018) in version 0.6.3. See <https://github.com/r-spatial/sf> for more details.

CC BY Ryan Garnett <http://github.com/ryangarnett>
<https://creativecommons.org/licenses/by/4.0/>

Figura 166 - La sheet dei comandi di sf (11).

La libreria terra per i dati raster

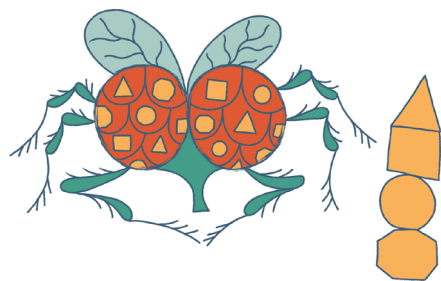


Figura 167 - Il formato raster corrisponde a quello usato dagli organismi, che usano cellule specializzare per catturare e scomporre le immagini.

Il modello di dati spaziali raster rappresenta il mondo con una griglia continua di celle (spesso chiamate anche *pixel*). Questo modello di dati si riferisce spesso alle cosiddette griglie regolari, in cui ogni cella ha la stessa dimensione costante, e in questo libro ci concentreremo solo sulle griglie regolari. Il modello di dati raster consiste solitamente in un'intestazione raster e in una matrice (con righe e colonne) che rappresenta celle equidistanti (spesso chiamate anche *pixel*). L'intestazione raster definisce il sistema di riferimento delle coordinate, l'estensione e l'origine. L'origine (o punto di partenza) è spesso la coordinata dell'angolo inferiore sinistro della matrice (il pacchetto terra, tuttavia, utilizza l'angolo superiore sinistro, per impostazione predefinita). L'intestazione definisce l'estensione attraverso il numero di colonne, il numero di righe e la risoluzione delle dimensioni delle celle.

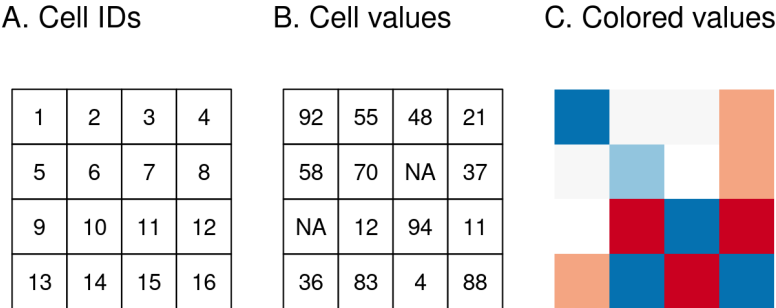


Figura 168 - Rappresentazione dei tipi di dati raster. Fonte: <https://r.geocompx.org/spatial-class#fig:raster-intro-plot>.

Le mappe raster di solito rappresentano fenomeni continui come l'altitudine, la temperatura, la densità di popolazione o i dati spettrali. Anche le caratteristiche discrete, come le classi di suolo o di copertura del suolo, possono essere rappresentate nel modello di dati raster. I confini delle caratteristiche discrete possano diventare sfumati nei set di dati raster. A seconda della natura dell'applicazione, le rappresentazioni vettoriali delle caratteristiche discrete possono essere più adatte.

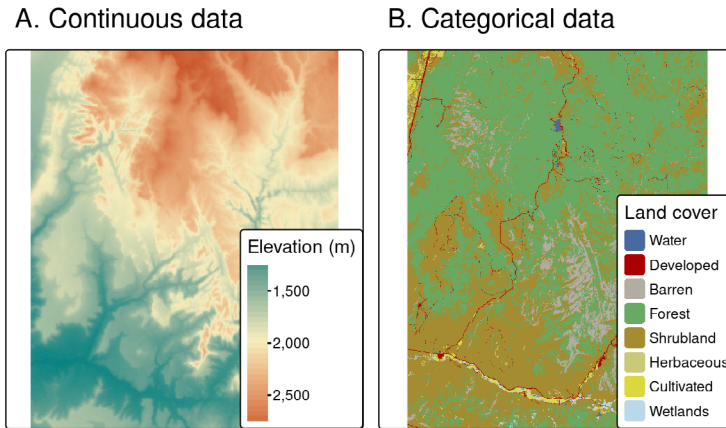


Figura 169 - Esempi di mappe raster di dati continui o categorici. Fonte: <https://r.geocompx.org/spatial-class#fig:raster-intro-plot>.

Negli ultimi due decenni sono stati sviluppati diversi pacchetti per la lettura e l'elaborazione di set di dati raster. Il principale è stato **raster**, che ha portato ad un cambio di passo quando è stato lanciato nel 2010 ed è stato il primo pacchetto in questo settore fino allo sviluppo di **terra** e **stars**. Entrambi i pacchetti, sviluppati di recente, forniscono funzioni potenti e performanti per lavorare con insiemi di dati raster e vi è una sostanziale sovrapposizione tra i loro possibili utilizzi. Qui ci concentreremo su **terra**, che sostituisce il più vecchio e (nella maggior parte dei casi) più lento **raster**.

Comunque è bene sapere che **terra** si concentra sul modello di dati raster più comune (griglie regolari), mentre **stars** consente di memorizzare anche modelli meno diffusi (tra cui griglie regolari, ruotate, tagliate, rettilinee e curvilinee). Mentre **terra** gestisce solitamente raster a uno o più livelli, il pacchetto **stars** fornisce modi per memorizzare cubi di

dati raster – un oggetto raster con molti livelli (ad esempio, bande), per molti momenti nel tempo (ad esempio, mesi) e molti attributi (ad esempio, tipo di sensore A e tipo di sensore B). In entrambi i pacchetti, tutti i livelli o gli elementi di un cubo di dati devono avere le stesse dimensioni spaziali e la stessa estensione.

In secondo luogo, entrambi i pacchetti consentono di leggere tutti i dati raster in memoria o di leggerne solo i metadati; di solito ciò avviene automaticamente in base alle dimensioni del *file* di *input*. Tuttavia, memorizzano i valori dei raster in modo molto diverso. **terra** è basato su codice C++ e utilizza principalmente puntatori C++. **stars** memorizza i valori come liste di array per i raster più piccoli o solo un percorso di *file* per quelli più grandi.

In terzo luogo, le funzioni di **stars** sono strettamente correlate agli oggetti e alle funzioni vettoriali di **sf**, mentre **terra** utilizza una propria classe di oggetti per i dati vettoriali, ovvero **SpatVector**, ma accetta anche quelli di **sf**.

In quarto luogo, entrambi i pacchetti hanno un approccio diverso per quanto riguarda il modo in cui le varie funzioni lavorano sui loro oggetti. Il pacchetto **terra** si affida principalmente ad un gran numero di funzioni incorporate, dove ogni funzione ha uno scopo specifico (ad esempio, il ricampionamento o il ritaglio). **stars** invece utilizza alcune funzioni integrate (di solito con nomi che iniziano con **st_**), alcune funzioni esistenti di **dplyr** (ad esempio, **filter()** o **slice()**) e ha anche i propri metodi per le funzioni esistenti di **R** (ad esempio, **split()** o **aggregate()**).

È importante notare che è semplice convertire gli oggetti da **terra** a **stars** (usando **st_as_stars()**) e viceversa (usando **rast()**).

Il pacchetto **terra** fornisce un ampio insieme di funzioni per creare, leggere, esportare, manipolare ed elaborare insiemi di dati raster. Le funzionalità di **terra** sono in gran parte le stesse del più maturo pacchetto **raster**, ma ci sono alcune differenze: le funzioni di **terra** sono solitamente più efficienti dal punto di vista computazionale rispetto alle equivalenti di **raster**. D'altra parte, il sistema di classi **raster** è popolare ed utilizzato da molti altri pacchetti. È possibile passare da un tipo di oggetto all'altro senza problemi per garantire la retrocompatibilità con i vecchi *script* e pacchetti, ad esempio con le funzioni **raster()**, **stack()** e **brick()** del pacchetto **raster** (si veda il capitolo precedente

per maggiori informazioni sull'evoluzione dei pacchetti *R* per il lavoro con i dati geografici).

Oltre alle funzioni per la manipolazione dei dati raster, **terra** fornisce molte funzioni di basso livello che possono costituire la base per lo sviluppo di nuovi strumenti per lavorare con i *dataset* raster. **terra** consente anche di lavorare su *dataset* raster di grandi dimensioni che sono troppo grandi per essere inseriti nella memoria principale. In questo caso, **terra** offre la possibilità di dividere il raster in parti più piccole, che vengono elaborate iterativamente invece di caricare l'intero *file* raster nella RAM.

Come prima cosa, possiamo imparare a generare *ex novo* un oggetto raster. Ci sono essenzialmente due modi per farlo:

- usando la funzione **rast**, cioè in assoluto la funzione di base e più generica del pacchetto **terra**. Questa funzione permette di generare, da zero o da un altro oggetto già esistente, uno **SpatRaster**, cioè una superficie riferita allo spazio e divisa in celle tridimensionali (righe, colonne e livelli). Un vantaggio computazionale è che, quando uno **SpatRaster** viene creato da un *file*, non carica i valori delle celle (*pixel*) nella memoria (RAM). Legge solo i parametri che descrivono la geometria dello **SpatRaster**, come il numero di righe e colonne e il sistema di riferimento delle coordinate. I valori effettivi verranno letti quando necessario;
- usando la funzione **rasterFromXYZ** del pacchetto **raster**, molto simile a **rast**, che permette di trasferire i valori associati alle geometrie di dati vettoriali a un raster (in sostanza di coercire e trasformare un dato spaziale da vettoriale in raster).

Vediamo degli esempi:

```
library(terra)
my_rast <- rast(nrows = 10, ncols = 10,
               vals = rnorm(100, 0, 3),
               names = "variable",
               xmin = 0, xmax = 100)
head(my_rast)
##      variable
## 1 -2.8380906
## 2  0.3865201
```



```
## 3 -4.5825380
## 4 -1.5662647
## 5 -1.8658151
## 6 -3.2493770
plot(my_rast)
```

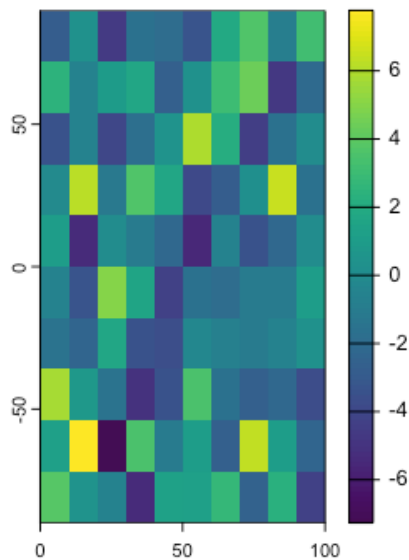


Figura 170 - Il plot del nostro raster.

Nella figura precedente c'è un semplice raster formato da 100 celle, disposte su 10 righe e 10 colonne, a cui abbiamo associato 100 valori estratti da una distribuzione normale avente media 0 e varianza 9. Ci sono alcune opzioni che non abbiamo usato, la più importante delle quali è l'argomento `crs`, il cui uso è però molto intuitivo.

Adesso facciamo conoscenza anche con `rasterFromXYZ`:

```
library(raster)
longs = seq(12, 15, by = 0.5)
lats = seq(35, 38, by = 0.5)
coo = expand.grid(longs, lats)
colnames(coo) = c("long", "lat")
coo$variable = rnorm(nrow(coo), 0, 3)
```

```

coo_rast <- rasterFromXYZ(coo)
coo_rast
## class      : RasterLayer
## dimensions : 7, 7, 49  (nrow, ncol, ncell)
## resolution : 0.5, 0.5  (x, y)
## extent     : 11.75, 15.25, 34.75, 38.25  (xmin, xmax, ymin, ymax)
## crs        : NA
## source     : memory
## names      : variable
## values     : -7.563709, 11.76423  (min, max)
plot(coo_rast)

```

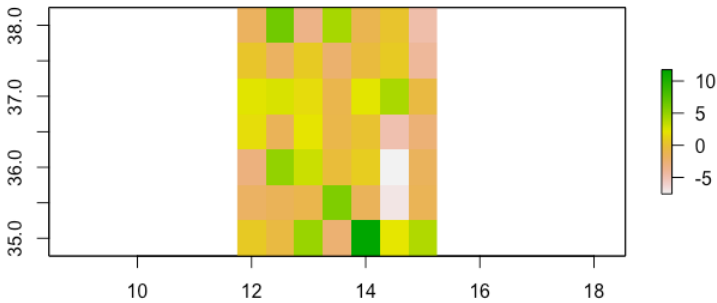


Figura 171 - Un altro plot del nostro raster.

Anche in questo caso è semplice capire cosa abbiamo fatto, ma è fondamentale fornire un importante avvertimento: `rasterFromXYZ` accetterà di fare il suo lavoro solo se le coordinate che gli forniamo sono disposte in modo da formare una griglia regolare... Infatti:

```

library(terra)
catch2rast <- unique(catch[, c("LON", "LAT", "DEPTH")])
head(catch2rast)
##      LON      LAT  DEPTH
## 1 12.21350 36.88200 -538.00
## 2 12.41383 36.76233 -561.50
## 3 13.36367 36.10117 -680.00
## 4 14.00150 36.53517 -675.50

```

```
## 5 13.60408 36.39292 -735.75
## 6 13.80733 36.73483 -784.00
catch_rast <- rasterFromXYZ(catch2rast)
## Error in rasterFromXYZ(catch2rast): x cell sizes are not regular
```

Abbiamo ottenuto un errore perché un raster non può essere definito in modo da avere una struttura irregolare.

Vediamo in figura un esempio di mappa raster con dati di catture:

```
library(sf)

catch2rast = st_as_sf(x = catch[, c("LON", "LAT", "KG")],
                      coords = c("LON", "LAT"),
                      crs = 4326)

v_x <- vect(catch2rast)

catch_rast = rasterize(x = v_x,
                       y = rast("https://github.com/tommaso-russo/LibroR2/raw/main/
Spatial/GSA16_Depth2.tiff"),
                       field = "KG",
                       fun = "mean", na.rm = T)

plot(catch_rast)
```

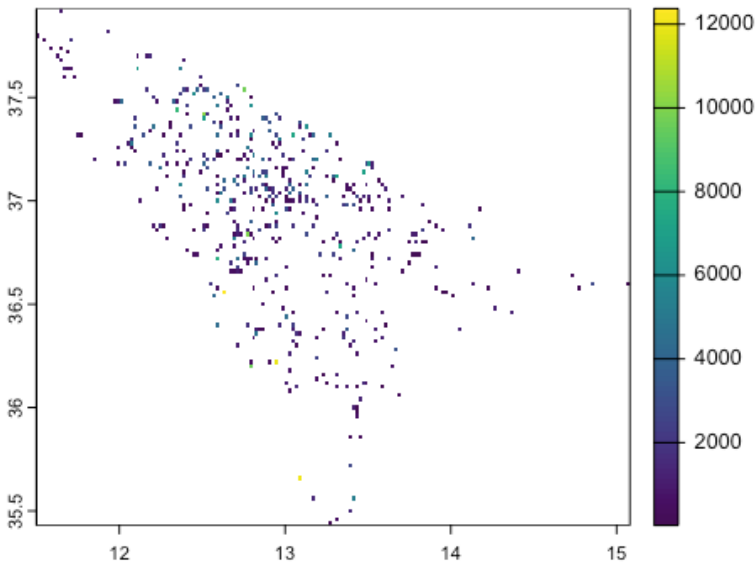


Figura 172- Il raster delle catture.

Dunque, fatte le presentazioni usando esempi astratti, per cominciare a fare pratica davvero scarichiamo un raster (in formato *tiff*) della batimetria di una zona che comprende ampiamente i dati sui quali abbiamo già lavorato (le catture nella GSA16) e un'ampia porzione dell'isola stessa. Ovviamente useremo il pacchetto *terra* e, più nello specifico, la funzione *rast*.

```
depth16 = rast("https://github.com/tommaso-russo/LibroR2/raw/main/
Spatial/GSA16_Depth2.tiff")
class(depth16)
## [1] "SpatRaster"
## attr(,"package")
## [1] "terra"
depth16
## class      : SpatRaster
## dimensions : 125, 179, 1  (nrow, ncol, nlyr)
## resolution : 0.02, 0.02  (x, y)
```

```
## extent      : 11.5, 15.08, 35.43, 37.93 (xmin, xmax, ymin, ymax)
## coord. ref.  : lon/lat WGS 84
## source      : GSA16_Depth2.tiff
## name        : GSA16_Depth2
## min value    : -1813.688
## max value    : 2825.767
```

Come possiamo vedere da questa anteprima, ci troviamo di fronte a una situazione simile a quella vista con `sf`. Ci vengono mostrati:

- la classe dell'oggetto,
- le sue dimensioni: 125 righe, 179 colonne (e quindi un totale di 22375 pixels) e 1 colonna di valori,
- la risoluzione (0.02 gradi),
- gli estremi del dominio spaziale,
- il *CRS*,
- il suo nome e il nome del campo,
- i valori minimo e massimo del campo valori (cioè, in questo caso, la profondità).

Se plottiamo l'oggetto otterremo:

```
plot(depth16, col = topo.colors(10))
```

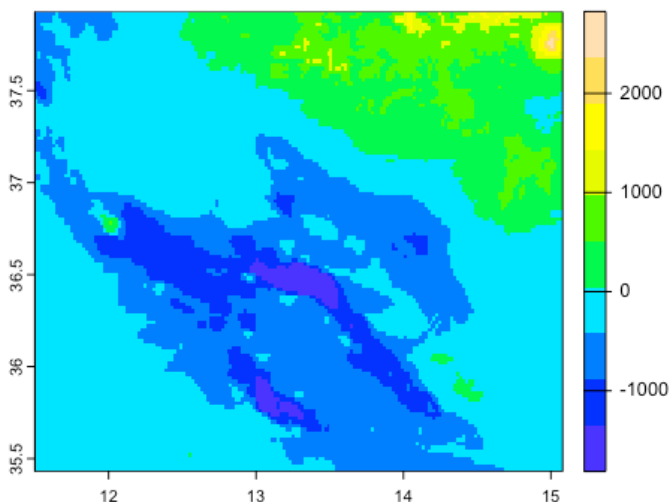


Figura 173 - Il raster della profondità.

Una prima operazione semplice ma opportuna potrebbe essere, visto che siamo sempre nell'ambito marino, di eliminare la porzione del raster riferito alle terre emerse. Questo ci consentirà, poi, di visualizzare il raster sovrapponendolo ad una mappa geografica. Per farlo, usiamo la funzione `clamp` e definiamo un nuovo oggetto denominato `depth16_sea`:

```
depth16_sea <- clamp(depth16, upper = 0, value=FALSE)
```

La funzione `clamp`, come possiamo verificare dal suo `help`, ci consente di selezionare le celle di un raster in un certo *range* (definito dagli argomenti `lower` e `upper`) del suo campo di valori (non di coordinate, attenzione!). L'aspetto del nuovo raster è:

```
plot(depth16_sea, col = topo.colors(10))
```

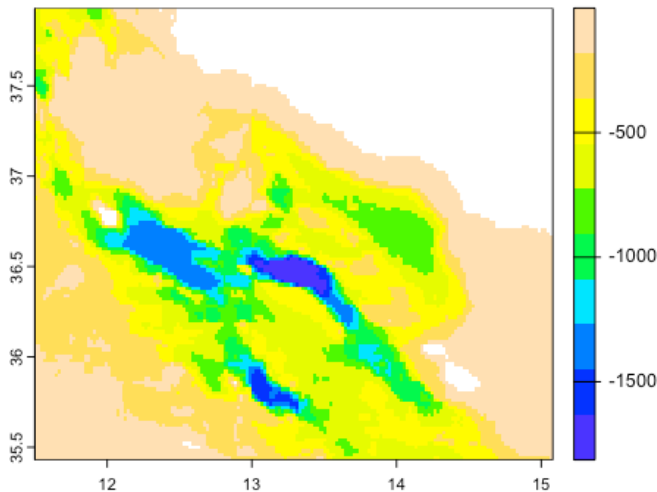


Figura 174 - Un'altra visualizzazione del raster della profondità.

Ora, in maniera analoga a quanto avevamo fatto con i vettori in `sf`, aggiungiamo i riferimenti topografici e la terraferma. Per farlo, possiamo generare un apposito oggetto raster mediante la funzione `get_tiles` del pacchetto `maptiles`.

```
library(maptiles)
bg <- get_tiles(ext(depth16))
plotRGB(bg)
plot(depth16_sea, col = topo.colors(10), add = TRUE)
```

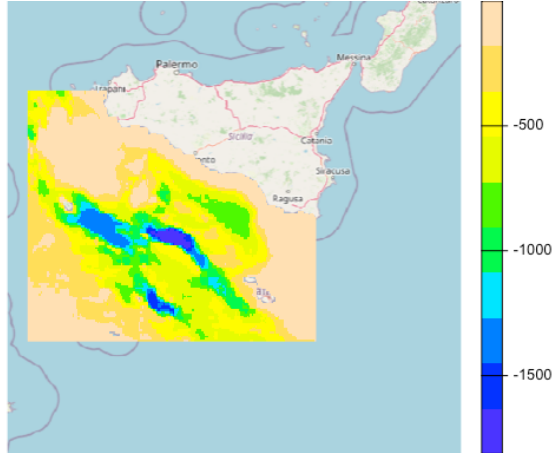


Figura 175 - Il raster della profondità con i riferimenti geografici.

Con pochi comandi abbiamo già ottenuto una visualizzazione decente dei nostri dati. Il raster che abbiamo appena usato contiene un campo numerico (la profondità). Adesso proviamo a usare un raster che contenga, per ogni cella, la codifica di una variabile qualitativa. Continuando a esplorare le coste meridionali della Sicilia, useremo un raster che contiene i dati relativi alla tipologia di habitat sul fondale, secondo la classificazione EUNIS. Carichiamo il raster e guardiamo cosa contiene:

```
library(terra)
eunis16 = rast("https://github.com/tommaso-russo/LibroR2/raw/main/
Spatial/EUNIS_substrate_GSA16.tif")
class(eunis16)
## [1] "SpatRaster"
## attr(,"package")
## [1] "terra"
eunis16
## class      : SpatRaster
## dimensions : 416, 479, 1 (nrow, ncol, nlyr)
```

```
## resolution : 1000, 1000 (x, y)
## extent      : 1224360, 1703360, 4163874, 4579874 (xmin, xmax,
ymin, ymax)
## coord. ref. : WGS 84 / Pseudo-Mercator (EPSG:3857)
## source      : EUNIS_substrate_GSA16.tif
## name        : EUNIS_substrate_GSA16
plot(eunis16, col = rev(rainbow(12)))
```

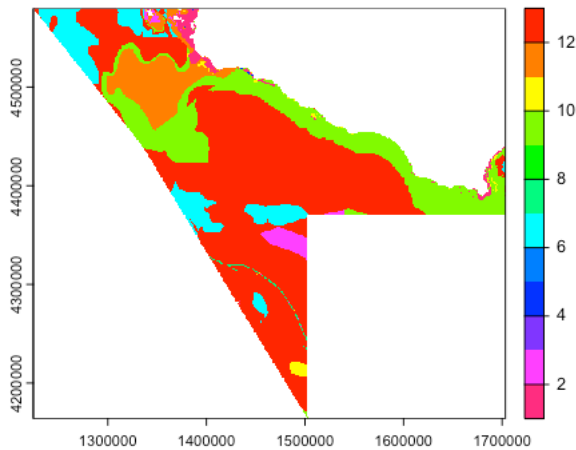


Figura 176 - Il raster degli habitat del fondale.

Notiamo che, rispetto alla profondità, sono cambiate diverse cose:

- la risoluzione (molto maggiore rispetto al raster della profondità, come si nota dalle linee molto smussate),
- il CRS: sebbene sia ancora WGS84, questa volta è proiettato e infatti cambia anche...
- ...il dominio (*extent*).

Un'altra cosa da notare è che la tipologia di substrato è stata codificata con un codice numerico, e infatti *R* ce la rappresenta come tale. Ci proponiamo dunque di manipolare questo raster per uniformare il CRS a quello della profondità e per cambiare la scala in modo che contenga i nomi delle diverse tipologie di substrato. Per farlo possiamo usare direttamente il nostro raster "preferito" o di riferimento, in questo caso *depth16*:


```
eunis16 <- project(eunis16, depth16)
plot(eunis16, col = rev(rainbow(12)))
```

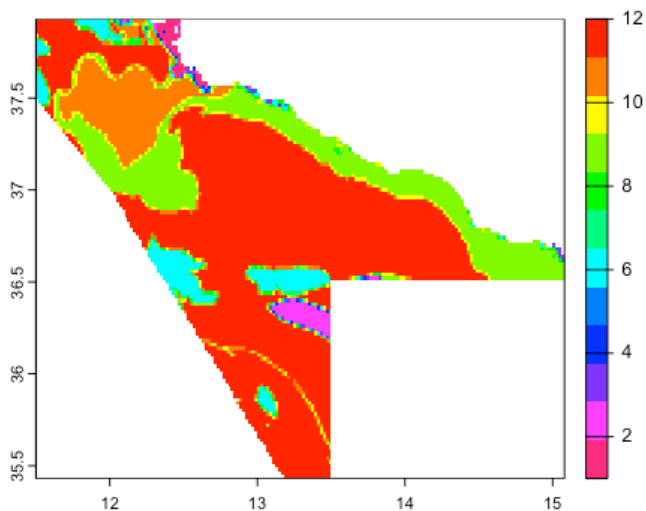


Figura 177 - Ancora il raster degli habitat del fondale.

Per completezza, diciamo che avremmo potuto usare anche un modo alternativo, rappresentato dalla funzione `crs` di `terra`.

L'operazione di proiezione, comunque, attuata mediante la funzione `project`, ha cambiato anche la risoluzione del raster EUNIS. Infatti:

```
eunis16
## class      : SpatRaster
## dimensions  : 125, 179, 1  (nrow, ncol, nlyr)
## resolution  : 0.02, 0.02  (x, y)
## extent     : 11.5, 15.08, 35.43, 37.93  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84
## source(s)   : memory
## varname     : GSA16_Depth2
## name       : EUNIS_substrate_GSA16
## min value   :          1
## max value   :          12
```

Come si vede, adesso i due raster sono completamente sovrapponibili!

```

depth16_sea
## class      : SpatRaster
## dimensions : 125, 179, 1  (nrow, ncol, nlyr)
## resolution : 0.02, 0.02  (x, y)
## extent     : 11.5, 15.08, 35.43, 37.93  (xmin, xmax, ymin, ymax)
## coord. ref.: lon/lat WGS 84
## source(s)  : memory
## varname    : GSA16_Depth2
## name       : GSA16_Depth2
## min value  :   -1813.688
## max value  :    -0.015

```

Esistono altre funzioni simili a `project` che lavorano su due raster e ci permettono, ad esempio, di ritagliare un raster usando come riferimento un altro raster: la funzione `crop`.

Altre funzioni di cui non mostreremo esempio ma sono veramente intuitive sono:

- `mask`, che permette di generare un nuovo raster escludendo valori indesiderati (esempio gli `NA`);
- `trim`, che permette di rimuovere le righe e le colonne esterne che sono `NA` o un altro valore.

Adesso dobbiamo mettere mano al campo dei valori. Per farlo abbiamo bisogno di associare a ciascuno dei valori numerici (da 1 a 12) i corrispondenti nomi degli habitat. Infatti l'operazione di ridefinizione dei valori del raster può essere facilmente effettuata usando la funzione `values`, ma prima dobbiamo definire un data frame che definisca le corrispondenze tra i codici numerici e i substrati corrispondenti. Quindi:

```

eunis16 <- clamp(eunis16, lower = 1, upper = 12, value=FALSE)

hab_codes = data.frame(v = c(1:12),
                        hab = c("Posidonia oceanica meadows",
                               "Coarse and mixed sediment",
                               "Coralligenous platforms",
                               "Dead mattes of Posidonia oceanica",

```

```

      "Eurythermal lagoon biocenosis",
      "Fine mud",
      "Fine mud or Sandy mud or Muddy sand",
      "Mixed sediment",
      "Muddy sand",
      "Rock or other hard substrata",
      "Sand",
      "Sandy mud"))

values(eunis16) <- round(values(eunis16), 0)
values(eunis16) <- hab_codes$hab[match(values(eunis16), hab_
codes$v)]
plot(eunis16, col = rev(rainbow(12)))

```

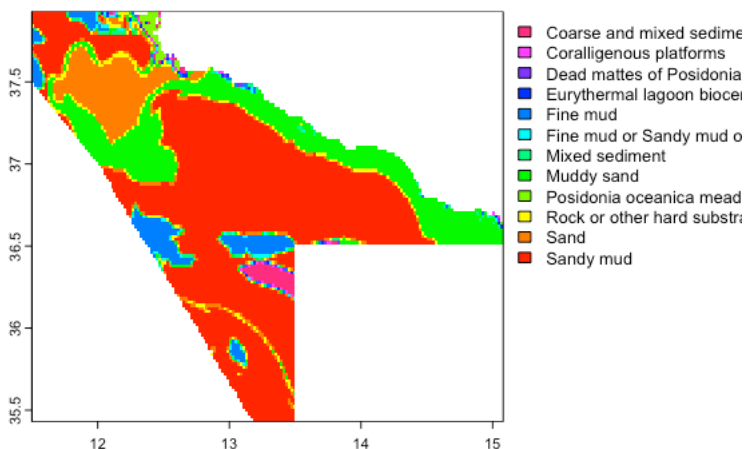


Figura 178 - Gli habitat del fondale con una legenda fatta bene.

A questo punto introduciamo un aspetto importante, e cioè la modalità con la quale si possono estrarre, a partire da un raster, i valori corrispondenti ad alcuni punti specifici all'interno di esso.

Prima di tutto impariamo a generare un insieme di punti "a caso" usando la funzione `spatSample`. Si tratta di una funzione molto utile perché consente di applicare 4 diverse metodologie:

- `regular`: punti regolari all'interno del dominio;

- **random**: punti casuali all'interno del dominio;
- **stratified**: un punto per ognuno degli strati all'interno del dominio;
- **weights**: associando un peso ovvero una probabilità di campionamento.

Vediamo degli esempi:

10 punti regolari

```
xy <- spatSample(eunis16, 10, "regular", na.rm = TRUE, xy = TRUE)
plot(eunis16, col = rev(rainbow(12)))
points(xy, pch=20, col="black", cex=2)
```

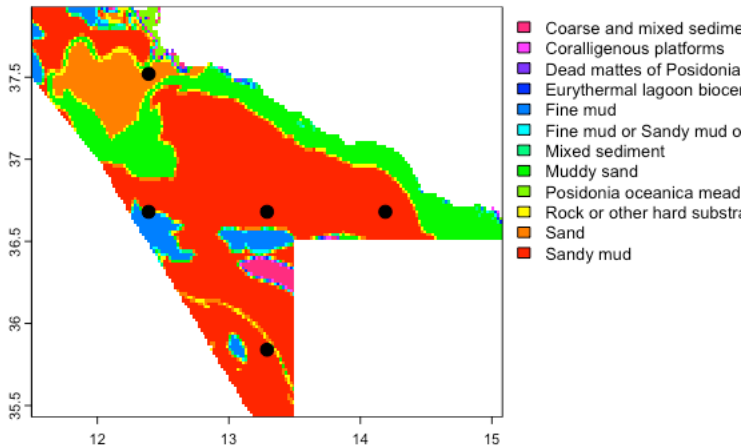


Figura 179 - Punti regolari sul raster.

Nel codice c'è scritto «10 punti regolari», ma in figura vediamo solo 5 punti neri... dove sono finiti gli altri 5?

Vediamo la figura e:

10 punti a caso

```
xy <- spatSample(eunis16, 10, "random", na.rm = TRUE, xy = TRUE)
plot(eunis16, col = rev(rainbow(12)))
points(xy, pch=20, col="black", cex=2)
```

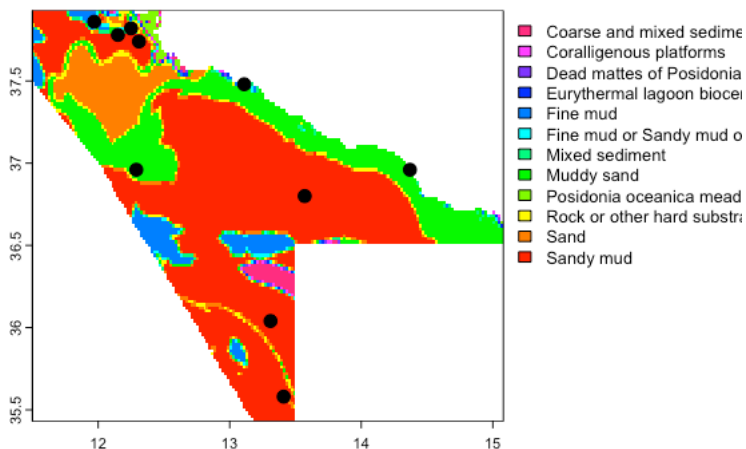


Figura 180 - Punti a caso sul raster.

```
# 10 punti a caso per ciascuna tipologia di habitat
xy <- spatSample(eunis16, 10, "stratified", na.rm = TRUE, xy = TRUE)
## Warning in cbind(cell = cells, values(x), abs(cos(pi *
values(init(x,
## "y"))/360))): number of rows of result is not a multiple of
vector length (arg
## 1)
plot(eunis16, col = rev(rainbow(12)))
points(xy, pch=20, col="black", cex=1)
```

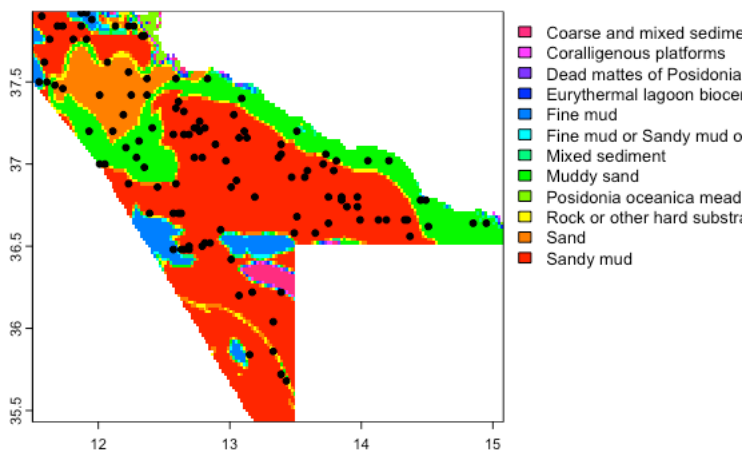


Figura 181 - Punti random-stratificati sul raster.

In tutti questi esempi abbiamo usato la funzione `spatSample` per definire un oggetto che abbiamo chiamato `xy`. Se adesso guardiamo la sua struttura...

```
head(xy)
##           x           y EUNIS_substrate_GSA16
## 1 14.13 36.66           1
## 2 14.19 36.66           1
## 3 14.95 36.64           1
## 4 14.85 36.64           1
## 5 12.57 36.70           1
## 6 12.85 36.52           1
```

... ci accorgiamo che esso contiene già, oltre alle coordinate, anche il valore corrispondente.

Questo, però, non risolve tutte le nostre potenziali esigenze. Immaginiamo, infatti, di voler conoscere il valore di un raster in corrispondenza di punti specifici che non sono scelti dal raster stesso ma imposti "dall'esterno". Immaginiamo, ad esempio, di prendere le coordinate di alcuni punti in cui sono state pescate le specie di nostro interesse (ci concentreremo sullo scampo perché ci piace il cocktail di scampi). Quindi:

```
# Estraiamo tutte le posizioni di pesca per lo scampo...
catch_NEP = catch[which(catch$SPECIES == "NEP"),]
catch_NEP = catch_NEP[, c("LON", "LAT", "KG")]
colnames(catch_NEP) = c("long", "lat", "NEPcatch")

# Estraiamo un campione di esse
sample_catch_NEP = catch_NEP[sample(1:nrow(catch_NEP), 10),]
sample_catch_NEP
##           long           lat NEPcatch
## 531 13.27175 36.64658 263.5000
## 611 12.75158 37.04042 742.0000
## 574 12.25883 36.87617 2217.0000
## 567 12.72617 36.66883  99.0000
```

```
## 603 13.06750 36.09092 1479.0000
## 572 11.67917 37.64433 1104.0000
## 592 12.80511 36.82006 1430.6667
## 597 12.93050 36.56628 265.6667
## 552 12.61472 37.37111 164.0000
## 536 13.77400 36.74567 51.0000
```

A questo punto, usiamo la funzione `extract`:

```
sample_catch_NEP$Depth =
  extract(depth16_sea, sample_catch_NEP[, c("long", "lat")])$GSA16_
Depth2
```

```
sample_catch_NEP
##          long          lat NEPcatch      Depth
## 531 13.27175 36.64658 263.5000 -481.9400
## 611 12.75158 37.04042 742.0000 -385.6250
## 574 12.25883 36.87617 2217.0000 -477.1200
## 567 12.72617 36.66883 99.0000 -689.6245
## 603 13.06750 36.09092 1479.0000 -512.1900
## 572 11.67917 37.64433 1104.0000 -377.3075
## 592 12.80511 36.82006 1430.6667 -95.8100
## 597 12.93050 36.56628 265.6667 -1084.2557
## 552 12.61472 37.37111 164.0000 -164.4325
## 536 13.77400 36.74567 51.0000 -785.9350
```

Come si può vedere, la funzione `extract` vuole solo 2 argomenti: il raster da cui estrarre e una matrice (o data frame) con le coordinate dei punti rispetto ai quali fare l'estrazione. Quindi, allo stesso modo:

```
sample_catch_NEP$Habitat =
  extract(eunis16, sample_catch_NEP[, c("long", "lat")])$EUNIS_
substrate_GSA16
```

```
sample_catch_NEP
##          long          lat NEPcatch      Depth      Habitat
```

##	531	13.27175	36.64658	263.5000	-481.9400	Sandy mud
##	611	12.75158	37.04042	742.0000	-385.6250	Sandy mud
##	574	12.25883	36.87617	2217.0000	-477.1200	Sandy mud
##	567	12.72617	36.66883	99.0000	-689.6245	Sandy mud
##	603	13.06750	36.09092	1479.0000	-512.1900	Sandy mud
##	572	11.67917	37.64433	1104.0000	-377.3075	Sandy mud
##	592	12.80511	36.82006	1430.6667	-95.8100	Sandy mud
##	597	12.93050	36.56628	265.6667	-1084.2557	Muddy sand
##	552	12.61472	37.37111	164.0000	-164.4325	Sandy mud
##	536	13.77400	36.74567	51.0000	-785.9350	Sandy mud

Con questo abbiamo concluso la nostra panoramica degli aspetti tecnici di base... siamo pronti ad entrare nel mondo della Geostatistica.

Un po' di Geostatistica

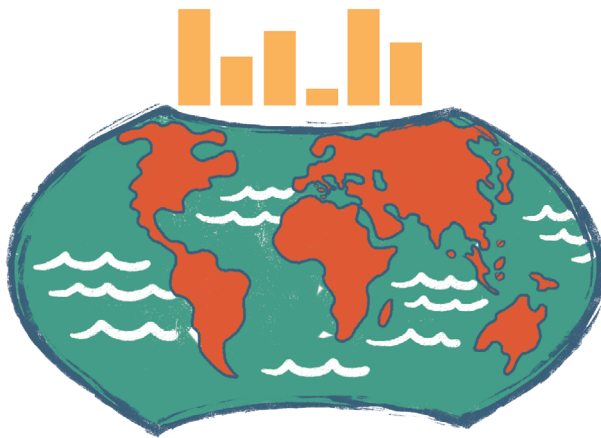


Figura 182 - La geostatistica è la scienza che “fa i conti con i dati spaziali”.

Dobbiamo fare una piccola digressione teorica, senza alcuna pretesa di affrontare in maniera “seria” l’argomento, ed introdurre i concetti di **varianza** e **covarianza** spaziali. Prendiamo una **Variabile regionalizzata (VR)**, cioè una qualsiasi proprietà che è possibile definire numericamente (come la profondità, ad esempio) ed il cui valore dipende dalla localizzazione. La localizzazione di una VR può essere normalmente

espressa mediante le sue coordinate spaziali, e la *VR* stessa si presenta strutturata spazialmente. Diciamo (per adesso) che *VR* è una variabile puramente deterministica e che, da un punto di vista matematico, può essere definita come una funzione $z(x)$, che assume in ogni punto x dello spazio un determinato valore numerico.

Per illustrare questo concetto, ritorniamo al raster della profondità nella GSA 16. Estraiamo il valore della profondità lungo un transetto che attraversi tutta l'area in corrispondenza del valore di latitudine uguale a 36.5:

```
library(ggplot2)
longs = seq(11.5, 15, by = 0.025)
lats = rep(36.5, length(longs))
xy = data.frame(Lon = longs, Lat = lats)
plot(depth16_sea, col = topo.colors(10))
points(xy, pch = 19, col = "black", cex = 0.2)
```

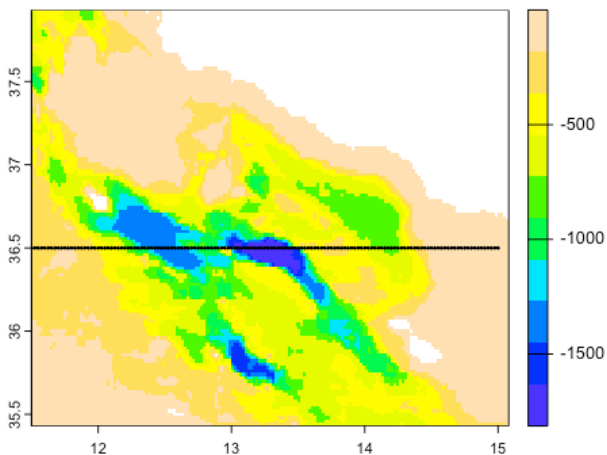


Figura 183 - Un transetto sul raster della profondità.

Come si vede dalla mappa, questo transetto di punti attraversa aree più o meno profonde, e la *VR* che ne otteniamo ha l'andamento seguente:

```
library(ggplot2)
xy$Depth = extract(depth16_sea, xy)$GSA16_Depth2
```

```
ggplot(data = xy, aes(x = Lon, y = Depth)) +
  geom_path() +
  theme_minimal()
```

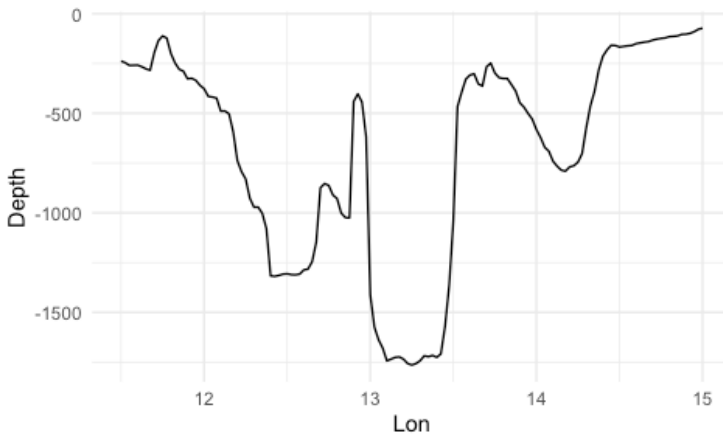


Figura 184 - Il nostro primo correlogramma.

Il grafico ci mostra, da una parte, un andamento irregolare alla piccola scala, e dall'altra, una variabilità strutturata, cioè una variabilità che sembra ubbidire a delle regole. Vi si notano infatti tratti con elevati valori della profondità e tratti con valori medi e bassi. Pertanto, siti in vicinanza dei tratti ad alto valore avranno una elevata probabilità di avere una profondità elevata, mentre avranno una media e bassa probabilità i siti negli altri tratti.

Adesso consideriamo una coppia di punti lungo questo transetto. Poniamo che i punti siano a una distanza h , dove h è un vettore, e definiamoli quindi x e $x + h$. In corrispondenza dei due punti la VR assume i valori $z(x)$ e $z(x+h)$. È possibile immaginare, in questo modo, di definire una lunga serie di "coppie" a seconda di dove poniamo x e di quale è il valore di h . Dopo aver annotato per un certo numero di volte i valori di $z(x)$ e $z(x+h)$, possiamo usarli per costruire uno *scatterplot* (grafico a dispersione) in cui i valori di $z(x)$ sono l'ascissa e quelli di $z(x+h)$ l'ordinata.

Per esercitarci e dimostrare la funzionalità di questo approccio, scegliamo 3 valori di h e scegliamo 30 coppie di punti definendo altrettanti valori di x . Per comodità e semplicità definiremo h come distanza

in termini di numero di punti (e quindi come multiplo della distanza 0.025 gradi precedentemente usata per generare i punti del transetto). Costruiamo 4 grafici diversi, uno per ciascun valore di h , e calcoliamo la correlazione (ρ) tra $z(x)$ e $z(x+h)$ in ciascuno dei casi:

```
library(gridExtra)
nx = 100
h = 1

x_sample = sample(1: (nrow(xy)-30), nx, rep = F)
x_sample_plus_h = x_sample + h
z_x = xy$Depth[x_sample]
z_x_plus_h = xy$Depth[x_sample_plus_h]
z = data.frame(z_x = z_x, z_x_plus_h = z_x_plus_h)

g1 = ggplot(data = z,
             aes(x = z_x, y = z_x_plus_h)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +
  theme_minimal() +
  ggtitle(paste("h = 1 -  $\rho =$ ", round(cor(z_x, z_x_plus_h), 2)))

h = 5

x_sample = sample(1: (nrow(xy)-30), nx, rep = F)
x_sample_plus_h = x_sample + h
z_x = xy$Depth[x_sample]
z_x_plus_h = xy$Depth[x_sample_plus_h]
z = data.frame(z_x = z_x, z_x_plus_h = z_x_plus_h)

g2 = ggplot(data = z,
             aes(x = z_x, y = z_x_plus_h)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +
  theme_minimal() +
  ggtitle(paste("h = 10 -  $\rho =$ ", round(cor(z_x, z_x_plus_h), 2)))
```

```

h = 10

x_sample = sample(1: (nrow(xy)-30), nx, rep = F)
x_sample_plus_h = x_sample + h
z_x = xy$Depth[x_sample]
z_x_plus_h = xy$Depth[x_sample_plus_h]
z = data.frame(z_x = z_x, z_x_plus_h = z_x_plus_h)

g3 = ggplot(data = z,
            aes(x = z_x, y = z_x_plus_h)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +
  theme_minimal() +
  ggtitle(paste("h = 25 - p = ", round(cor(z_x, z_x_plus_h), 2)))

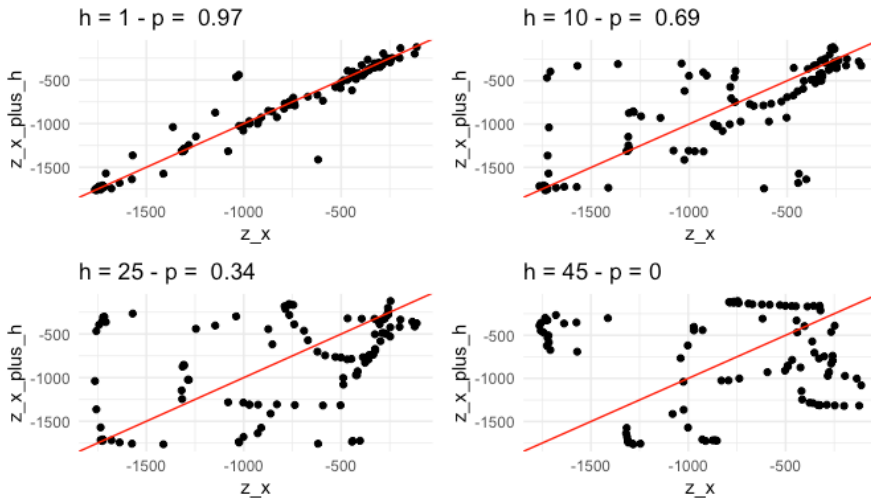
h = 25

x_sample = sample(1: (nrow(xy)-30), nx, rep = F)
x_sample_plus_h = x_sample + h
z_x = xy$Depth[x_sample]
z_x_plus_h = xy$Depth[x_sample_plus_h]
z = data.frame(z_x = z_x, z_x_plus_h = z_x_plus_h)

g4 = ggplot(data = z,
            aes(x = z_x, y = z_x_plus_h)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +
  theme_minimal() +
  ggtitle(paste0("h = 45 - p = ", round(cor(z_x, z_x_plus_h), 2)))

grid.arrange(g1, g2, g3, g4, nrow = 2)

```

Figura 185 - Correlazione tra p ed h .

Cosa dimostra questo esercizio? Come si vede dai grafici e dai valori di p , quando il valore di h è piccolo esiste una forte correlazione tra $z(x)$ e $z(x+h)$. Questa correlazione però decresce rapidamente (vale 0.67 quando $h = 10$) e crolla per valori di h più grandi. Questo avviene perchè quando h è piccolo saranno verosimilmente piccole anche le differenze tra le coppie di valori, e la nuvola sarà poco dispersa attorno alla retta a 45° . Per una distanza tra le coppie pari a zero, le differenze sono tutte nulle ed i punti della nuvola si collocano esattamente sulla retta a 45° .

Adesso ripetiamo questo esercizio tante volte per ottenere un'analisi completa dell'andamento di p in funzione di h :

```

nx = 120
hs = seq(1, 20, by = 1)
ps = numeric(length(hs))

for(i in 1:length(hs)){
  x_sample = sample(1: (nrow(xy)-20), nx, rep = F)
  x_sample_plus_h = x_sample + hs[i]
  z_x = xy$Depth[x_sample]
  z_x_plus_h = xy$Depth[x_sample_plus_h]

```

```

ps[i] = cor(z_x, z_x_plus_h)
}

hp = data.frame(h = hs, p = ps)
ggplot(data = hp, aes(x = h, y = p)) +
  geom_line() +
  theme_minimal()

```

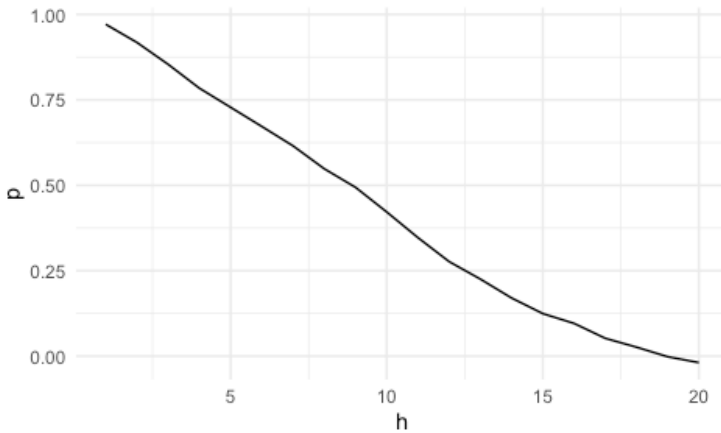


Figura 186 - L'andamento di p rispetto ad h .

L'andamento di p rispetto ad h mostra inequivocabilmente che esiste una relazione inversa tra i due valori. Questo semplice risultato è coerente con la **Teoria delle Variabili Regionalizzate** di Matheron (1965), secondo la quale i valori delle *VR* tendono ad essere correlati (luoghi più vicini sono più simili ad altri maggiormente distanti) a certe scale. La teoria di Matheron quantifica questa correlazione. Adesso ripetiamo lo stesso esercizio ma, invece di calcolare il valore di p rispetto ad h , calcoleremo, per ciascun valore di h , la varianza delle differenze (in valore assoluto) tra $z(x)$ e $z(x+h)$:

```

nx = 1000
hs = seq(1, 25, by = 1)
vs = numeric(length(hs))

```

```

for(i in 1:length(hs)){
  x_sample = sample(1: (nrow(xy)-30), nx, rep = T)
  x_sample_plus_h = x_sample + hs[i]
  z_x = xy$Depth[x_sample]
  z_x_plus_h = xy$Depth[x_sample_plus_h]
  vs[i] = var(abs(z_x - z_x_plus_h))
}

hv = data.frame(h = hs, v = vs)
ggplot(data = hv, aes(x = h, y = v)) +
  geom_line() +
  theme_minimal()

```

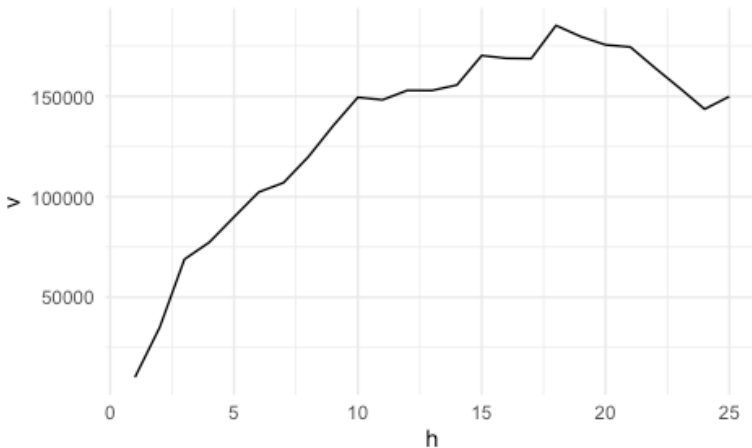


Figura 187 - Un variogramma.

Abbiamo ottenuto un andamento opposto al precedente: Quando $h = 0$ la varianza è zero e generalmente aumenta all'aumentare di h , assumendo un andamento asintotico. Il trasferimento di questi concetti di carattere empirico nel quadro di un formalismo teorico esigente costituisce l'approccio probabilistico della Geostatistica allo studio dei fenomeni naturali a carattere spaziale. Il grafico appena ottenuto si chiama **variogramma**. Se invece di calcolare la varianza usassimo la **semivarianza**, cioè la metà della varianza, otterremmo il **semivariogramma**.

```
hsv = data.frame(h = hs, sv = vs/2)
ggplot(data = hsv, aes(x = h, y = sv)) +
  geom_line() +
  theme_minimal()
```

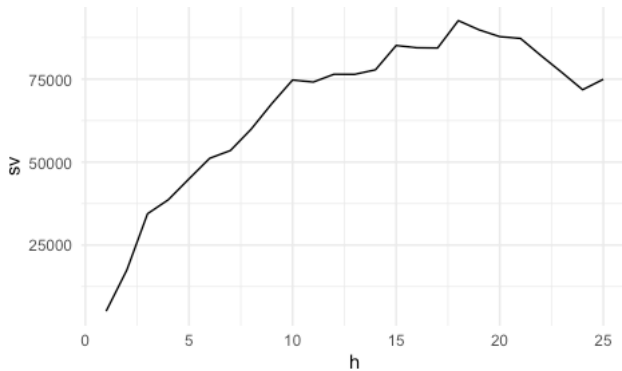


Figura 188 - Un semivariogramma.

La parte più importante del variogramma (o del semivariogramma) è la sua forma nei pressi dell'origine poiché ai punti più vicini verrà dato un peso maggiore durante l'interpolazione (che vedremo tra un attimo).



Figura 189 - Il variogramma ci mostra come una variabile spaziale cambia man mano che ci spostiamo da un punto qualsiasi.

Metodi di interpolazione

Supponiamo adesso di voler affrontare una piccola sfida: ricostruire il profilo della profondità (in sostanza la stessa *VR* rappresentata nel raster `depth16_sea`) a partire dalla profondità del fondale registrata dai nostri pescherecci durante la loro attività. In pratica, partendo dai valori (noti) della profondità in alcune aree di pesca prese a caso, vorremmo conoscere la profondità del fondale in località in cui non abbiamo mai pescato. Per dare una risposta a questo quesito dobbiamo, sostanzialmente, **interpolare** i valori della quantità di nostro interesse (la profondità) nello spazio. Citando Wikipedia: «In matematica, e in particolare in analisi numerica, per interpolazione si intende un metodo per individuare nuovi punti del piano cartesiano a partire da un insieme finito di punti dati, nell'ipotesi che tutti i punti si possano riferire ad una funzione $f(x)$ di una data famiglia di funzioni di una variabile reale». Il segreto di qualsiasi tecnica di interpolazione sta proprio nello sfruttare la **continuità spaziale**, che può essere quantificata mediante opportuni strumenti come i variogrammi e i covariogrammi che abbiamo visto prima.

Ma come possiamo sfruttare la **continuità spaziale**? Concettualmente possiamo facilmente intuire che, se vogliamo conoscere il valore di una certa *VR* in un certo punto, e se sappiamo che la *VR* che stiamo studiando è caratterizzata da un certo grado di **continuità spaziale**, allora dobbiamo trovare il modo di “sfruttare” i valori noti della *VR* vicino ai punti che sono per noi sconosciuti. Per farlo possiamo però usare due diverse famiglie di approcci:

- i metodi **deterministici**, in cui le nostre previsioni dei valori della *VR* (ad esempio delle catture di scampo) in siti sconosciuti sono completamente (e senza incertezza) determinati dai valori noti opportunamente combinati;
- i metodi **statistici**, in cui le nostre previsioni dei valori della *VR* (ad esempio delle catture di scampo) in siti sconosciuti saranno stimate a partire dai valori noti e, insieme alle stime, avremo anche un intervallo di confidenza.



Figura 190 - L'RMSE è un indicatore di quanto siamo andati (con le nostre stime) lontani dal bersaglio (i valori veri).

Di seguito vedremo esempi di entrambe le tipologie di metodi. Prima di farlo, però, introduciamo un ultimo strumento che ci sarà utile per valutare le capacità dei diversi metodi. Si tratta dell'Errore quadratico medio (**root-mean-square error – RMSE**), un indicatore che misura lo “scarto” tra due quantità vettoriali. Useremo questo indicatore per confrontare, su un insieme *test* di dati noti ma che non useremo per stimare i vari modelli (come facevamo con le reti neurali), le stime ottenute con i valori “veri”. L'RMSE è definito qui di seguito ed è ovvio che, tanto più esso è basso, tanto migliori sono le stime fornite dai modelli.

```
RMSE <- function(observed, predicted) {
  sqrt(mean((predicted - observed) ^ 2, na.rm=TRUE))
}
```

Poi, prendiamo il data frame *catch* (che conteneva anche il dato relativo alla profondità) e usiamolo in un esperimento ideale in cui i nostri pescherecci sono dei rilevatori della profondità del fondale marino. Dal data frame degli univoci di *Lon*, *Lat* e *Depth* possiamo generare due sottoinsiemi casuali del data frame *catch*:

- un sottoinsieme di 100 punti che denomineremo **training** e che useremo per stimare i modelli di interpolazione;
- un sottoinsieme di 100 punti che denomineremo **test** e che useremo per testare le capacità predittive delle nostre interpolazioni.

```

set.seed(1234)
df_depth = unique(catch[, c("LON", "LAT", "DEPTH")])
df_depth$LON = round(df_depth$LON, 1)
df_depth$LAT = round(df_depth$LAT, 1)
colnames(df_depth) = c("long", "lat", "depth")
df_depth = aggregate(data = df_depth,
                      depth ~.,
                      FUN = "median")

df_depth = data.matrix(df_depth)

id_training = sample(1:nrow(df_depth), 100, re = F)
id_test = sample(1:nrow(df_depth), 100, re = F)
#id_test = setdiff(1:nrow(df_depth), id_training)

training = df_depth[id_training,]
test = df_depth[id_test,]

```

Guardiamo la distribuzione dei due data frame ottenuti:

```

par(mfrow = c(1,2))
plotRGB(bg)
plot(depth16_sea, col = topo.colors(10), add = TRUE)
points(training, cex = 0.5, col = "firebrick", pch = 19)
title(main = "Training")

plotRGB(bg)
plot(depth16_sea, col = topo.colors(10), add = TRUE)
points(test, cex = 0.5, col = "firebrick", pch = 19)
title(main = "Test")

```

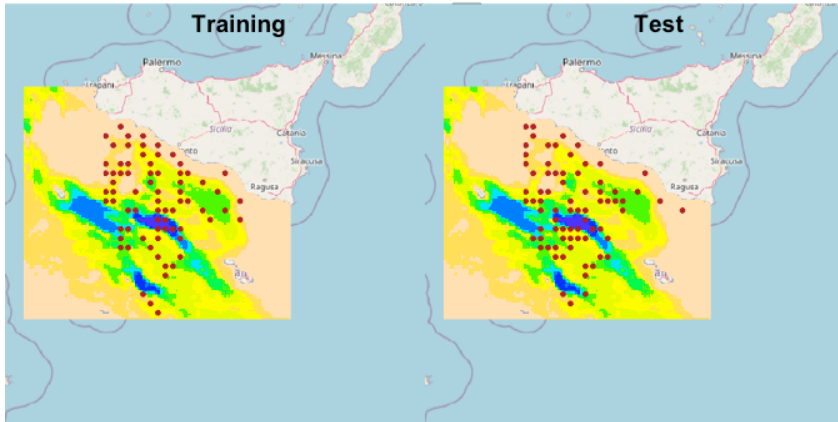


Figura 191 - Rappresentazione di punti di training e test.

Metodi deterministici

In questo gruppo di metodi, i valori sconosciuti della *VR* sono ottenuti mediante **funzioni lineari** i cui coefficienti vengono opportunamente calcolati e, quindi, non si tiene esplicitamente conto dell'autocorrelazione spaziale (misurata mediante un variogramma). Quindi, ogni valore predetto è il risultato di una funzione deterministica: le variabili possono assumere in un determinato istante uno e un solo valore.

Tra i modelli deterministici maggiormente usati ci sono:

- i **Poligoni di influenza** ottenuti mediante la ***Nearest Neighbour Analysis – NNR***;
- la **Interpolazione tramite Media Mobile**;
- l'**Inverso delle distanze** o ***Inverse Distance Weighted – IDW***.

La *Nearest Neighbour Analysis*, letteralmente l'“Analisi del vicino più prossimo”, è un approccio basato sull'assunto che il valore della nostra *VR* in un punto sconosciuto sia coincidente con quello della nostra osservazione più vicina. In sostanza, una estremizzazione del concetto «dimmi con chi vai e ti dirò chi sei». Naturalmente si tratta di una assunzione molto forte, ma non per questo sempre irragionevole o inutile. Nella figura seguente questo concetto è rappresentato nella situazione in cui vorremmo conoscere il valore della *VR* in corrispondenza del punto x_0 . Sebbene esistano 5 osservazioni intorno a

x_0 , il nostro modello associerà a x_0 lo stesso valore del punto 1, che è il più vicino ad esso.

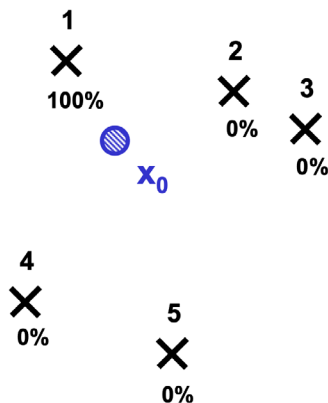


Figura 192 - Una rappresentazione del razionale alla base della NNR.

Per applicare questo metodo in R, dobbiamo fare conoscenza con la funzione `interpNear`. Questa funzione ci permette di applicare non solo la *Nearest Neighbour Analysis* ma, volendo, anche una interpolazione con una media mobile. L'opzione che ci permette di passare da *Nearest Neighbour Analysis* a media mobile è `interpolate`. Per adesso lasciamo questa opzione `FALSE` come da *default* e procediamo con la *Nearest Neighbour Analysis*.

Un ultimo argomento necessario è il raggio (`radius`). Infatti attraverso questo valore possiamo controllare l'estensione del dominio: punti lontani più del valore impostato con `radius` non saranno stimati e quindi troveremo dei `NA`. Applichiamo questa funzione per stimare, a partire dai punti del `training`, il valore delle catture di scampo nei punti del `test`. Prima di tutto generiamo un layer raster a partire dai dati del `training`, impostando un valore del raggio pari a 0.1 gradi:

```
# NNR: Nearest Neighbour Analysis
depth_star <- interpNear(depth16_sea,
                          df_depth[, c("long", "lat", "depth")],
                          radius = 0.1)

plotRGB(bg)
plot(depth_star, col = rev(heat.colors(10)), add = TRUE)
```

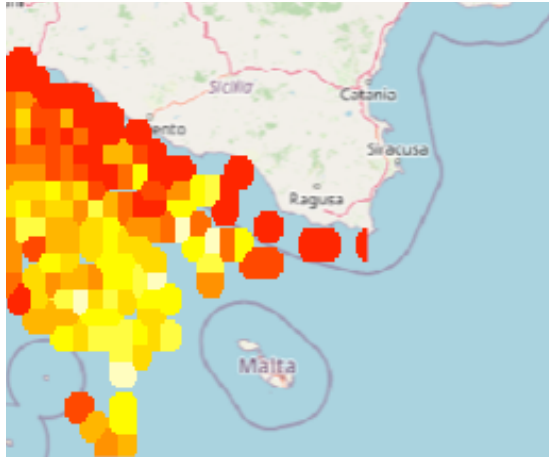


Figura 193 - Risultato della Nearest Neighbour Analysis.

Come possiamo vedere, intorno ai punti del *training* si è generato un cerchio. All'interno di ciascun cerchio tutta la superficie assume valori identici al punto che ha generato il cerchio stesso. Se due cerchi si sovrappongono, vince il cerchio del punto più vicino.

Con un valore del raggio pari a 0.1 otteniamo una copertura parziale dell'area. Proviamo a incrementare il raggio a 0.3:

```
# NNR: Nearest Neighbour Analysis
depth_star <- interpNear(depth16_sea,
                          training[, c("long", "lat", "depth")],
                          radius = 0.3)

plotRGB(bg)
plot(depth_star, col = rev(heat.colors(10)), add = TRUE)
```

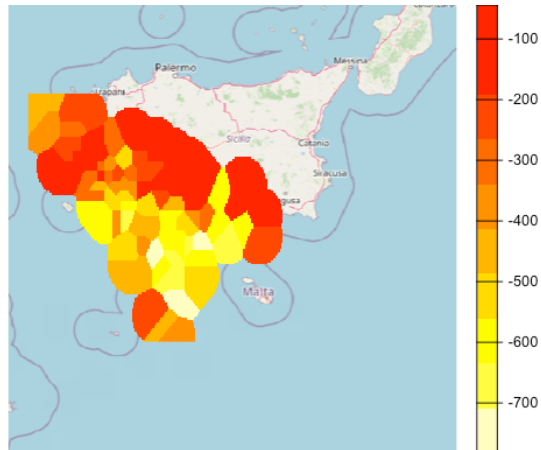


Figura 194 - Un altro risultato della Nearest Neighbour Analysis con un raggio maggiore.

Con un valore del raggio pari a 0.3 otteniamo una copertura quasi totale. Ora possiamo usare questo modello per stimare i valori delle catture di scampo nei punti del data frame test. Per farlo, dobbiamo nuovamente usare la funzione `extract`:

```
test_star = extract(depth_star, test[, c("long", "lat")])
df = data.frame(Observed = test[,3], Predicted = test_star[,1])
if(length(which(is.na(df)))>0) df = df[-which(is.na(df), arr.ind =
T)[,1],]

ggplot(data = df,
       aes(x = Observed, y = Predicted)) +
  geom_point(alpha = 0.5, pch = 21, fill = "darkgrey", cex = 2) +
  theme_test() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +
  annotate("text",
         x = 0.8 * min(df$Observed, na.rm = T),
         y = 0.2 * min(df$Predicted, na.rm = T),
         label = paste0("Adj R2 = ",
                        round(summary(lm(df$Predicted ~
df$Observed))$r.squared,3)))
```

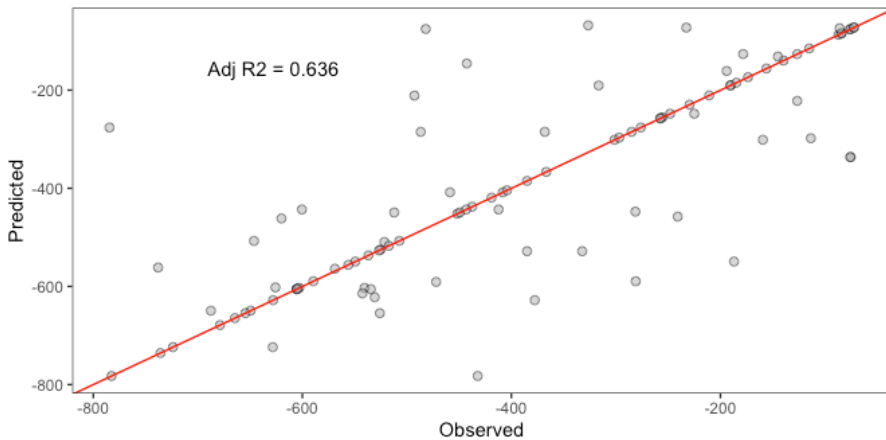


Figura 195 - Scatterplot dei valori predetti e osservati per il test dataset (metodo NNR).

```
R2_NNR = round(summary(lm(df$Predicted ~ df$Observed))$r.squared, 3)
```

Che dire? Una prestazione abbastanza buona nel predire i valori del *test*. Si vede chiaramente una relazione tra valori predetti e attesi, anche se alcuni punti del *test* sono lontani dalla bisettrice.

L' *RMSE* corrispondente a questa previsione è:

```
RMSE_NNR <- RMSE(df$Observed, df$Predicted)
RMSE_NNR
## [1] 129.8257
```

Un approccio simile, ma forse più chiaro come risultato grafico è la **tessellazione di Voronoi**. La **tessellazione di Voronoi** o **tessellazione di Dirichlet** è un tipo di decomposizione di uno spazio metrico (ad esempio un piano cartesiano come la rappresentazione Lon/Lat della nostra area di studio) determinata dalle distanze rispetto ad un determinato insieme discreto di elementi dello spazio (in questo caso, i punti del *training*).

In un caso di studio come il nostro, il *diagramma di Voronoi* è la partizione del piano che associa una regione $V(p)$ ad ogni punto p in modo tale che tutti i punti all'interno del perimetro di $V(p)$ siano più vicini a p che ad ogni altro punto all'interno del dominio spaziale definito. Per ottenerla, useremo la funzione `voronoi` di `terra`:


```
library(terra)
p <- vect(data.frame(training), geom = c("long", "lat"))
v <- voronoi(p)
depth_star <- interpNear(depth16_sea,
                          training[, c("long", "lat", "depth")],
                          radius = 0.2)

plotRGB(bg)
plot(depth_star, col = rev(heat.colors(10)), add = TRUE)
lines(v)
```

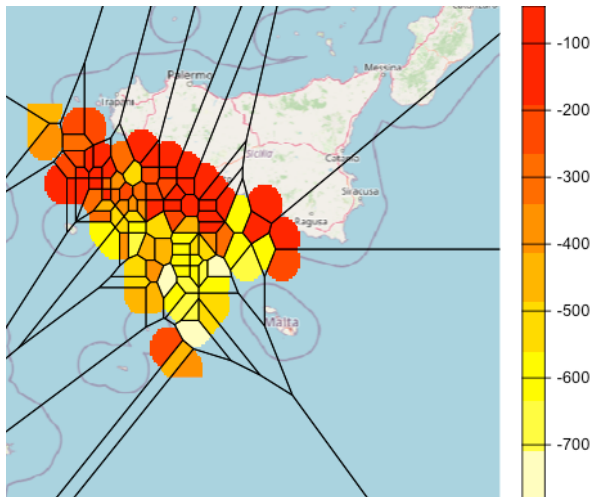


Figura 196 - Il risultato della tessellazione di Voronoi.

Ecco una rappresentazione dei valori delle catture predetti ed osservati:

```
test_star = extract(depth_star, test[, c("long", "lat")])
df = data.frame(Observed = test[,3], Predicted = test_star[,1])
if(length(which(is.na(df)))>0) df = df[-which(is.na(df), arr.ind =
T)[,1],]

ggplot(data = df,
       aes(x = Observed, y = Predicted)) +
  geom_point(alpha = 0.5, pch = 21, fill = "darkgrey", cex = 2) +
  theme_test() +
```

```
geom_abline(slope = 1, intercept = 0, colour = "red") +
annotate("text",
  x = 0.8 * min(df$Observed, na.rm = T),
  y = 0.2 * min(df$Predicted, na.rm = T),
  label = paste0("Adj R2 = ",
    round(summary(lm(df$Predicted ~
df$Observed))$r.squared,4)))
```

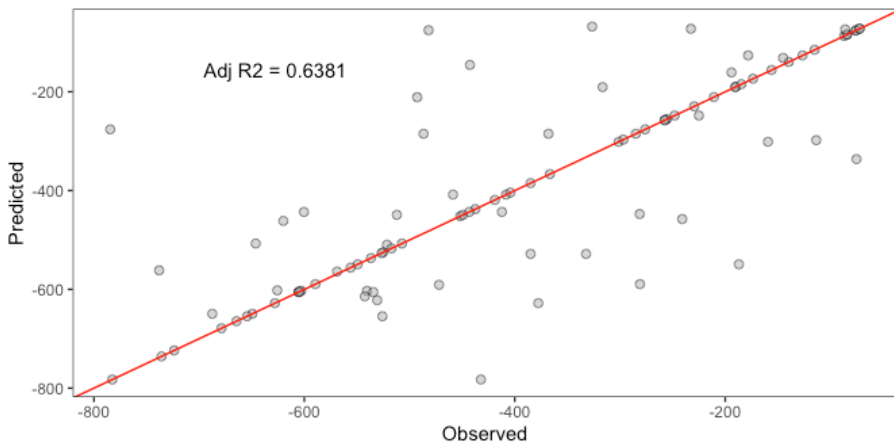


Figura 197 - Scatterplot dei valori predetti e osservati per il test dataset (metodo Voronoi).

```
R2_VOR = round(summary(lm(df$Predicted ~ df$Observed))$r.squared,3)
RMSE_VOR <- RMSE(df$Observed, df$Predicted)
RMSE_VOR
## [1] 128.393
```

Il nuovo modello non sembra discostarsi di molto da precedente, ma l'*RMSE* è appena più basso.

Come possiamo fare meglio? Tanto per cominciare, potremmo "rilassare" un metodo così rigido e brutale e usare una interpolazione lineare (di fatto una *media mobile*, o *moving average*) per calcolare il valore predetto a partire dai punti intorno, pesati in maniera identica. Quindi, per ogni punto x_0 , il valore predetto sarà la media negli n punti in un suo vicinato di raggio fissato.

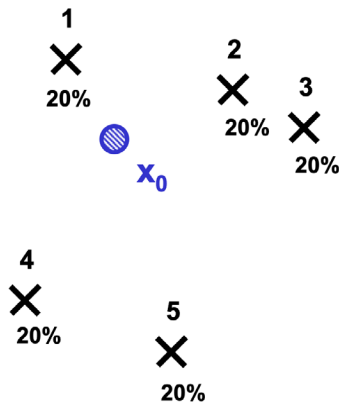


Figura 198 - Una rappresentazione del razionale alla base della interpolazione tramite Media Mobile.

Per fare questo, usiamo l'opzione `interpolate` della funzione `interpNear`:

```
# MA: Moving average
depth_star <- interpNear(depth16_sea,
                          training[, c("long", "lat", "depth")],
                          interpolate = TRUE,
                          radius = 0.3)

plotRGB(bg)
plot(depth_star, col = rev(heat.colors(10)), add = TRUE)
```

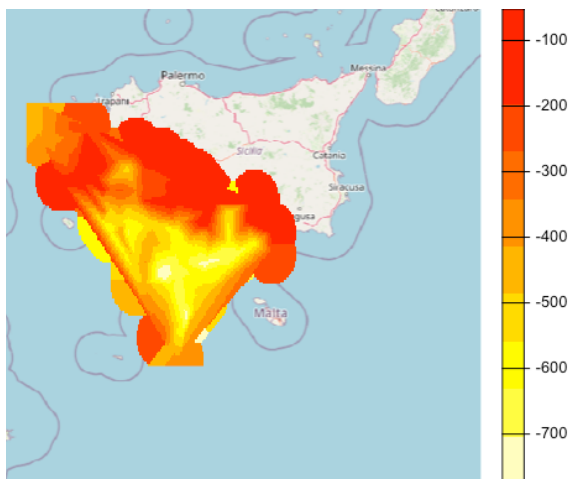


Figura 199 - Il risultato della Moving average.

Adesso riproviamo a ottenere le previsioni per il test:

```
test_star = extract(depth_star, test[, c("long", "lat")])
df = data.frame(Observed = test[,3], Predicted = test_star[,1])
if(length(which(is.na(df)))>0) df = df[-which(is.na(df), arr.ind = T)[,1],]

ggplot(data = df,
       aes(x = Observed, y = Predicted)) +
  geom_point(alpha = 0.5, pch = 21, fill = "darkgrey", cex = 2) +
  theme_test() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +
  annotate("text",
         x = 0.8 * min(df$Observed, na.rm = T),
         y = 0.2 * min(df$Predicted, na.rm = T),
         label = paste0("Adj R2 = ",
                        round(summary(lm(df$Predicted ~
df$Observed))$r.squared,4)))
```

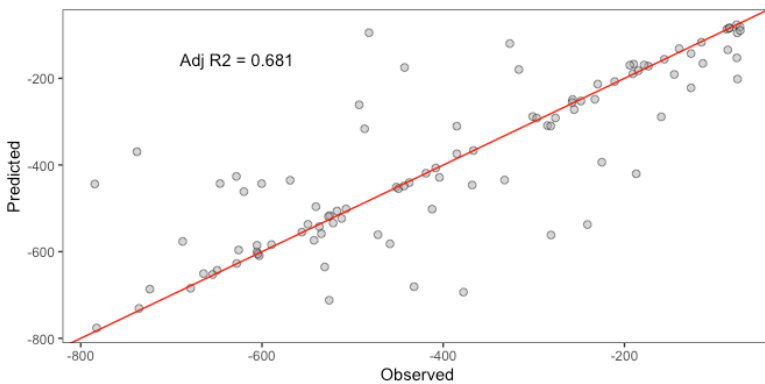


Figura 200 - Scatterplot dei valori predetti e osservati per il test dataset (metodo della Moving average).

```
R2_MA = round(summary(lm(df$Predicted ~ df$Observed))$r.squared,3)
```

Questa volta il valore di R^2 è sensibilmente aumentato. Vediamo l'*RMSE*:

```
RMSE_MA <- RMSE(df$Observed, df$Predicted)
RMSE_MA
## [1] 117.1135
```

Ehi, anche lui è sceso un po'!

A questo punto proviamo un altro approccio, cioè l'**Inverso delle distanze** o **Inverse Distance Weighted – IDW**. In questo caso, il valore della nostra variabile in un dato punto dello spazio sarà definito dalla media pesata (come inverso della distanza) delle osservazioni in un raggio r dal punto stesso.

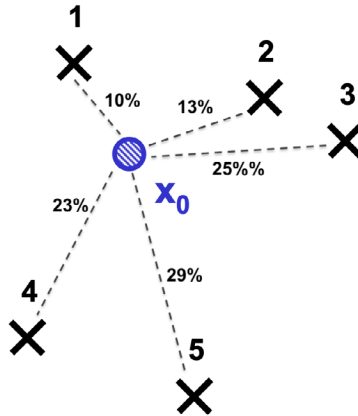


Figura 201 - Una rappresentazione del razionale alla base della IDW.

In effetti la distanza di ciascun punto nell'intorno può essere usata per calcolare il peso (cioè il contributo del punto alla media pesata) usando anche relazioni quadratiche o cubiche. Questo argomento può essere specificato tramite l'opzione `power` della funzione `interpIDW`, cioè la nuova funzione da usare. Vediamo cosa succede impostando `power = 2` (un peso calcolato come $1/\text{il quadrato della distanza}$), un raggio di 0.2 e un numero massimo di vicini (`maxPoints`) pari a 10:

```
# IDW: Inverse Distance Weighted
depth_star <- interpIDW(depth16_sea,
                        training[, c("long", "lat", "depth")],
```

```

        power = 2,
        maxPoints = 10,
        radius = 0.2)

plotRGB(bg)
plot(depth_star, col = rev(heat.colors(10)), add = TRUE)

```

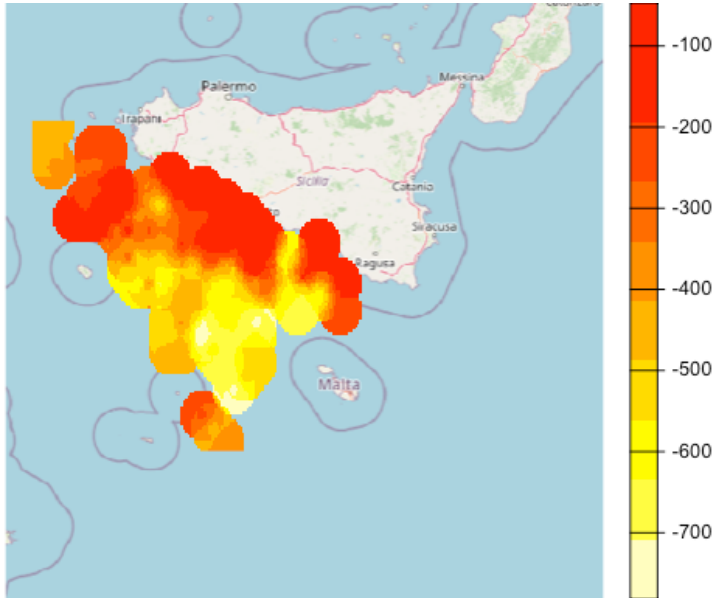


Figura 202 - Il risultato della Inverse Distance Weighted.

Dopo aver visto la nuova mappa, proviamo a fare una predizione sul test:

```

test_star = extract(depth_star, test[, c("long", "lat")])
df = data.frame(Observed = test[,3], Predicted = test_star[,1])
if(length(which(is.na(df)))>0) df = df[-which(is.na(df)), arr.ind =
T][,1,]

ggplot(data = df,
       aes(x = Observed, y = Predicted)) +
  geom_point(alpha = 0.5, pch = 21, fill = "darkgrey", cex = 2) +
  theme_test() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +

```

```

annotate("text",
  x = 0.8 * min(df$Observed, na.rm = T),
  y = 0.2 * min(df$Predicted, na.rm = T),
  label = paste0("Adj R2 = ",
    round(summary(lm(df$Predicted ~
df$Observed))$r.squared,4)))

```

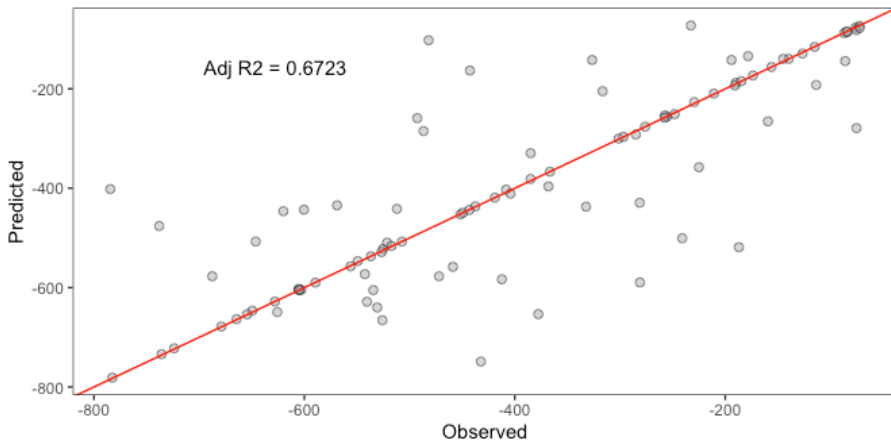


Figura 203 - Scatterplot dei valori predetti e osservati per il test dataset (metodo IDW).

```

R2_IDW = round(summary(lm(df$Predicted ~ df$Observed))$r.squared,3)
RMSE_IDW <- RMSE(df$Observed, df$Predicted)
RMSE_IDW
## [1] 119.9334

```

Ok, siamo ancora lì: abbiamo ottenuto risultati in linea con i precedenti. È venuto dunque il momento di fare un salto di qualità: ci serve un modello statistico.

Metodi statistici: Il *Kriging* ordinario e quello universale



Figura 204 - Il *Kriging* è il nostro personale raddomante che cammina per il deserto e ci indica dove potrebbe trovarsi l'acqua che cerchiamo.

Con il termine “*Kriging*” ci si riferisce a una famiglia di procedure geo-statistiche che assumono la presenza di una correlazione spaziale tra i valori misurati. Questa correlazione è stimata attraverso la costruzione dei variogrammi (che abbiamo incontrato precedentemente). Il nome *Kriging* viene da D. B. Krige, un ingegnere minerario sudafricano che ha definito il metodo (1951) insieme al suo collega H. S. Sichel. Krige e Sichel erano interessati ad ottimizzare le indagini minerarie, e quindi volevano fornire uno strumento che guidasse gli scavi verso località in cui le risorse attese fossero massime (a partire dalla conoscenza di località in cui già si estraevano minerali) come strumento di indagine dei giacimenti minerari. 10 anni dopo l'introduzione del *Kriging*, e prendendo spunto dal lavoro di Krige e Sichel, G. Matheron formalizzò la Teoria delle VR.

Il *Kriging* è, dunque, un metodo di interpolazione statistico che assume la condizione di **stazionarietà**. Per comprendere appieno come funziona, consideriamo la figura seguente (generata facendo uno zoom della nostra area di studio):

```
set.seed(12344)
dxy = data.frame(x = sample(seq(13.4, 13.8, 0.02), 4), y
```



```

=sample(seq(36.6, 36.9, 0.02), 4))
dxym = data.frame(x = mean(dxy$x), y = mean(dxy$y))
plot(depth16_sea, col = topo.colors(100),
      xlim = c(13, 14), ylim = c(36.5, 37))
points(dxy, pch = 19, col = "red", cex = 1)
points(dxym, pch = 19, col = "black", cex = 2)
points(dxym, pch = 21, col = "black", cex = 25)

```

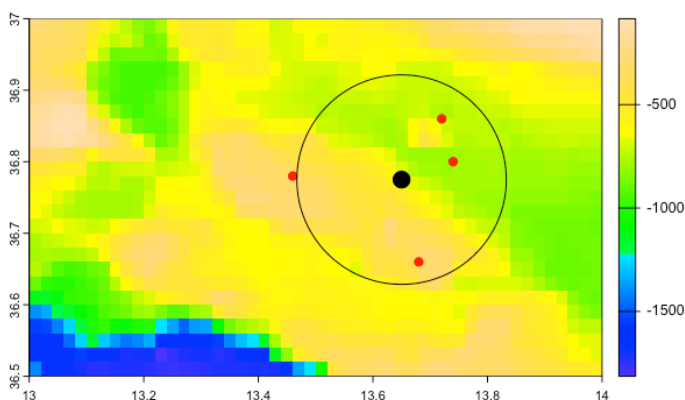


Figura 205 - Zoom della nostra area di studio.

Immaginiamo di voler conoscere il valore (ignoto) della profondità del fondale marino in corrispondenza del punto nero, partendo da rilevazioni note per i punti rossi. Il *Kriging* assume che, in un'area piccola attorno a ciascun punto della nostra rilevazione (delimitata dal cerchio nero che ha come centro il punto nero), la media della variabile (la profondità, nel nostro esempio) sia costante. I valori in ogni punto rosso cambieranno, perché la variabile avrà una certa varianza locale. Comunque, la media generale, assumendo che si possa campionare ovunque, rimarrà costante e uguale alla media aritmetica dei punti rossi. Quindi, assumendo che la nostra variabile si possa approssimare ad una distribuzione normale, se conosciamo la media possiamo calcolare il valore sconosciuto alla posizione n (punto nero) aggiungendo o sottraendo la varianza. Ma come? La varianza locale, la differenza tra il punto nero ed i vicini rossi, dipenderà dalla distanza tra loro. Il *Kriging* modella questa

differenza creando il variogramma, che consiste in una funzione che ci dice quant'è la varianza del pallino nero in relazione alla sua distanza dai vicini rossi. La semivarianza ottenuta dal variogramma è la metà della somma delle varianze (quadrato della deviazione standard) tra ogni valore z ed ognuno dei punti alla stessa distanza, e ci fornisce una misura della interdipendenza dei valori di profondità basata su quanto vicini essi sono (una misura del grado di dipendenza spaziale tra i campioni). Abbiamo già incontrato e calcolato un variogramma. Adesso dobbiamo vedere meglio cosa esso rappresenta, e per farlo prendiamo come esempio il grafico in figura.

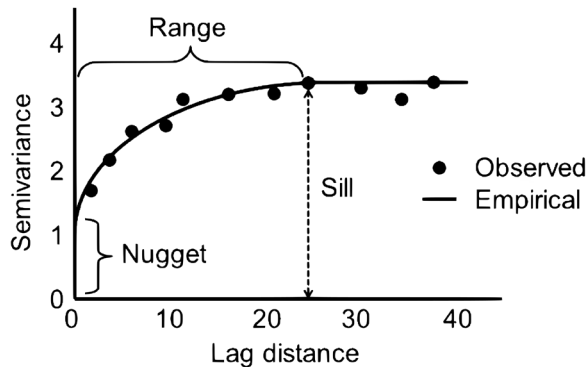


Figura 206 - Un esempio schematico di variogramma, dove i punti rappresentano i dati osservati e la curva la funzione del modello empirico utilizzato. Fonte: https://en.wikipedia.org/wiki/Variogram#/media/File:Schematic_variogram.svg.

Il variogramma stima il grado medio varianza tra valori in funzione della loro distanza nello spazio geografico. Il **nugget** rappresenta le variazioni casuali, della variabilità su piccola scala (inferiore cioè alla scala del campionamento), la variazione che ci sarebbe tra due punti più vicini rispetto alla distanza minima dei punti campionati. Il **range** corrisponde alla distanza in cui il semivariogramma (in un modello stazionario) raggiunge un valore costante detto **sill**. Intuitivamente, è la massima distanza per cui due punti possono essere considerati spazialmente correlati. Il **sill** rappresenta la varianza massima tra i punti misurati, che per modelli stazionari corrisponde alla varianza di campionamento.

Naturalmente la parte più importante del variogramma è la sua forma nei pressi dell'origine, poiché ai punti più vicini verrà dato un peso maggiore durante l'interpolazione. Dunque, a partire dalla forma del semivariogramma e dall'assunto che la variabilità locale intorno ad ogni punto segue una distribuzione normale con media stazionaria (*anisotropia*), non dobbiamo fare altro che stimare la forma del variogramma (la sua "legge") e i suoi parametri per ottenere il nostro modello. In particolare abbiamo:

- il **Nugget**: la (semi)varianza a scala spaziale infinitesimale (variabilità di corto raggio). Rappresenta una stima del residuo, rumore spazialmente non correlato. Combina le variazioni residue degli errori di misurazione con le variazioni spaziali che si verificano su distanze più brevi del passo (*lag*) di campionamento;
- il **sill**: la (semi)varianza che ci si aspetta su una scala spaziale infinita (molto grande);
- il **Range**: essenzialmente il dominio spaziale intorno ad ogni punto e all'interno del quale il modello sarà applicato.

R fornisce, attraverso la libreria **gstat**, che introduciamo ora, un'ampia varietà di funzioni per la geostatistica. In effetti, questa libreria è un riferimento per tutta la *community R* su questo argomento. Questa libreria ci fornisce, ad esempio, la funzione **variogram**. Il suo utilizzo è simile a quello di qualsiasi altro modello in R, come **lm**. Dobbiamo, cioè, indicare il data frame da usare come *input* e la forma del nostro modello. Questo può essere specificato mediante una formula che, generalmente, se abbiamo a che fare con un caso semplice come il nostro (profondità in funzione della località) assume una forma del tipo:

`dept ~ 1`

La funzione ci fornisce direttamente la distribuzione stimata della semivarianza rispetto alla distanza, e ci consente di esaminarla anche da un punto di vista visivo.

```
library(gstat)
df_training = as.data.frame(training)
colnames(df_training) = c("x", "y", "depth")
```

```
v <- variogram(depth ~ 1, ~ x + y, data = df_training)
```

```
v
```

##	np	dist	gamma	dir.hor	dir.ver	id
## 1	141	0.1196825	6987.411	0	0	var1
## 2	174	0.2157379	15714.982	0	0	var1
## 3	218	0.3040213	19917.504	0	0	var1
## 4	155	0.3735338	28417.941	0	0	var1
## 5	254	0.4285029	26281.629	0	0	var1
## 6	332	0.5191871	34172.035	0	0	var1
## 7	336	0.6137992	40992.875	0	0	var1
## 8	334	0.7061594	46349.325	0	0	var1
## 9	284	0.7898696	45745.000	0	0	var1
## 10	271	0.8613576	45034.681	0	0	var1
## 11	231	0.9279069	49875.005	0	0	var1
## 12	319	1.0169257	50109.709	0	0	var1
## 13	268	1.1117855	56881.340	0	0	var1
## 14	239	1.1998455	47767.542			

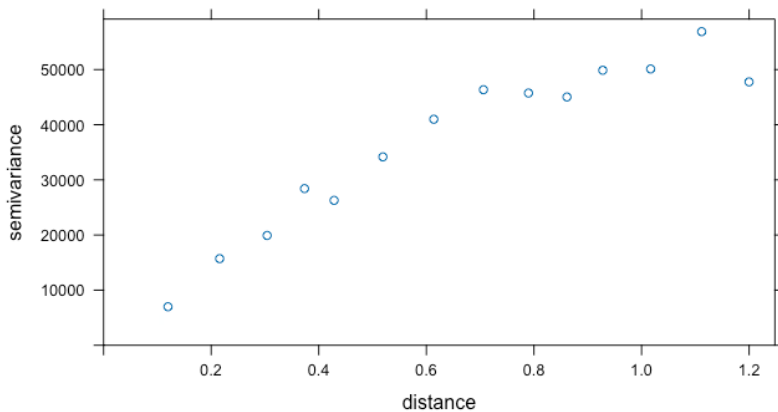


Figura 207 - Il semivariogramma ottenuto.

Generalmente il *Kriging* vuole che sia l'utente a scegliere i parametri e la forma del modello da usare per stimare il variogramma sperimentale. In figura troviamo degli esempi di forme per un variogramma.

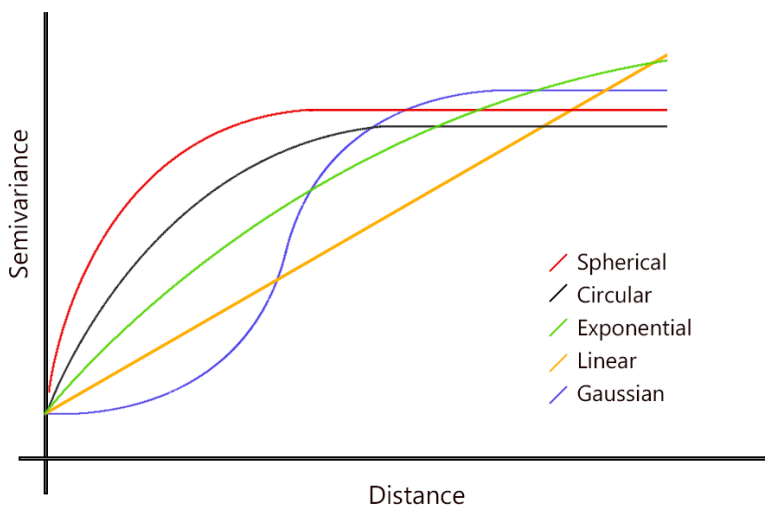


Figura 208 - Esempi di forme per il variogramma - Fonte: <https://www.aspexit.com/wp-content/uploads/2018/08/Variogram-models.png>.

Noi, però, non vogliamo addentrarci in questo genere di impresa, e proveremo ad automatizzare il tutto. Ad esempio, la funzione `fit.variogram` ci consente di ottenere un fit anche senza specificare una scelta della forma (notare l'uso della funzione `vgm` all'interno dell'esempio seguente) e i parametri di partenza.

```
fitted.variogram = fit.variogram(object = v,
                                model = vgm(c("Exp", "Ste", "Sph", "Mat")),
                                fit.kappa = TRUE,
                                fit.sills = TRUE,
                                fit.ranges = TRUE)

fitted.variogram
##   model      psill    range kappa
## 1  Nug     610.2241 0.0000000  0.0
## 2  Mat 58400.0046 0.3732491  0.9
```

Possiamo dare un'occhiata al variogramma fittato per verificare che abbia un bell'aspetto.

```
plot(v, fitted.variogram)
```

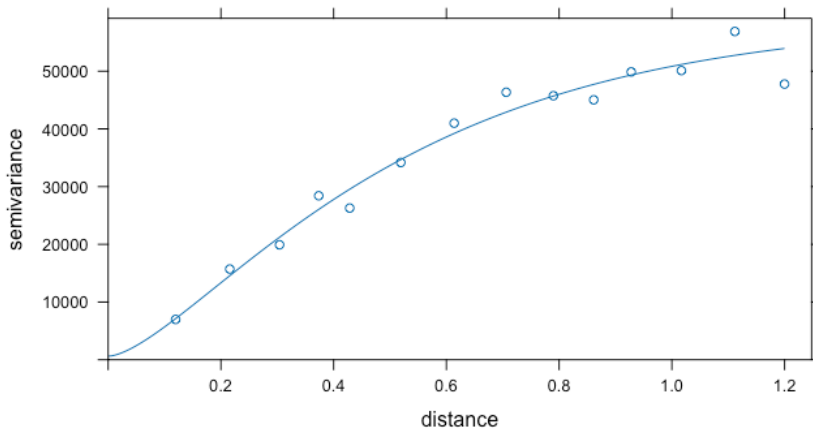


Figura 209 - Il semivariogramma fittato.

Ok, sembra tutto a posto. Non ci resta che procedere. Sempre usando **gstat**, fitteremo il nostro primo **Kriging ordinario** (o *Ordinary Kriging*), cioè la versione più semplice di *Kriging*.

Nel codice che segue faremo, nell'ordine, 4 cose:

- generiamo un raster vuoto all'interno del quale interpoleremo la profondità secondo il modello *Kriging* fittato;
- usiamo la funzione **gstat** per stimare il *Kriging*. La formula è la stessa usata per il variogramma, ma qui specifichiamo anche **depth**, che deve essere stimata dalle **locations**, e il variogramma fittato;
- mediante la funzione **interpolate** di **terra**, riempiamo il raster con le predizioni del nostro *Kriging*;
- generiamo due mappe diverse. La prima, **OK_pred**, contiene le predizioni del nostro *Ordinary Kriging*. La seconda, **OK_var**, contiene la varianza, cioè la distribuzione dell'errore (e quindi dell'incertezza delle nostre stime). Non ci siamo mica dimenticati che il *Kriging* è un metodo statistico (e non deterministico) e, quindi, che le stime che otteniamo sono sempre accompagnate da una distribuzione dell'errore.

```
grid_rast <- rast(nrows = 20 * 26,
                 ncols = 20 * 36,
                 nlyrs = 1,
                 xmin = st_bbox(depth16_sea)[1],
                 xmax = st_bbox(depth16_sea)[3],
```

```

ymin = st_bbox(depth16_sea)[2],
ymax = st_bbox(depth16_sea)[4],
crs = st_crs(depth16_sea))

# Ordinary Kriging
depth_OK <- gstat(NULL, "depth", depth ~ 1,
  data = df_training,
  locations = ~ x + y,
  model = fitted.variogram)

OK <- interpolate(object = grid_rast,
  model = depth_OK,
  debug.level = 0)

OK_pred <- OK[[c("depth.pred")]]
OK_var <- OK[[c("depth.var")]]

```

Adesso vediamo l'aspetto del nostro modello fittato sul raster:

```

plotRGB(bg)
plot(OK_pred, col = topo.colors(10), add = TRUE)

```

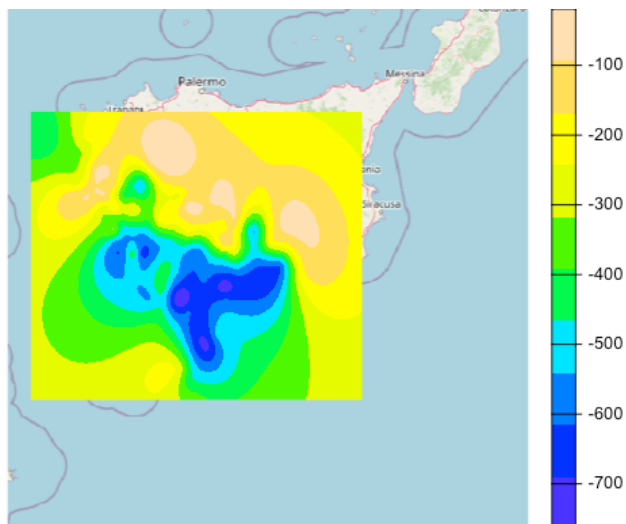


Figura 210 - Mappa delle stime locali del Kriging.

```
plotRGB(bg)
plot(OK_var, col = topo.colors(10), add = TRUE)
```

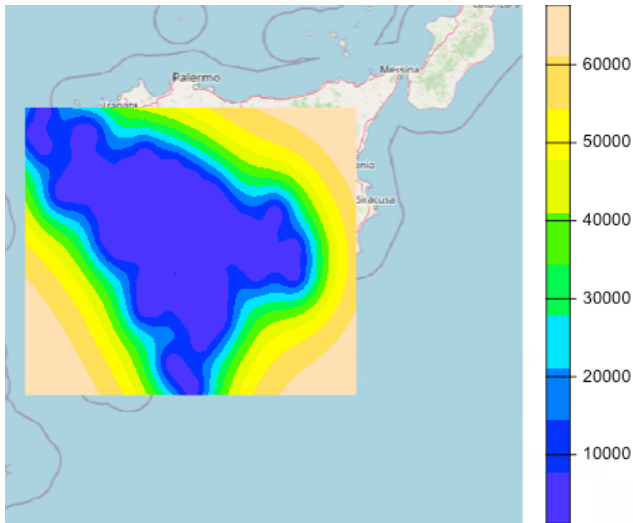


Figura 211 - Mappa della varianza del Kriging.

A quanto pare, l'incertezza cresce dal centro della nostra area di studio verso l'esterno, come era lecito attendersi visto che le nostre osservazioni sono concentrate in una sottoporzione del raster generato per il dominio. Adesso valutiamo il nostro modello rispetto al *test*...

```
pred_OK = extract(OK, test[, c("long", "lat")])$depth.pred
df = data.frame(Observed = test[,3], Predicted = pred_OK)

ggplot(data = df,
       aes(x = Observed, y = Predicted)) +
  geom_point(alpha = 0.5, pch = 21, fill = "darkgrey", cex = 2) +
  theme_test() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +
  annotate("text",
         x = 0.8 * min(df$Observed, na.rm = T),
         y = 0.2 * min(df$Predicted, na.rm = T),
         label = paste0("Adj R2 = ",
```



```
round(summary(lm(df$Predicted ~
df$Observed))$r.squared,4))
```

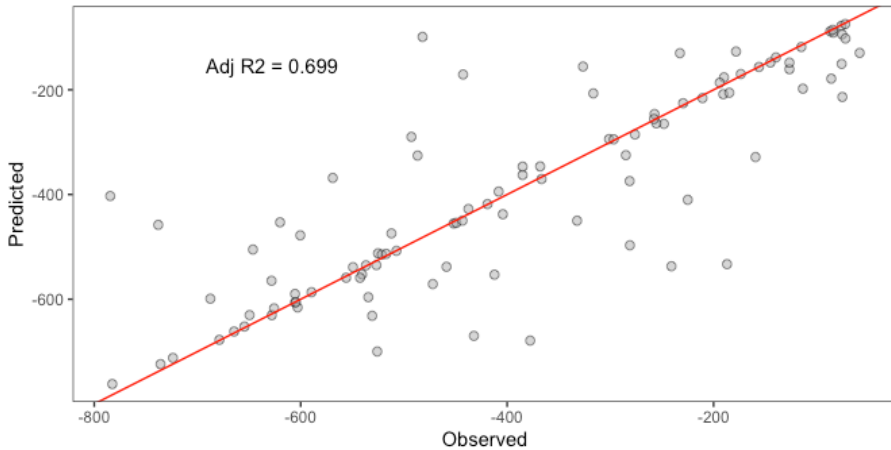


Figura 212 - Scatterplot dei valori predetti e osservati per il test dataset (metodo del Kriging ordinario).

```
R2_OK = round(summary(lm(df$Predicted ~ df$Observed))$r.squared,3)
```

L' R^2 è aumentato ancora. Passiamo all'*RMSE*...

```
RMSE_OK <- RMSE(df$Observed, df$Predicted)
RMSE_OK
## [1] 114.3419
```

Anche questo valore è migliorato (è diventato più piccolo).

L'ultimo metodo che introdurremo è una variante del precedente, ed è noto come **Universal Kriging (UK)**. Si differenzia dal precedente perché non assume che la media sia **stazionaria**. Anzi, lo *Universal Kriging* prevede che la media locale abbia una dipendenza funzionale dalla posizione spaziale e possa essere approssimata da un modello. Lo *Universal Kriging* è noto anche come "*Kriging con tendenza*" o "*Kriging in presenza di deriva*". La tendenza spaziale o la deriva rappresentano qualsiasi tendenza rilevabile dei valori a cambiare in funzione delle variabili coordinate.

Mettiamolo alla prova e diamo un'occhiata ai risultati.

```
# Universal Kriging
depth_UK <- gstat(NULL, "depth", depth ~ x + y,
  data = df_training,
  locations = ~ x + y,
  model = fitted.variogram)

UK <- interpolate(object = grid_rast,
  model = depth_UK,
  debug.level = 0)

UK_pred <- UK[[c("depth.pred")]]
UK_var <- UK[[c("depth.var")]]

plotRGB(bg)
plot(UK_pred, col = topo.colors(10), add = TRUE)
```

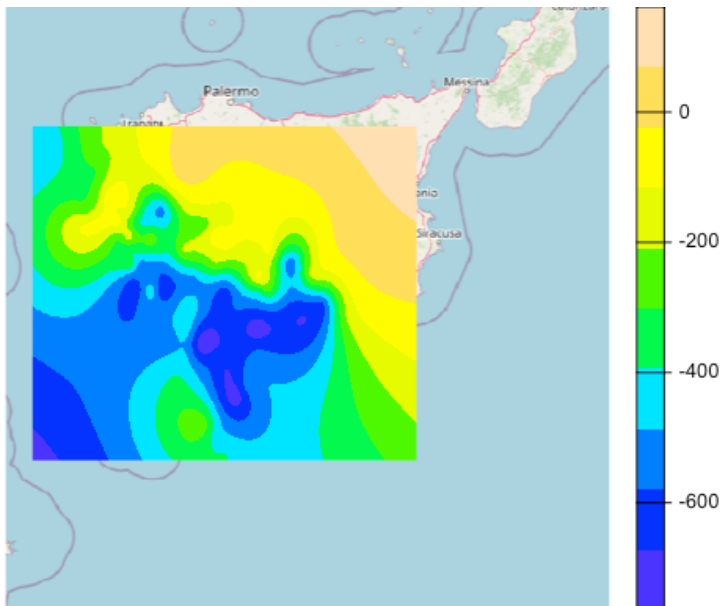


Figura 213 - La mappa della profondità ottenuta dall'applicazione del Kriging universale.

```
plotRGB(bg)
plot(UK_var, col = topo.colors(10), add = TRUE)
```

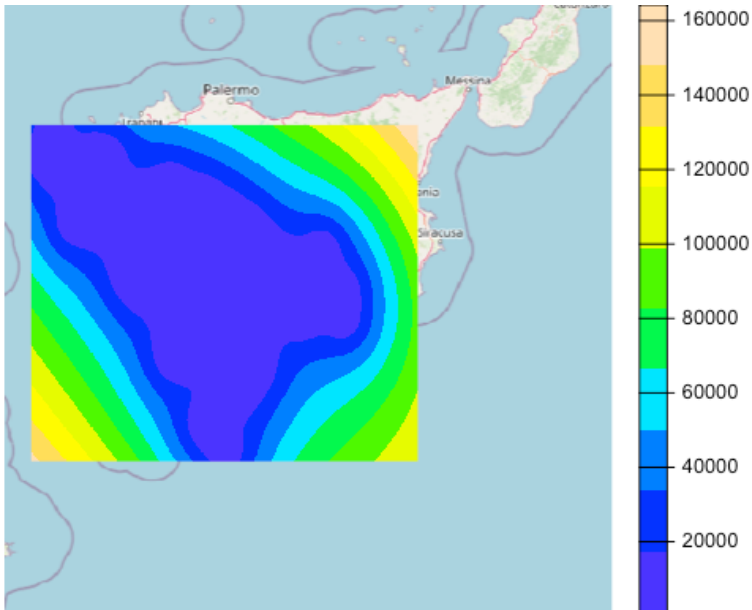


Figura 214 - L'incertezza associata alle predizioni fornite dal Kriging universale.

Ed infine, valutiamone le prestazioni.

```
pred_UK = extract(UK, test[, c("long", "lat")])$depth.pred
df = data.frame(Observed = test[,3], Predicted = pred_UK)

ggplot(data = df,
       aes(x = Observed, y = Predicted)) +
  geom_point(alpha = 0.5, pch = 21, fill = "darkgrey", cex = 2) +
  theme_test() +
  geom_abline(slope = 1, intercept = 0, colour = "red") +
  annotate("text",
         x = 0.8 * min(df$Observed, na.rm = T),
         y = 0.2 * min(df$Predicted, na.rm = T),
         label = paste0("Adj R2 = ",
                        round(summary(lm(df$Predicted ~
df$Observed))$r.squared,4)))
```

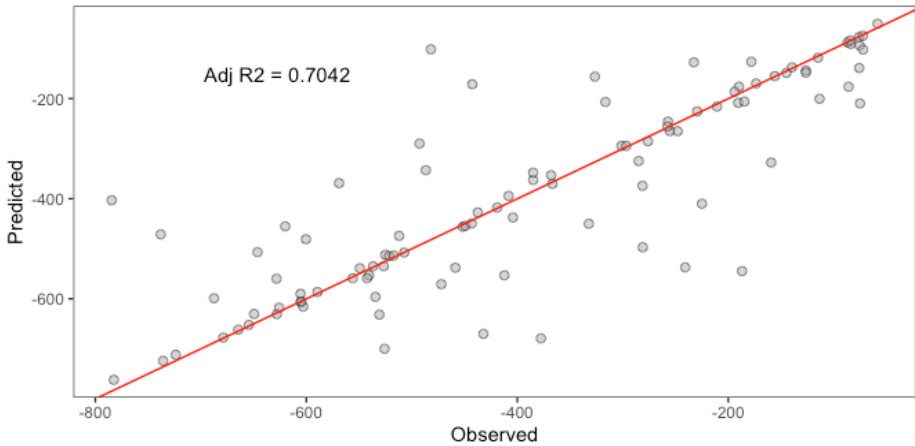


Figura 215 - Scatterplot dei valori predetti e osservati per il test dataset (metodo Kriging universale).

```
R2_UK = round(summary(lm(df$Predicted ~ df$Observed))$r.squared,3)
```

Non resta che l'*RMSE*:

```
RMSE_UK <- RMSE(df$Observed, df$Predicted)
RMSE_UK
## [1] 113.6076
```

Siamo arrivati ad un valore di R^2 leggermente superiore a 0.7, ed un valore dell'*RMSE* pari a 113.6076, il più piccolo tra quelli ottenuti.

Possiamo ricapitolare i risultati ottenuti con i vari modelli preparando un grafico che ci permetta di fare dei confronti sulla base dei valori di R^2 e *RMSE* ottenuti.

```
library(ggplot2)
dfc = data.frame(Method = c("Nearest Neighbour Analysis",
                           "Voronoi",
                           "Moving average",
                           "Inverse Distance Weighted",
                           "Ordinary Kriging",
                           "Universal Kriging"),
```

```

R2 = c(R2_NNR,
        R2_VOR,
        R2_MA,
        R2_IDW,
        R2_OK,
        R2_UK),
RMSE = c(RMSE_NNR,
          RMSE_VOR,
          RMSE_MA,
          RMSE_IDW,
          RMSE_OK,
          RMSE_UK))

ggplot(data = dfc,
       aes(x = R2, y = RMSE)) +
  geom_point() +
  theme_minimal() +
  xlab(expression(R2)) +
  geom_label_repel(aes(label = Method))

```

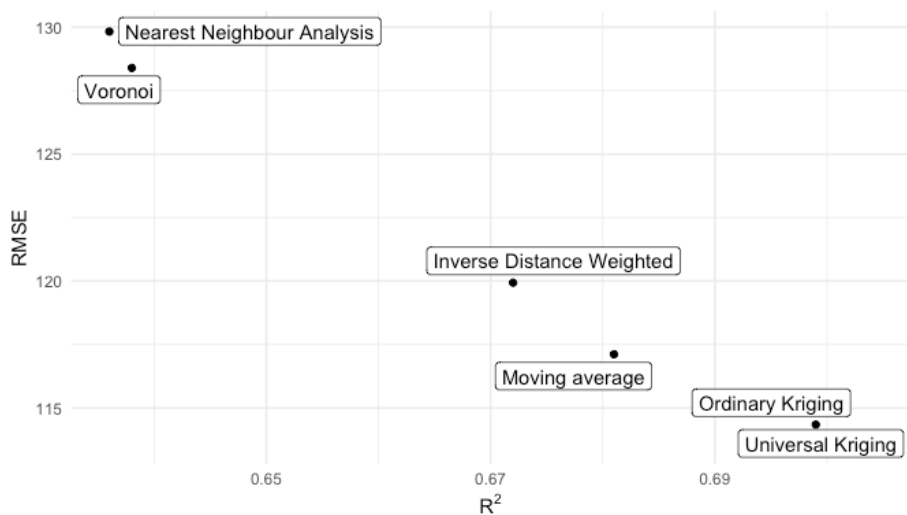


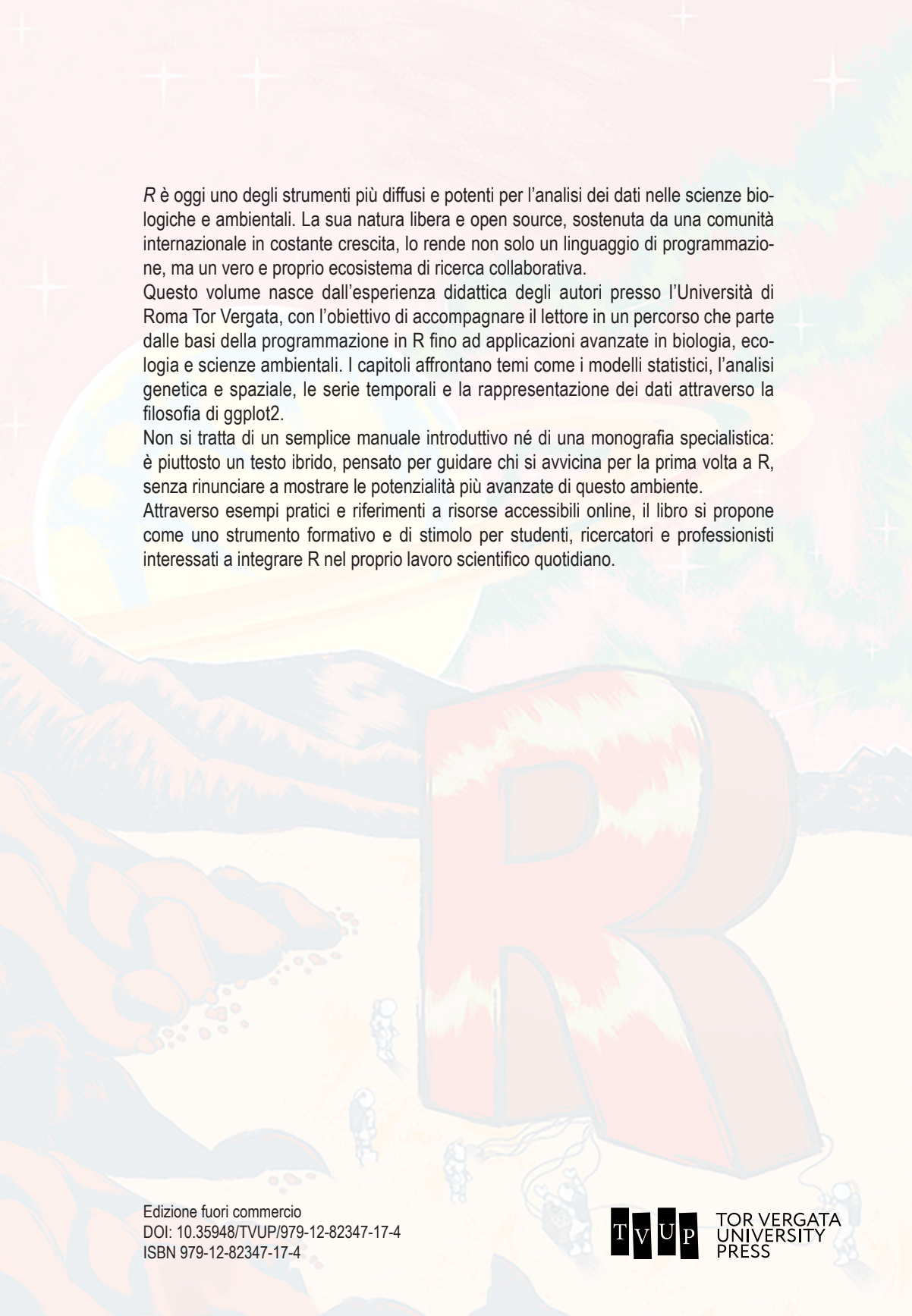
Figura 216 - Un confronto finale tra i metodi che abbiamo provato.

Come si vede dal grafico, abbiamo compiuto un ideale percorso che ci ha condotto ad avere valori sempre più alti di R^2 e, al tempo stesso, più bassi di $RMSE$. Possiamo ritenerci, tutto sommato, abbastanza soddisfatti!

Risultati migliori si possono comunque ottenere con altri approcci, che però esulano dagli argomenti trattati in questo capitolo.



Figura 217 - Usando alcuni indicatori si possono valutare i modelli e scegliere il migliore per una data applicazione.



R è oggi uno degli strumenti più diffusi e potenti per l'analisi dei dati nelle scienze biologiche e ambientali. La sua natura libera e open source, sostenuta da una comunità internazionale in costante crescita, lo rende non solo un linguaggio di programmazione, ma un vero e proprio ecosistema di ricerca collaborativa.

Questo volume nasce dall'esperienza didattica degli autori presso l'Università di Roma Tor Vergata, con l'obiettivo di accompagnare il lettore in un percorso che parte dalle basi della programmazione in *R* fino ad applicazioni avanzate in biologia, ecologia e scienze ambientali. I capitoli affrontano temi come i modelli statistici, l'analisi genetica e spaziale, le serie temporali e la rappresentazione dei dati attraverso la filosofia di *ggplot2*.

Non si tratta di un semplice manuale introduttivo né di una monografia specialistica: è piuttosto un testo ibrido, pensato per guidare chi si avvicina per la prima volta a *R*, senza rinunciare a mostrare le potenzialità più avanzate di questo ambiente.

Attraverso esempi pratici e riferimenti a risorse accessibili online, il libro si propone come uno strumento formativo e di stimolo per studenti, ricercatori e professionisti interessati a integrare *R* nel proprio lavoro scientifico quotidiano.