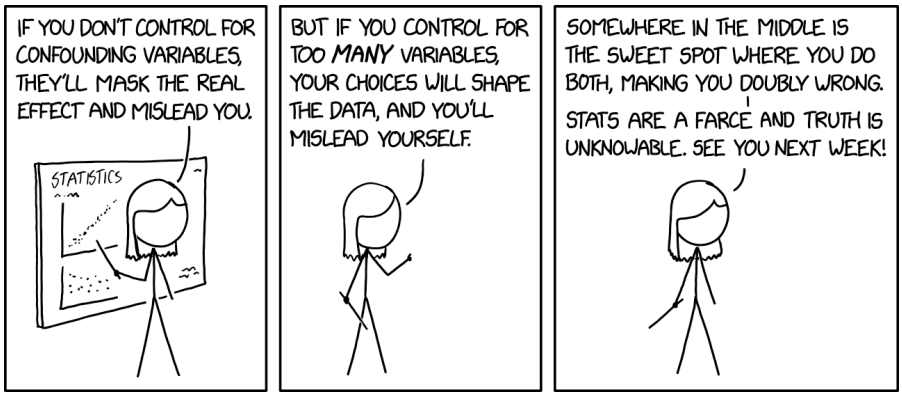
Corrado Caudek

Data Science per psicologi

Psicometria – AA 2021/2022



Indice

Elenco delle figure

Elenco delle tabelle

Prefazione

Data Science per psicologi contiene il materiale delle lezioni dell'insegnamento di *Psicometria B000286* (A.A. 2021/2022) rivolto agli studenti del primo anno del Corso di Laurea in Scienze e Tecniche Psicologiche dell'Università degli Studi di Firenze. *Psicometria* si propone di fornire agli studenti un'introduzione all'analisi dei dati in psicologia. Le conoscenze/competenze che verranno sviluppate in questo insegnamento sono quelle della Data science, ovvero un insieme di conoscenze/competenze che si pongono all'intersezione tra statistica (ovvero, richiedono la capacità di comprendere teoremi statistici) e informatica (ovvero, richiedono la capacità di sapere utilizzare un software).

La psicologia e la Data science

Sembra sensato spendere due parole su un tema che è importante per gli studenti: quello indicato dal titolo di questo Capitolo. È ovvio che agli studenti di psicologia la statistica non piace. Se piacesse, forse studierebbero Data science e non psicologia; ma non lo fanno. Di conseguenza, gli studenti di psicologia si chiedono: "perché dobbiamo perdere tanto tempo a studiare queste cose quando in realtà quello che ci interessa è tutt'altro?" Questa è una bella domanda.

C'è una ragione molto semplice che dovrebbe farci capire perché la Data science è così importante per la psicologia. Infatti, a ben pensarci, la psicologia è una disciplina intrinsecamente statistica, se per statistica intendiamo quella disciplina che studia la variazione delle caratteristiche degli individui nella popolazione. La psicologia studia *gli individui* ed è proprio la variabilità inter- e intra-individuale ciò che vogliamo descrivere e, in certi casi, predire. In questo senso, la psicologia è molto diversa dall'ingegneria, per esempio. Le proprietà di un determinato ponte sotto certe condizioni, ad esempio, sono molto simili a quelle di un altro pon-

te, sotto le medesime condizioni. Quindi, per un ingegnere la statistica è poco importante: le proprietà dei materiali sono unicamente dipendenti dalla loro composizione e restano costanti. Ma lo stesso non può dirsi degli individui: ogni individuo è unico e cambia nel tempo. E le variazioni tra gli individui, e di un individuo nel tempo, sono l'oggetto di studio proprio della psicologia: è dunque chiaro che i problemi che la psicologia si pone sono molto diversi da quelli affrontati, per esempio, dagli ingegneri. Questa è la ragione per cui abbiamo tanto bisogno della Data science in psicologia: perché la Data science ci consente di descrivere la variazione e il cambiamento. E queste sono appunto le caratteristiche di base dei fenomeni psicologici.

Sono sicuro che, leggendo queste righe, a molti studenti sarà venuta in mente la seguente domanda: perché non chiediamo a qualche esperto di fare il "lavoro sporco" (ovvero le analisi statistiche) per noi, mentre noi (gli psicologi) ci occupiamo solo di ciò che ci interessa, ovvero dei problemi psicologici slegati dai dettagli "tecnici" della Data science? La risposta a questa domanda è che non è possibile progettare uno studio psicologico sensato senza avere almeno una comprensione rudimentale della Data science. Le tematiche della Data science non possono essere ignorate né dai ricercatori in psicologia né da coloro che svolgono la professione di psicologo al di fuori dell'Università. Infatti, anche i professionisti al di fuori dall'università non possono fare a meno di leggere la letteratura psicologica più recente: il continuo aggiornamento delle conoscenze è infatti richiesto dalla deontologia della professione. Ma per potere fare questo è necessario conoscere un bel po' di Data science! Basta aprire a caso una rivista specialistica di psicologia per rendersi conto di quanto ciò sia vero: gli articoli che riportano i risultati delle ricerche psicologiche sono zeppi di analisi statistiche e di modelli formali. E la comprensione della letteratura psicologica rappresenta un requisito minimo nel bagaglio professionale dello psicologo.

Le considerazioni precedenti cercano di chiarire il seguente punto: la Data science non è qualcosa da studiare a malincuore, in un singolo insegnamento universitario, per poi poterla tranquillamente dimenticare. Nel bene e nel male, gli psicologi usano gli strumenti della Data science in tantissimi ambiti della loro attività professionale: in particolare quando costruiscono, somministrano e interpretano i test psicometrici. È dunque chiaro che possedere delle solide basi di Data science è un tassello imprescindibile del bagaglio professionale dello psicologo. In questo insegnamento verranno trattati i temi base della Data science e verrà

adottato un punto di vista bayesiano, che corrisponde all'approccio più recente e sempre più diffuso in psicologia.

Come studiare

Il giusto metodo di studio per prepararsi all'esame di Psicometria è quello di seguire attivamente le lezioni, assimilare i concetti via via che essi vengono presentati e verificare in autonomia le procedure presentate a lezione. Incoraggio gli studenti a farmi domande per chiarire ciò che non è stato capito appieno. Incoraggio gli studenti a utilizzare i forum attivi su Moodle e, soprattutto, a svolgere gli esercizi proposti su Moodle. I problemi forniti su Moodle rappresentano il livello di difficoltà richiesto per superare l'esame e consentono allo studente di comprendere se le competenze sviluppate fino a quel punto sono sufficienti rispetto alle richieste dell'esame.

La prima fase dello studio, che è sicuramente individuale, è quella in cui è necessario acquisire le conoscenze teoriche relative ai problemi che saranno presentati all'esame. La seconda fase di studio, che può essere facilitata da scambi con altri e da incontri di gruppo, porta ad acquisire la capacità di applicare le conoscenze: è necessario capire come usare un software (R) per applicare i concetti statistici alla specifica situazione del problema che si vuole risolvere. Le due fasi non sono però separate: il saper fare molto spesso ci aiuta a capire meglio.

Sviluppare un metodo di studio efficace

Avendo insegnato molte volte in passato un corso introduttivo di analisi dei dati ho notato nel corso degli anni che gli studenti con l'atteggiamento mentale che descriverò qui sotto generalmente ottengono ottimi risultati. Alcuni studenti sviluppano naturalmente questo approccio allo studio, ma altri hanno bisogno di fare uno sforzo per maturarlo. Fornisco qui sotto una breve descrizione del "metodo di studio" che, nella mia esperienza, è il più efficace per affrontare le richieste di questo insegnamento.

- Dedicate un tempo sufficiente al materiale di base, apparentemente facile; assicuratevi di averlo capito bene. Cercate le lacune nella vostra comprensione. Leggere presentazioni diverse dello stesso materiale (in libri o articoli diversi) può fornire nuove intuizioni.
- Gli errori che facciamo sono i nostri migliori maestri. Istintivamente cerchiamo di dimenticare subito i nostri errori. Ma il miglior modo di imparare è apprendere dagli errori che commettiamo. In questo senso, una soluzione corretta è meno utile di una soluzione sbagliata. Quando commettiamo un errore questo ci fornisce un'informazione importante: ci fa capire qual è il materiale di studio sul quale dobbiamo ritornare e che dobbiamo capire meglio.
- C'è ovviamente un aspetto "psicologico" nello studio. Quando un esercizio o problema ci sembra incomprensibile, la cosa migliore da fare è dire: "mi arrendo", "non ho idea di cosa fare!". Questo ci rilassa: ci siamo già arresi, quindi non abbiamo niente da perdere, non dobbiamo più preoccuparci. Ma non dobbiamo fermarci qui. Le cose "migliori" che faccio (se ci sono) le faccio quando non ho voglia di lavorare. Alle volte, quando c'è qualcosa che non so fare e non ho idea di come affontare, mi dico: "oggi non ho proprio voglia di fare fatica", non ho voglia di mettermi nello stato mentale per cui "in 10 minuti devo risolvere il problema perché dopo devo fare altre cose". Però ho voglia di *divertirmi* con quel problema e allora mi dedico a qualche aspetto "marginale" del problema, che so come affrontare, oppure considero l'aspetto più difficile del problema, quello che non so come risolvere, ma invece di cercare di risolverlo, guardo come altre persone hanno affrontato problemi simili, opppure lo stesso problema in un altro contesto. Non mi pongo l'obiettivo "risolvi il problema in 10 minuti", ma invece quello di farmi un'idea "generale" del problema, o quello di capire un caso più specifico e più semplice del problema. Senza nessuna pressione. Infatti, in quel momento ho deciso di non lavorare (ovvero, di non fare fatica). Va benissimo se "parto per la tangente", ovvero se mi metto a leggere del materiale che sembra avere poco a che fare con il problema centrale (le nostre intuizioni e la nostra curiosità solitamente ci indirizzano sulla strada giusta). Quando faccio così, molto spesso trovo la soluzione del problema che mi ero posto e, paradossalmente, la trovo in un tempo minore di quello che, in precedenza, avevo dedicato a "lavorare" al problema. Allora perché non faccio sempre così? C'è ovviamente l'aspetto dei "10 minuti" che non è sempre facile da dimenticare. Sotto pressione, possiamo solo agire in maniera automatica, ovvero possia-

mo solo applicare qualcosa che già sappiamo fare. Ma se dobbiamo imparare qualcosa di nuovo, la pressione è un impedimento.

- È utile farsi da soli delle domande sugli argomenti trattati, senza limitarsi a cercare di risolvere gli esercizi che vengono assegnati. Quando studio qualcosa mi viene in mente: "se questo è vero, allora deve succedere quest'altra cosa". Allora verifico se questo è vero, di solito con una simulazione. Se i risultati della simulazione sono quelli che mi aspetto, allora vuol dire che ho capito. Se i risultati sono diversi da quelli che mi aspettavo, allora mi rendo conto di non avere capito e ritorno indietro a studiare con più attenzione la teoria che pensavo di avere capito – e ovviamente mi rendo conto che c'era un aspetto che avevo frainteso. Questo tipo di verifica è qualcosa che dobbiamo fare da soli, in prima persona: nessun altro può fare questo al posto nostro.
- Non aspettatevi di capire tutto la prima volta che incontrate un argomento nuovo.[1] È utile farsi una nota mentalmente delle lacune nella vostra comprensione e tornare su di esse in seguito per carcare di colmarle. L'atteggiamento naturale, quando non capiamo i dettagli di qualcosa, è quello di pensare: "non importa, ho capito in maniera approssimativa questo punto, non devo preoccuparmi del resto". Ma in realtà non è vero: se la nostra comprensione è superficiale, quando il problema verrà presentato in una nuova forma, non riusciremo a risolverlo. Per cui i dubbi che ci vengono quando studiamo qualcosa sono il nostro alleato più prezioso: ci dicono esattamente quali sono gli aspetti che dobbiamo approfondire per potere migliorare la nostra preparazione.
- È utile sviluppare una visione d'insieme degli argomenti trattati, capire l'obiettivo generale che si vuole raggiungere e avere chiaro il contributo che i vari pezzi di informazione forniscono al raggiungimento di tale obiettivo. Questa organizzazione mentale del materiale di studio facilita la comprensione. È estremamente utile creare degli schemi di ciò che si sta studiando. Non aspettate che sia io a fornirvi un riepilogo di ciò che dovete imparare: sviluppate da soli tali schemi e tali riassunti.
- Tutti noi dobbiamo imparare l'arte di trovare le informazioni, non solo nel caso di questo insegnamento. Quando vi trovate di fronte a qualcosa che non capite, o ottenete un oscuro messaggio di errore da

[1] Ricordatevi inoltre che gli individui tendono a sottostimare la propria capacità di apprendere (Horn and Loewenstein, 2021).

un software, ricordatevi: “Google is your friend”!

Corrado Caudek
Marzo 2022

Parte I

Nozioni preliminari

1

Concetti chiave

La *data science* si pone all'intersezione tra statistica e informatica. La statistica è un insieme di metodi ugilizzati per estrarre informazioni dai dati; l'informatica implementa tali procedure in un software. In questo Capitolo vengono introdotti i concetti fondamentali.

1.1 Popolazioni e campioni

Popolazione. L'analisi dei dati inizia con l'individuazione delle unità portatrici di informazioni circa il fenomeno di interesse. Si dice popolazione (o universo) l'insieme Ω delle entità capaci di fornire informazioni sul fenomeno oggetto dell'indagine statistica. Possiamo scrivere $\Omega = \{\omega_i\}_{i=1,...,n} = \{\omega_1, \omega_2, ..., \omega_n\}$, oppure $\Omega = \{\omega_1, \omega_2, ...\}$ nel caso di popolazioni finite o infinite, rispettivamente.

L'obiettivo principale della ricerca psicologica è conoscere gli esiti psicologici e i loro fattori trainanti nella popolazione. Questo è l'obiettivo delle sperimentazioni psicologiche e della maggior parte degli studi osservazionali in psicologia. È quindi necessario essere molto chiari sulla popolazione a cui si applicano i risultati della ricerca. La popolazione può essere ben definita, ad esempio, tutte le persone che si trovavano nella città di Hiroshima al momento dei bombardamenti atomici e sono sopravvissute al primo anno, o può essere ipotetica, ad esempio, tutte le persone depresse che hanno subito o saranno sottoporsi ad un intervento di psicoterapia. Il ricercatore deve sempre essere in grado di determinare se un soggetto appartiene alla popolazione oggetto di interesse.

Una *sottopopolazione* è una popolazione in sé e per sé che soddisfa proprietà ben definite. Negli esempi precedenti, potremmo essere interessati alla sottopopolazione di uomini di età inferiore ai 20 anni o di pazienti depressi sottoposti ad uno specifico intervento psicologico. Molte questio-

ni scientifiche riguardano le differenze tra sottopopolazioni; ad esempio, confrontando i gruppi con o senza psicoterapia per determinare se il trattamento è vantaggioso. I modelli di regressione, introdotti nel Capitolo **??** riguardano le sottopopolazioni, in quanto stimano il risultato medio per diversi gruppi (sottopopolazioni) definiti dalle covariate.

Campione. Gli elementi ω_i dell'insieme Ω sono detti *unità statistiche*. Un sottoinsieme della popolazione, ovvero un insieme di elementi ω_i, viene chiamato *campione*. Ciascuna unità statistica ω_i (abbreviata con u.s.) è portatrice dell'informazione che verrà rilevata mediante un'operazione di misurazione.

Un campione è dunque un sottoinsieme della popolazione utilizzato per conoscere tale popolazione. A differenza di una sottopopolazione definita in base a chiari criteri, un campione viene generalmente selezionato tramite un procedura casuale. Il *campionamento casuale* consente allo scienziato di trarre conclusioni sulla popolazione e, soprattutto, di quantificare l'incertezza sui risultati. I campioni di un sondaggio sono esempi di campioni casuali, ma molti studi osservazionali non sono campionati casualmente. Possono essere *campioni di convenienza*, come coorti di studenti in un unico istituto, che consistono di tutti gli studenti sottoposti ad un certo intervento psicologico in quell'istituto. Indipendentemente da come vengono ottenuti i campioni, il loro uso al fine di conoscere una popolazione target significa che i problemi di rappresentatività sono inevitabili e devono essere affrontati.

1.2 Variabili e costanti

Definiamo *variabile statistica* la proprietà (o grandezza) che è oggetto di studio nell'analisi dei dati. Una variabile è una proprietà di un fenomeno che può essere espressa in più valori sia numerici sia categoriali. Il termine "variabile" si contrappone al termine "costante" che descrive una proprietà invariante di tutte le unità statistiche.

Si dice *modalità* ciascuna delle varianti con cui una variabile statistica può presentarsi. Definiamo *insieme delle modalità* di una variabile statistica l'insieme M di tutte le possibili espressioni con cui la variabile può manifestarsi. Le modalità osservate e facenti parte del campione si chiamano *dati* (si veda la Tabella 1.1).

Esempio 1.1. Supponiamo che il fenomeno studiato sia l'intelligenza. In uno studio, la popolazione potrebbe corrispondere all'insieme di tutti gli italiani adulti. La variabile considerata potrebbe essere il punteggio del test standardizzato WAIS-IV. Le modalità di tale variabile potrebbero essere $112, 92, 121, ...$ Tale variabile è di tipo quantitativo discreto.

Esempio 1.2. Supponiamo che il fenomeno studiato sia il compito Stroop. La popolazione potrebbe corrispondere all'insieme dei bambini dai 6 agli 8 anni. La variabile considerata potrebbe essere il reciproco dei tempi di reazione in secondi. Le modalità di tale variabile potrebbero essere $1/2.35, 1/1.49, 1/2.93, ...$ La variabile è di tipo quantitativo continuo.

Esempio 1.3. Supponiamo che il fenomeno studiato sia il disturbo di personalità. La popolazione potrebbe corrispondere all'insieme dei detenuti nelle carceri italiane. La variabile considerata potrebbe essere l'assessment del disturbo di personalità tramite interviste cliniche strutturate. Le modalità di tale variabile potrebbero essere i Cluster A, Cluster B, Cluster C descritti dal DSM-V. Tale variabile è di tipo qualitativo.

1.2.1 Variabili casuali

Il termine *variabile* usato nella statistica è equivalente al termine *variabile casuale* usato nella teoria delle probabilità. Lo studio dei risultati degli interventi psicologici è lo studio delle variabili casuali che misurano questi risultati. Una variabile casuale cattura una caratteristica specifica degli individui nella popolazione e i suoi valori variano tipicamente tra gli individui. Ogni variabile casuale può assumere in teoria una gamma di valori sebbene, in pratica, osserviamo un valore specifico per ogni individuo. Quando faremo riferiremo alle variabili casuali considerate in termini generali useremo lettere maiuscole come X e Y; quando faremo riferimento ai valori che una variabile casuale assume in determinate circostanze useremo lettere minuscole come x e y.

1.2.2 Variabili indipendenti e variabili dipendenti

Un primo compito fondamentale in qualsiasi analisi dei dati è l'identificazione delle variabili dipendenti (Y) e delle variabili indipendenti (X). Le variabili dipendenti sono anche chiamate variabili di esito o di risposta e le variabili indipendenti sono anche chiamate predittori o covariate. Ad esempio, nell'analisi di regressione, che esamineremo in seguito, la

domanda centrale è quella di capire come Y cambia al variare di X. Più precisamente, la domanda che viene posta è: se il valore della variabile indipendente X cambia, qual è la conseguenza per la variabile dipendente Y? In parole povere, le variabili indipendenti e dipendenti sono analoghe a "cause" ed "effetti", laddove le virgolette usate qui sottolineano che questa è solo un'analogia e che la determinazione delle cause può avvenire soltanto mediante l'utilizzo di un appropriato disegno sperimentale e di un'adeguata analisi statistica.

Se una variabile è una variabile indipendente o dipendente dipende dalla domanda di ricerca. A volte può essere difficile decidere quale variabile è dipendente e quale è indipendente, in particolare quando siamo specificamente interessati ai rapporti di causa/effetto. Ad esempio, supponiamo di indagare l'associazione tra esercizio fisico e insonnia. Vi sono evidenze che l'esercizio fisico (fatto al momento giusto della giornata) può ridurre l'insonnia. Ma l'insonnia può anche ridurre la capacità di una persona di fare esercizio fisico. In questo caso, dunque, non è facile capire quale sia la causa e quale l'effetto, quale sia la variabile dipendente e quale la variabile indipendente. La possibilità di identificare il ruolo delle variabili (dipendente/indipendente) dipende dalla nostra comprensione del fenomeno in esame.

Esempio 1.4. Uno psicologo convoca 120 studenti universitari per un test di memoria. Prima di iniziare l'esperimento, a metà dei soggetti viene detto che si tratta di un compito particolarmente difficile; agli altri soggetti non viene data alcuna indicazione. Lo psicologo misura il punteggio nella prova di memoria di ciascun soggetto.

In questo esperimento, la variabile indipendente è l'informazione sulla difficoltà della prova. La variabile indipendente viene manipolata dallo sperimentatore assegnando i soggetti (di solito in maniera causale) o alla condizione (modalità) "informazione assegnata" o "informazione non data". La variabile dipendente è ciò che viene misurato nell'esperimento, ovvero il punteggio nella prova di memoria di ciascun soggetto.

1.2.3 La matrice dei dati

Le realizzazioni delle variabili esaminate in una rilevazione statistica vengono organizzate in una *matrice dei dati*. Le colonne della matrice dei dati contengono gli insiemi dei dati individuali di ciascuna variabile statistica considerata. Ogni riga della matrice contiene tutte le informazioni

relative alla stessa unità statistica. Una generica matrice dei dati ha l'aspetto seguente:

$$D_{m,n} = \begin{pmatrix} \omega_1 & a_1 & b_1 & \cdots & x_1 & y_1 \\ \omega_2 & a_2 & b_2 & \cdots & x_2 & y_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \omega_n & a_n & b_n & \cdots & x_n & y_n \end{pmatrix}$$

dove, nel caso presente, la prima colonna contiene il nome delle unità statistiche, la seconda e la terza colonna si riferiscono a due mutabili statistiche (variabili categoriali; A e B) e ne presentano le modalità osservate nel campione mentre le ultime due colonne si riferiscono a due variabili statistiche (X e Y) e ne presentano le modalità osservate nel campione. Generalmente, tra le unità statistiche ω_i non esiste un ordine progressivo; l'indice attribuito alle unità statistiche nella matrice dei dati si riferisce semplicemente alla riga che esse occupano.

1.3 Parametri e modelli

Ogni variabile casuale ha una *distribuzione* che descrive la probabilità che la variabile assuma qualsiasi valore in un dato intervallo.[1] Senza ulteriori specificazioni, una distribuzione può fare riferimento a un'intera famiglia di distribuzioni. I parametri, tipicamente indicati con lettere greche come μ e α, ci permettono di specificare di quale membro della famiglia stiamo parlando. Quindi, si può parlare di una variabile casuale con una distribuzione Normale, ma se viene specificata la media $\mu = 100$ e la varianza $\sigma^2 = 15$, viene individuata una specifica distribuzione Normale – nell'esempio, la distribuzione del quoziente di intelligenza.

I metodi statistici parametrici specificano la famiglia delle distribuzioni e quindi utilizzano i dati per individuare, stimando i parametri, una specifica distribuzione all'interno della famiglia di distribuzioni ipotizzata. Se f è la PDF di una variabile casuale Y, l'interesse può concentrarsi

[1] In questo e nei successivi Paragrafi di questo Capitolo introduco gli obiettivi della *data science* utilizzando una serie di concetti che saranno chiariti solo in seguito. Questa breve panoramica risulterà dunque solo in parte comprensibile ad una prima lettura e serve solo per definire la *big picture* dei temi trattati in questo insegnamento. Il significato dei termini qui utilizzati sarà chiarito nei Capitoli successivi.

sulla sua media e varianza. Nell'analisi di regressione, ad esempio, cerchiamo di spiegare come i parametri di f dipendano dalle covariate X. Nella regressione lineare classica, assumiamo che Y abbia una distribuzione normale con media $\mu = \mathbb{E}(Y)$, e stimiamo come $\mathbb{E}(Y)$ dipenda da X. Poiché molti esiti psicologici non seguono una distribuzione normale, verranno introdotte distribuzioni più appropriate per questi risultati. I metodi non parametrici, invece, non specificano una famiglia di distribuzioni per f. In queste dispense faremo riferimento a metodi non parametrici quando discuteremo della statistica descrittiva.

Il termine *modello* è onnipresente in statistica e nella *data science*. Il modello statistico include le ipotesi e le specifiche matematiche relative alla distribuzione della variabile casuale di interesse. Il modello dipende dai dati e dalla domanda di ricerca, ma raramente è unico; nella maggior parte dei casi, esiste più di un modello che potrebbe ragionevolmente usato per affrontare la stessa domanda di ricerca e avendo a disposizione i dati osservati. Nella previsione delle aspettative future dei pazienti depressi che discuteremo in seguito (Zetsche et al., 2019), ad esempio, la specifica del modello include l'insieme delle covariate candidate, l'espressione matematica che collega i predittori con le aspattative future e qualsiasi ipotesi sulla distribuzione della variabile dipendente. La domanda di cosa costituisca un buon modello è una domanda su cui torneremo ripetutamente in questo insegnamento.

1.4 Effetto

L'*effetto* è una qualche misura dei dati. Dipende dal tipo di dati e dal tipo di test statistico che si vuole utilizzare. Ad esempio, se viene lanciata una moneta 100 volte e esce testa 66 volte, l'effetto sarà 66/100. Diventa poi possibile confrontare l'effetto ottenuto con l'effetto nullo che ci si aspetterebbe da una moneta bilanciata (50/100), o con qualsiasi altro effetto che può essere scelto. La *dimensione dell'effetto* si riferisce alla differenza tra l'effetto misurato nei dati e l'effetto nullo (di solito un valore che ci si aspetta di ottenere in base al caso soltanto).

1.5 Stima e inferenza

La stima è il processo mediante il quale il campione viene utilizzato per conoscere le proprietà di interesse della popolazione. La media campionaria è una stima naturale della media della popolazione e la mediana campionaria è una stima naturale della mediana della popolazione. Quando parliamo di stimare una proprietà della popolazione (a volte indicata come parametro della popolazione) o di stimare la distribuzione di una variabile casuale, stiamo parlando dell'utilizzo dei dati osservati per conoscere le proprietà di interesse della popolazione. L'inferenza statistica è il processo mediante il quale le stime campionarie vengono utilizzate per rispondere a domande di ricerca e per valutare specifiche ipotesi relative alla popolazione. Discuteremo le procedure bayesiane dell'inferenza nell'ultima parte di queste dispense.

1.6 Metodi e procedure della psicologia

Un modello psicologico di un qualche aspetto del comportamento umano o della mente ha le seguenti proprietà:

1. descrive le caratteristiche del comportamento in questione,
2. formula predizioni sulle caratteristiche future del comportamento,
3. è sostenuto da evidenze empiriche,
4. deve essere falsificabile (ovvero, in linea di principio, deve potere fare delle predizioni su aspetti del fenomeno considerato che non sono ancora noti e che, se venissero indagati, potrebbero portare a rigettare il modello, se si dimostrassero incompatibili con esso).

L'analisi dei dati valuta un modello psicologico utilizzando strumenti statistici.

Questa dispensa è strutturata in maniera tale da rispecchiare la suddivisione tra i temi della misurazione, dell'analisi descrittiva e dell'inferenza. Nel prossimo Capitolo sarà affrontato il tema della misurazione e, nell'ul-

tima parte della dispensa verrà discusso l'argomento più difficile, quello dell'inferenza. Prima di affrontare il secondo tema, l'analisi descrittiva dei dati, sarà necessario introdurre il linguaggio di programmazione statistica R (un'introduzione a R è fornita in Appendice). Inoltre, prima di potere discutere l'inferenza, dovranno essere introdotti i concetti di base della teoria delle probabilità, in quanto l'inferenza non è che l'applicazione della teoria delle probabilità all'analisi dei dati.

2

Modello Normale-Normale

2.1 Distribuzione Normale-Normale con varianza nota

Per σ^2 nota, la v.c. gaussiana è distribuzione a priori coniugata della v.c. gaussiana. Siano $Y_1, \dots, Y_n$ n variabili casuali i.i.d. che seguono la distribuzione gaussiana:

$$Y_1, \dots, Y_n \overset{iid}{\sim} \mathcal{N}(\mu, \sigma).$$

Si vuole stimare μ sulla base di n osservazioni $y_1, \dots, y_n$. Considereremo qui solamente il caso in cui σ^2 sia supposta perfettamente nota.

Ricordiamo che la densità di una gaussiana è

$$p(y_i \mid \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(y_i - \mu)^2}{2\sigma^2}\right\}.$$

Essendo le variabili i.i.d., possiamo scrivere la densità congiunta come il prodotto delle singole densità e quindi si ottiene

$$p(y \mid \mu) = \prod_{i=1}^{n} p(y_i \mid \mu).$$

Una volta osservati i dati y, la verosimiglianza diventa

$$\mathcal{L}(\mu \mid y) = \prod_{i=1}^{n} p(y_i \mid \mu) = \\ \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(y_1-\mu)^2}{2\sigma^2}\right\} \times \\ \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(y_2-\mu)^2}{2\sigma^2}\right\} \times \\ \vdots \\ \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{(y_n-\mu)^2}{2\sigma^2}\right\}. \tag{2.1}$$

Se viene scelta una densità a priori gaussiana, ciò fa sì che anche la densità a posteriori sia gaussiana. Supponiamo che

$$p(\mu) = \frac{1}{\tau_0\sqrt{2\pi}} \exp\left\{-\frac{(\mu-\mu_0)^2}{2\tau_0^2}\right\}, \tag{2.2}$$

ovvero che la distribuzione a priori di μ sia gaussiana con media μ_0 e varianza τ_0^2. Possiamo dire che μ_0 rappresenta il valore ritenuto più probabile per μ e τ_0^2 il grado di incertezza che abbiamo rispetto a tale valore.

Svolgendo una serie di passaggi algebrici, si arriva a

$$p(\mu \mid y) = \frac{1}{\tau_p\sqrt{2\pi}} \exp\left\{-\frac{(\mu-\mu_p)^2}{2\tau_p^2}\right\}, \tag{2.3}$$

dove

$$\mu_p = \frac{\frac{1}{\tau_0^2}\mu_0 + \frac{n}{\sigma^2}\bar{y}}{\frac{1}{\tau_0^2} + \frac{n}{\sigma^2}} \tag{2.4}$$

e

$$\tau_p^2 = \frac{1}{\frac{1}{\tau_0^2} + \frac{n}{\sigma^2}}. \tag{2.5}$$

In altri termini, se la distribuzione a priori per μ è gaussiana, la distribuzione a posteriori è anch'essa gaussiana con valore atteso (a posteriori) μ_p e varianza (a posteriori) τ_p^2 date dalle espressioni precedenti.

In conclusione, il risultato trovato indica che:

- il valore atteso a posteriori è una media pesata fra il valore atteso a priori μ_0 e la media campionaria $\bar{y}$; il peso della media campionaria è tanto maggiore tanto più è grande n (il numero di osservazioni) e τ_0^2 (l'incertezza iniziale);
- l'incertezza (varianza) a posteriori τ_p^2 è sempre più piccola dell'incertezza a priori τ_0^2 e diminuisce al crescere di n.

2.2 Il modello Normale con Stan

Per esaminare un esempio pratico, consideriamo i 30 valori BDI-II dei soggetti clinici di Zetsche et al. (2019):

```
df <- data.frame(
  y = c(
    26.0, 35.0, 30, 25, 44, 30, 33, 43, 22, 43,
    24, 19, 39, 31, 25, 28, 35, 30, 26, 31, 41,
    36, 26, 35, 33, 28, 27, 34, 27, 22
  )
)
```

Calcoliamo le statistiche descrittive del campione di dati:

```
df %>%
  summarise(
    sample_mean = mean(y),
    sample_sd = sd(y)
  )
#>   sample_mean sample_sd
#> 1       30.93     6.607
```

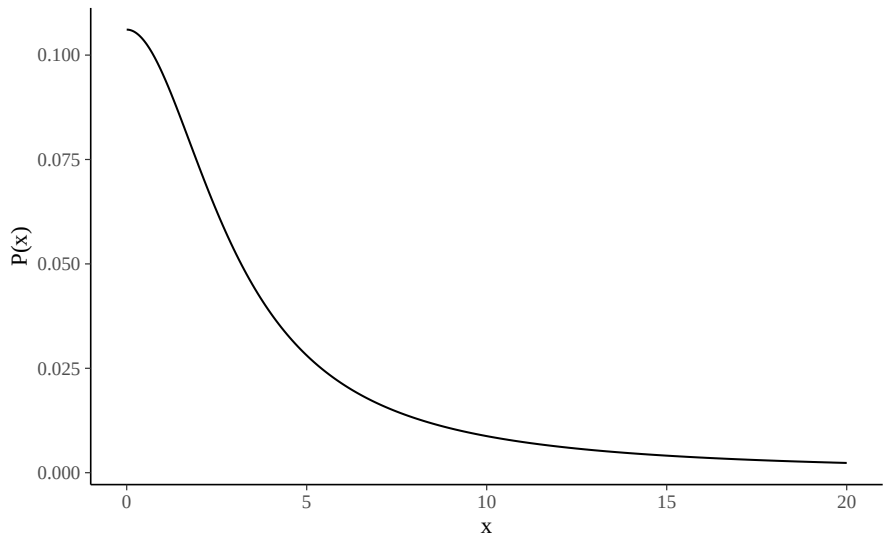
Nella discussione seguente assumeremo che μ e σ siano indipendenti. Assegneremo a μ una distribuzione a priori $\mathcal{N}(25, 2)$ e a σ una distribuzione a priori $Cauchy(0, 3)$.

Il modello statistico diventa:

$$
\begin{aligned}
Y_i &\sim \mathcal{N}(\mu, \sigma) \\
\mu &\sim \mathcal{N}(\mu_\mu = 25, \sigma_\mu = 2) \\
\sigma &\sim \text{Cauchy}(0, 3)
\end{aligned}
$$

In base al modello definito, la variabile casuale Y segue la distribuzione Normale di parametri μ e σ. Il parametro μ è sconosciuto e abbiamo deciso di descrivere la nostra incertezza relativa ad esso mediante una distribuzione a priori Normale con media uguale a 25 e deviazione standard pari a 2. L'incertezza relativa a σ è quantificata da una distribuzione a priori half-Cauchy(0, 5), come indicato nella figura seguente:

```
data.frame(x = c(0, 20)) %>%
  ggplot(aes(x)) +
  stat_function(
    fun = dcauchy,
    n = 1e3,
    args = list(location = 0, scale = 3)
  ) +
  ylab("P(x)") +
  theme(legend.position = "none")
```

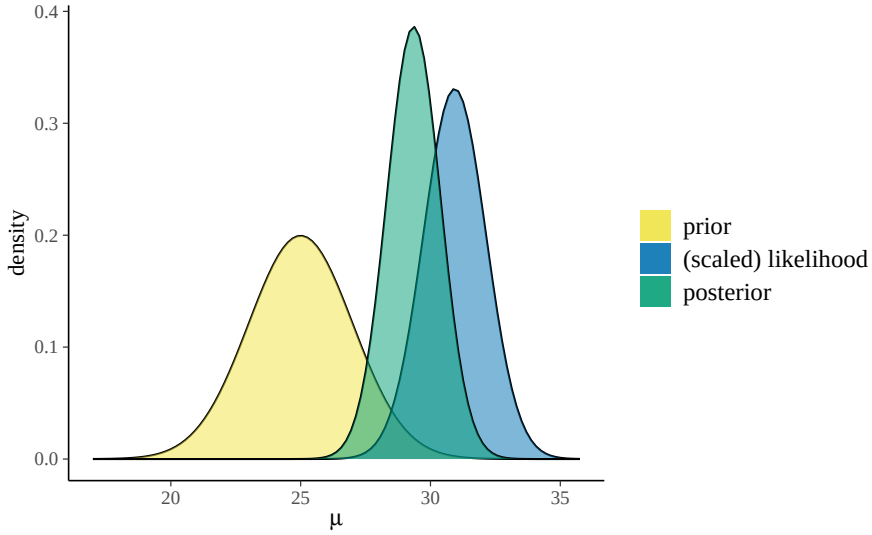


Dato che il modello è Normale-Normale, è possibile una soluzione analitica, come descritto in precedenza per il caso in cui σ è noto. In tali condizioni, la distribuzione a posteriori per μ può essere trovata con la funzione `bayesrules:::summarize_normal_normal()`:

```
bayesrules:::summarize_normal_normal(
  mean = 25,
  sd = 2,
  sigma = sd(df$y),
  y_bar = mean(df$y),
  n = 30
)
#>       model  mean  mode   var    sd
#> 1     prior 25.00 25.00 4.000 2.000
#> 2 posterior 29.35 29.35 1.067 1.033
```

La rappresentazione grafica della funzione a priori, della verosimiglianza e della distribuzione a posteriori per μ è fornita da:

```
bayesrules:::plot_normal_normal(
  mean = 25,
  sd = 2,
  sigma = sd(df$y),
  y_bar = mean(df$y),
  n = 30
)
```



La procedura MCMC utilizzata da Stan è basata su un campionamento Monte Carlo Hamiltoniano che non richiede l'uso di distribuzioni a priori coniugate. Pertanto per i parametri è possibile scegliere una qualunque distribuzione a priori arbitraria.

Per continuare con l'esempio, poniamoci il problema di trovare le distribuzioni a posteriori dei parametri μ e σ usando le funzioni del pacchetto `cmdstanr`. Il modello statistico descritto sopra si può scrivere in Stan nel modo seguente:

```
modelString = "
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real mu;
  real<lower=0> sigma;
}
model {
  mu ~ normal(25, 2);
  sigma ~ normal(0, 5);
  y ~ normal(mu, sigma);
```

```
}
"
writeLines(modelString, con = "code/normalmodel.stan")
```

Si noti che, nel modello, il parametro σ è considerato incognito.

Sistemiamo i dati nel formato appropriato per potere essere letti da Stan:

```
data_list <- list(
  N = length(df$y),
  y = df$y
)
```

Leggiamo il file in cui abbiamo salvato il codice Stan

```
file <- file.path("code", "normalmodel.stan")
```

compiliamo il modello

```
mod <- cmdstan_model(file)
```

ed eseguiamo il campionamento MCMC:

```
fit <- mod$sample(
  data = data_list,
  iter_sampling = 4000L,
  iter_warmup = 2000L,
  seed = SEED,
  chains = 4L,
  parallel_chains = 2L,
  refresh = 0,
  thin = 1
)
```

Le stime a posteriori dei parametri si ottengono con:

```
fit$summary(c("mu", "sigma"))
#> # A tibble: 2 x 10
#>   variable  mean median    sd   mad    q5   q95  rhat
#>   <chr>    <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 mu       29.3   29.3  1.11  1.10  27.4  31.0   1.00
#> 2 sigma     6.86   6.77 0.911 0.886  5.54  8.47  1.00
#> # ... with 2 more variables: ess_bulk <dbl>,
#> #   ess_tail <dbl>
```

oppure, dopo avere trasformato l'oggetto `fit` nel formato `stanfit`,

```
stanfit <- rstan::read_stan_csv(fit$output_files())
```

con

```
out <- rstantools::posterior_interval(
  as.matrix(stanfit),
  prob = 0.95
)
out
#>         2.5%   97.5%
#> mu     27.00  31.366
#> sigma   5.32   8.853
#> lp__  -77.04 -73.384
```

Possiamo dunque concludere, con un grado di certezza soggettiva del 95%, che siamo sicuri che la media della popolazione da cui abbiamo tratto i dati è compresa nell'intervallo [27, 31.37].

Considerazioni conclusive

Questo esempio ci mostra come calcolare l'intervallo di credibilità per la media di una v.c. gaussiana. La domanda più ovvia di analisi dei dati, dopo avere visto come creare l'intervallo di credibilità per la media di un gruppo, riguarda il confronto tra le medie di due gruppi. Questo però è un caso speciale di una tecnica di analisi dei dati più generale,

chiamate analisi di regressione lineare. Prima di discutere il problema del confronto tra le medie di due gruppi è dunque necessario esaminare il modello statistico di regressione lineare.

A

Simbologia di base

Per una scrittura più sintetica possono essere utilizzati alcuni simboli matematici.

- $\log(x)$: il logaritmo naturale di x.
- L'operatore logico booleano $\wedge$ significa "e" (congiunzione forte) mentre il connettivo di disgiunzione $\vee$ significa "o" (oppure) (congiunzione debole).
- Il quantificatore esistenziale $\exists$ vuol dire "esiste almeno un" e indica l'esistenza di almeno una istanza del concetto/oggetto indicato. Il quantificatore esistenziale di unicità $\exists!$ ("esiste soltanto un") indica l'esistenza di esattamente una istanza del concetto/oggetto indicato. Il quantificatore esistenziale $\nexists$ nega l'esistenza del concetto/oggetto indicato.
- Il quantificatore universale $\forall$ vuol dire "per ogni."
- $\mathcal{A}, \mathcal{S}$: insiemi.
- $x \in A$: x è un elemento dell'insieme A.
- L'implicazione logica "$\Rightarrow$" significa "implica" (se …allora). $P \Rightarrow Q$ vuol dire che P è condizione sufficiente per la verità di Q e che Q è condizione necessaria per la verità di P.
- L'equivalenza matematica " $\Longleftrightarrow$ " significa "se e solo se" e indica una condizione necessaria e sufficiente, o corrispondenza biunivoca.
- Il simbolo | si legge "tale che."
- Il simbolo $\triangleq$ (o $:=$) si legge "uguale per definizione."
- Il simbolo Δ indica la differenza fra due valori della variabile scritta a destra del simbolo.
- Il simbolo $\propto$ si legge "proporzionale a."
- Il simbolo $\approx$ si legge "circa."
- Il simbolo $\in$ della teoria degli insiemi vuol dire "appartiene" e indica l'appartenenza di un elemento ad un insieme. Il simbolo $\notin$ vuol dire "non appartiene."
- Il simbolo $\subseteq$ si legge "è un sottoinsieme di" (può coincidere con l'insieme stesso). Il simbolo $\subset$ si legge "è un sottoinsieme proprio di."

- Il simbolo # indica la cardinalità di un insieme.
- Il simbolo $\cap$ indica l'intersezione di due insiemi. Il simbolo $\cup$ indica l'unione di due insiemi.
- Il simbolo $\emptyset$ indica l'insieme vuoto o evento impossibile.
- In matematica, argmax identifica l'insieme dei punti per i quali una data funzione raggiunge il suo massimo. In altre parole, $\text{argmax}_x f(x)$ è l'insieme dei valori di x per i quali $f(x)$ raggiunge il valore più alto.
- a, c, α, γ: scalari.
- x, y: vettori.
- X, Y: matrici.
- $X \sim p$: la variabile casuale X si distribuisce come p.
- $p(\cdot)$: distribuzione di massa o di densità di probabilità.
- $p(y \mid x)$: la probabilità o densità di y dato x, ovvero $p(y = Y \mid x = X)$.
- $f(x)$: una funzione arbitraria di x.
- $f(X; \theta, \gamma)$: f è una funzione di X con parametri θ, γ. Questa notazione indica che X sono i dati che vengono passati ad un modello di parametri θ, γ.
- $\mathcal{N}(\mu, \sigma^2)$: distribuzione gaussiana di media μ e varianza $sigma^2$.
- Beta(α, β): distribuzione Beta di parametri α e β.
- $\mathcal{U}(a, b)$: distribuzione uniforme con limite inferiore a e limite superiore b.
- Cauchy(α, β): distribuzione di Cauchy di parametri α (posizione: media) e β (scala: radice quadrata della varianza).
- $\mathcal{B}(p)$: distribuzione di Bernoulli di parametro p (probabilità di successo).
- Bin(n, p): distribuzione binomiale di parametri n (numero di prove) e p (probabilità di successo).
- $\mathbb{KL}(p \mid\mid q)$: la divergenza di Kullback-Leibler da p a q.

B

Numeri binari, interi, razionali, irrazionali e reali

B.1 Numeri binari

I numeri più semplici sono quelli binari, cioè zero o uno. Useremo spesso numeri binari per indicare se qualcosa è vero o falso, presente o assente. I numeri binari sono molto utili per ottenere facilmente delle statistiche riassuntive in R.Supponiamo di chiedere a 10 studenti "Ti piacciono i mirtilli?" Poniamo che le risposte siano le seguenti:

```
opinion <- c(
  "Yes", "No", "Yes", "No", "Yes", "No", "Yes",
  "Yes", "Yes", "Yes"
)
opinion
#>  [1] "Yes" "No"  "Yes" "No"  "Yes" "No"  "Yes" "Yes"
#>  [9] "Yes" "Yes"
```

Tali risposte possono essere ricodificate nei termini di valori di verità, ovvero, vero e falso, generalmente denotati rispettivamente come 1 e 0. In R tale ricodifica può essere effettuata mediante l'operatore == che è un test per l'uguaglianza e restituisce il valore logico VERO se i due oggetti valutati sono uguali e FALSO se non lo sono:

```
opinion <- opinion == "Yes"
opinion
#>  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE
#>  [9]  TRUE  TRUE
```

R considera i valori di verità e i numeri binari in modo equivalente, con TRUE uguale a 1 e FALSE uguale a zero. Di conseguenza, possiamo effettuare operazioni algebriche sui valori logici VERO e FALSO. Nell'esempio, possiamo sommare i valori di verità e dividere per 10

```
sum(opinion) / length(opinion)
#> [1] 0.7
```

in modo tale da calcolare una propozione, il che ci consente di concludere che 7 risposte su 10 sono positive.

B.2 Numeri interi

Un numero intero è un numero senza decimali. Si dicono **naturali** i numeri che servono a contare, come 1, 2, … L'insieme dei numeri naturali si indica con il simbolo $\mathbb{N}$. È anche necessario introdurre i numeri con il segno per poter trattare grandezze negative. Si ottengono così l'insieme numerico dei numeri interi relativi: $\mathbb{Z} = \{0, \pm 1, \pm 2, ... \}$

B.3 Numeri razionali

I numeri razionali sono i numeri frazionari m/n, dove $m, n \in N$, con $n \neq 0$. Si ottengono così i numeri razionali: $\mathbb{Q} = \{\frac{m}{n} \,|\, m, n \in \mathbb{Z}, n \neq 0\}$. È evidente che $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q}$. Anche in questo caso è necessario poter trattare grandezze negative. I numeri razionali non negativi sono indicati con $\mathbb{Q}^+ = \{q \in \mathbb{Q} \,|\, q \geq 0\}$.

B.4 Numeri irrazionali

Tuttavia, non tutti i punti di una retta r possono essere rappresentati mediante i numeri interi e razionali. È dunque necessario introdurre un'altra classe di numeri. Si dicono *irrazionali*, e sono denotati con $\mathbb{R}$, i

numeri che possono essere scritti come una frazione a/b, con a e b interi e b diverso da 0. I numeri irrazionali sono i numeri illimitati e non periodici che quindi non possono essere espressi sotto forma di frazione. Per esempio, $\sqrt{2}$, $\sqrt{3}$ e $\pi = 3,141592...$ sono numeri irrazionali.

B.5 Numeri reali

I punti della retta r sono quindi "di più" dei numeri razionali. Per poter rappresentare tutti i punti della retta abbiamo dunque bisogno dei numeri *reali*. I numeri reali possono essere positivi, negativi o nulli e comprendono, come casi particolari, i numeri interi, i numeri razionali e i numeri irrazionali. Spesso in statisticac il numero dei decimali indica il grado di precisione della misurazione.

B.6 Intervalli

Un intervallo si dice chiuso se gli estremi sono compresi nell'intervallo, aperto se gli estremi non sono compresi. Le caratteristiche degli intervalli sono riportate nella tabella seguente.

Intervallo		
chiuso	$[a, b]$	$a \leq x \leq b$
aperto	(a, b)	$a < x < b$
chiuso a sinistra e aperto a destra	$[a, b)$	$a \leq x < b$
aperto a sinistra e chiuso a destra	$(a, b]$	$a < x \leq b$

C

Insiemi

Un insieme (o collezione, classe, gruppo, …) è un concetto primitivo, ovvero è un concetto che già possediamo. Georg Cantor l'ha definito nel modo seguente: *un insieme è una collezione di oggetti, determinati e distinti, della nostra percezione o del nostro pensiero, concepiti come un tutto unico; tali oggetti si dicono elementi dell'insieme.*

Mentre non è rilevante la natura degli oggetti che costituiscono l'insieme, ciò che importa è distinguere se un dato oggetto appartenga o meno ad un insieme. Deve essere vera una delle due possibilità: il dato oggetto è un elemento dell'insieme considerato oppure non è elemento dell'insieme considerato. Due insiemi A e B si dicono uguali se sono formati dagli stessi elementi, anche se disposti in ordine diverso: $A = B$. Due insiemi A e B si dicono diversi se non contengono gli stessi elementi: $A \neq B$. Ad esempio, i seguenti insiemi sono uguali:

$$\{1, 2, 3\} = \{3, 1, 2\} = \{1, 3, 2\} = \{1, 1, 1, 2, 3, 3, 3\}.$$

Gli insiemi sono denotati da una lettera maiuscola, mentre le lettere minuscole, di solito, designano gli elementi di un insieme. Per esempio, un generico insieme A si indica con

$$A = \{a_1, a_2, \dots, a_n\}, \quad \text{con } n > 0.$$

La scrittura $a \in A$ dice che a è un elemento di A. Per dire che b non è un elemento di A si scrive $b \notin A$.

Per quegli insiemi i cui elementi soddisfano una certa proprietà che li caratterizza, tale proprietà può essere usata per descrivere più sinteticamente l'insieme:

$$A = \{x \mid \text{proprietà posseduta da } x\},$$

che si legge come "A è l'insieme degli elementi x per cui è vera la proprietà indicata." Per esempio, per indicare l'insieme A delle coppie di numeri reali (x, y) che appartengono alla parabola $y = x^2 + 1$ si può scrivere:

$$A = \{(x, y) \mid y = x^2 + 1\}.$$

Dati due insiemi A e B, diremo che A è un *sottoinsieme* di B se e solo se tutti gli elementi di A sono anche elementi di B:

$$A \subseteq B \iff (\forall x \in A \Rightarrow x \in B).$$

Se esiste almeno un elemento di B che non appartiene ad A allora diremo che A è un *sottoinsieme proprio* di B:

$$A \subset B \iff (A \subseteq B, \exists\, x \in B \mid x \notin A).$$

Un altro insieme, detto *insieme delle parti*, o insieme potenza, che si associa all'insieme A è l'insieme di tutti i sottoinsiemi di A, inclusi l'insieme vuoto e A stesso. Per esempio, per l'insieme $A = \{a, b, c\}$, l'insieme delle parti è:

$$\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{c, b\}, \{a, b, c\}\}.$$

C.1 Operazioni tra insiemi

Si definisce *intersezione* di A e B l'insieme $A \cap B$ di tutti gli elementi x che appartengono ad A e contemporaneamente a B:

$$A \cap B = \{x \mid x \in A \wedge x \in B\}.$$

Si definisce *unione* di A e B l'insieme $A \cup B$ di tutti gli elementi x che appartengono ad A o a B, cioè

$$A \cup B = \{x \mid x \in A \vee x \in B\}.$$

Differenza. Si indica con $A \setminus B$ l'insieme degli elementi di A che non appartengono a B:

$$A \quad B = \{x \mid x \in A \wedge x \notin B\}.$$

Insieme complementare. Nel caso che sia $B \subseteq A$, l'insieme differenza $A \quad B$ è detto insieme complementare di B in A e si indica con B^C.

Dato un insieme S, una *partizione* di S è una collezione di sottoinsiemi di S, $S_1, \ldots, S_k$, tali che

$$S = S_1 \cup S_2 \cup \ldots S_k$$

e

$$S_i \cap S_j, \quad \text{con } i \neq j.$$

La relazione tra unione, intersezione e insieme complementare è data dalle leggi di DeMorgan:

$$(A \cup B)^c = A^c \cap B^c,$$
$$(A \cap B)^c = A^c \cup B^c.$$

C.2 Diagrammi di Eulero-Venn

In molte situazioni è utile servirsi dei cosiddetti diagrammi di Eulero-Venn per rappresentare gli insiemi e verificare le proprietà delle operazioni tra insiemi (si veda la figura C.1. I diagrammi di Venn sono così nominati in onore del matematico inglese del diciannovesimo secolo John Venn anche se Leibnitz e Eulero avevano già in precedenza utilizzato rappresentazioni simili. In tale rappresentazione, gli insiemi sono individuati da regioni del piano delimitate da una curva chiusa. Nel caso di insiemi finiti, è possibile evidenziare esplicitamente alcuni elementi di un insieme mediante punti, quando si possono anche evidenziare tutti gli elementi degli insiemi considerati.

I diagrammi di Eulero-Venn che forniscono una dimostrazione delle leggi di DeMorgan sono forniti nella figura C.2.

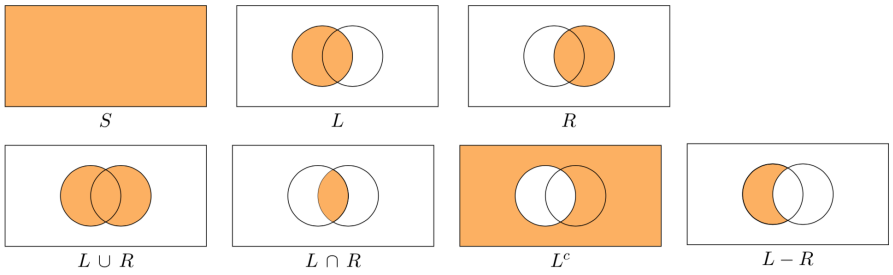


Figura C.1: *In tutte le figure S è la regione delimitata dal rettangolo, L è la regione all'interno del cerchio di sinistra e R è la regione all'interno del cerchio di destra. La regione evidenziata mostra l'insieme indicato sotto ciascuna figura.*

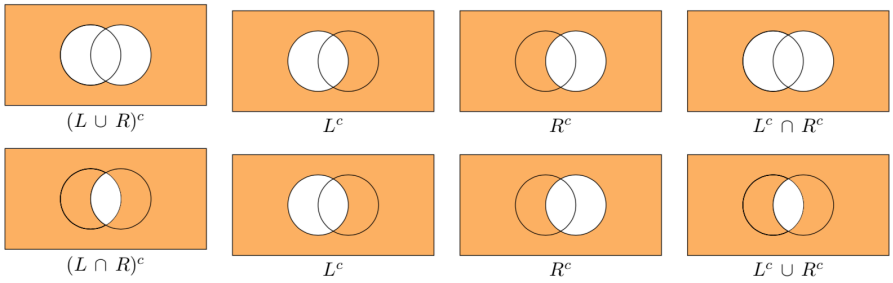


Figura C.2: *Dimostrazione delle leggi di DeMorgan.*

C.3 Coppie ordinate e prodotto cartesiano

Una coppia ordinata (x, y) è l'insieme i cui elementi sono $x \in A$ e $y \in B$ e nella quale x è la prima componente (o prima coordinata), y la seconda. L'insieme di tutte le coppie ordinate costruite a partire dagli insiemi A e B viene detto **prodotto cartesiano**:

$$A \times B = \{(x, y) \mid x \in A \wedge y \in B\}.$$

Ad esempio, sia $A = \{1, 2, 3\}$ e $B = \{a, b\}$. Allora,

$$\{1, 2\} \times \{a, b, c\} = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}.$$

C.4 Cardinalità

Si definisce *cardinalità* (o potenza) di un insieme finito il numero degli elementi dell'insieme. Viene indicata con $|A|, \#(A)$ o $\mathrm{c}(A)$.

D

Simbolo di somma (sommatorie)

Le somme si incontrano costantemente in svariati contesti matematici e statistici quindi abbiamo bisogno di una notazione adeguata che ci consenta di gestirle. La somma dei primi n numeri interi può essere scritta come $1 + 2 + \cdots + (n - 1) + n$, dove '...' ci dice di completare la sequenza definita dai termini che vengono prima e dopo. Ovviamente, una notazione come $1 + 7 + \cdots + 73.6$ non avrebbe alcun senso senza qualche altro tipo di precisazione. In generale, nel seguito incontreremo delle somme nella forma

$$x_1 + x_2 + \cdots + x_n,$$

dove x_i è un numero che è stato definito altrove. La notazione precedente, che fa uso dei tre puntini di sospensione, è utile in alcuni contesti ma in altri risulta ambigua. Pertanto la notazione di uso corrente è del tipo

$$\sum_{i=1}^{n} x_i$$

e si legge "sommatoria per i che va da 1 a n di x_i". Il simbolo $\sum$ (lettera sigma maiuscola dell'alfabeto greco) indica l'operazione di somma, il simbolo x_i indica il generico addendo della sommatoria, le lettere 1 ed n indicano i cosiddetti *estremi della sommatoria*, ovvero l'intervallo (da 1 fino a n estremi inclusi) in cui deve variare l'indice i allorché si sommano gli addendi x_i. Solitamente l'estremo inferiore è 1 ma potrebbe essere qualsiasi altri numero $m < n$. Quindi

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n.$$

Per esempio, se i valori x sono $\{3, 11, 4, 7\}$, si avrà

$$\sum_{i=1}^{4} x_i = 3 + 11 + 4 + 7 = 25$$

laddove $x_1 = 3$, $x_2 = 11$, eccetera. La quantità x_i nella formula precedente si dice l'*argomento* della sommatoria, mentre la variabile i, che prende i valori naturali successivi indicati nel simbolo, si dice *indice* della sommatoria.

La notazione di sommatoria può anche essere fornita nella forma seguente

$$\sum_{P(i)} x_i$$

dove $P(i)$ è qualsiasi proposizione riguardante i che può essere vera o falsa. Quando è ovvio che si vogliono sommare tutti i valori di n osservazioni, la notazione può essere semplificata nel modo seguente: $\sum_i x_i$ oppure $\sum x_i$. Al posto di i si possono trovare altre lettere: $k, j, l, ...,$.

D.1 Manipolazione di somme

È conveniente utilizzare le seguenti regole per semplificare i calcoli che coinvolgono l'operatore della sommatoria.

D.1.1 Proprietà 1

La sommatoria di n valori tutti pari alla stessa costante a è pari a n volte la costante stessa:

$$\sum_{i=1}^{n} a = \underbrace{a + a + \cdots + a}_{n \text{ volte}} = na.$$

D.1.2 Proprietà 2 (proprietà distributiva)

Nel caso in cui l'argomento contenga una costante, è possibile riscrivere la sommatoria. Ad esempio con

$$\sum_{i=1}^{n} ax_i = ax_1 + ax_2 + \cdots + ax_n$$

è possibile raccogliere la costante a e fare $a(x_1 + x_2 + \cdots + x_n)$. Quindi possiamo scrivere

$$\sum_{i=1}^{n} ax_i = a \sum_{i=1}^{n} x_i.$$

D.1.3 Proprietà 3 (proprietà associativa)

Nel caso in cui

$$\sum_{i=1}^{n} (a + x_i) = (a + x_1) + (a + x_1) + ... (a + x_n)$$

si ha che

$$\sum_{i=1}^{n} (a + x_i) = na + \sum_{i=1}^{n} x_i.$$

È dunque chiaro che in generale possiamo scrivere

$$\sum_{i=1}^{n} (x_i + y_i) = \sum_{i=1}^{n} x_i + \sum_{i=1}^{n} y_i.$$

D.1.4 Proprietà 4

Se deve essere eseguita un'operazione algebrica (innalzamento a potenza, logaritmo, ecc.) sull'argomento della sommatoria, allora tale operazione algebrica deve essere eseguita prima della somma. Per esempio,

$$\sum_{i=1}^{n} x_i^2 = x_1^2 + x_2^2 + \cdots + x_n^2 \neq \left(\sum_{i=1}^{n} x_i \right)^2.$$

D.1.5 Proprietà 5

Nel caso si voglia calcolare $\sum_{i=1}^{n} x_i y_i$, il prodotto tra i punteggi appaiati deve essere eseguito prima e la somma dopo:

$$\sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n,$$

infatti, $a_1 b_1 + a_2 b_2 \neq (a_1 + a_2)(b_1 + b_2)$.

D.2 Doppia sommatoria

È possibile incontrare la seguente espressione in cui figurano una doppia sommatoria e un doppio indice:

$$\sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij}.$$

La doppia sommatoria comporta che per ogni valore dell'indice esterno, i da 1 ad n, occorre sviluppare la seconda sommatoria per j da 1 ad m. Quindi,

$$\sum_{i=1}^{3} \sum_{j=4}^{6} x_{ij} = (x_{1,4}+x_{1,5}+x_{1,6})+(x_{2,4}+x_{2,5}+x_{2,6})+(x_{3,4}+x_{3,5}+x_{3,6}).$$

Un caso particolare interessante di doppia sommatoria è il seguente:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_i y_j$$

Si può osservare che nella sommatoria interna (quella che dipende dall'indice j), la quantità x_i è costante, ovvero non dipende dall'indice (che è j). Allora possiamo estrarre x_i dall'operatore di sommatoria interna e scrivere

$$\sum_{i=1}^{n} \left(x_i \sum_{j=1}^{n} y_j \right).$$

Allo stesso modo si può osservare che nell'argomento della sommatoria esterna la quantità costituita dalla sommatoria in j non dipende dall'indice i e quindi questa quantità può essere estratta dalla sommatoria esterna. Si ottiene quindi

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_i y_j = \sum_{i=1}^{n}\left(x_i \sum_{j=1}^{n} y_j\right) = \sum_{i=1}^{n}\ x_i \sum_{j=1}^{n} y_j.$$

Esercizio D.1. Si verifichi quanto detto sopra nel caso particolare di $x = \{2, 3, 1\}$ e $y = \{1, 4, 9\}$, svolgendo prima la doppia sommatoria per poi verificare che quanto così ottenuto sia uguale al prodotto delle due sommatorie.

$$\begin{aligned}\sum_{i=1}^{3}\sum_{j=1}^{3} x_i y_j &= x_1y_1 + x_1y_2 + x_1y_3 + x_2y_1 + x_2y_2 + x_2y_3 + x_3y_1 + x_3y_2 + x_3y_3 \\ &= 2 \times (1 + 4 + 9) + 3 \times (1 + 4 + 9) + 2 \times (1 + 4 + 9) = 84,\end{aligned}$$

ovvero

$$(2 + 3 + 1) \times (1 + 4 + 9) = 84.$$

D.3 Sommatorie (e produttorie) e operazioni vettoriali in R

Si noti che la notazione

$$\sum_{n=0}^{4} 3n$$

non è altro che un ciclo `for`:

```
sum <- 0
for (n in 0:4) {
  sum = sum + 3 * n
}
sum
#> [1] 30
```

In maniera equivalente, e più semplice, possiamo scrivere

```
sum(3 * (0:4))
#> [1] 30
```

Allo stesso modo, la notazione

$$\prod_{n=1}^{4} 2n$$

è anch'essa equivalente al ciclo `for`

```
prod <- 1
for (n in 1:4) {
  prod <- prod * 2 * n
}
prod
#> [1] 384
```

che si può scrivere, più semplicemente, come

```
prod(2 * (1:4))
#> [1] 384
```

In entrambi i casi precedenti, abbiamo sostituito le operazioni aritmetiche eseguite all'interno di un ciclo `for` con le stesse operazioni aritmetiche eseguite sui vettori elemento per elemento.

E

Cenni di calcolo combinatorio

La derivazione del coefficiente binomiale richiede l'uso di alcune nozioni di calcolo combinatorio. Iniziamo con il definire il concetto di permutazione.

E.1 Permutazioni semplici

Una *permutazione semplice* di un insieme di oggetti è un allineamento di n oggetti su n posti nel quale ogni oggetto viene presentato una ed una sola volta. Le permutazioni semplici si indicano con il simbolo P_n.

Il numero delle permutazioni semplici di n elementi distinti è uguale a

$$P_n = n! \tag{E.1}$$

Per esempio, nel caso dell'insieme $A = \{a, b, c\}$, le permutazioni possibili sono:

$$\{a, b, c\},\ \{a, c, b\},\ \{b, c, a\},\ \{b, a, c\},\ \{c, a, b\},\ \{c, b, a\},$$

Il numero di permutazioni di A è

$$P_n = P_3 = 3! = 3 \cdot 2 \cdot 1 = 6.$$

E.2 Disposizioni semplici

Supponiamo ora di voler selezionare una sequenza di k oggetti da un insieme di n e che l'ordine degli oggetti abbia importanza. Si chiamano *disposizioni semplici* di n elementi distinti presi a k a k (o disposizioni

della classe k) tutti i raggruppamenti che si possono formare con gli oggetti dati in modo che qualsiasi raggruppamento ne contenga k tutti distinti tra loro (ovvero, senza ripetizione) e che due raggruppamenti differiscano tra loro per qualche oggetto oppure per l'ordine secondo il quale gli oggetti si susseguono. Le disposizioni semplici della classe k si indicano con $D_{n,k}$.

Il numero delle disposizioni semplici di n elementi distinti della classe k è uguale a

$$D_{n,k} = \frac{n!}{(n-k)!}. \tag{E.2}$$

Per esempio, nel caso dell'insieme: $A = \{a, b, c\}$, le disposizioni semplici di classe 2 sono:

$$\{a, b\},\ \{b, a\},\ \{a, c\},\ \{c, a\},\ \{b, c\},\ \{c, b\}$$

Il numero di disposizioni semplici di classe 2 è

$$D_{n,k} = \frac{n!}{(n-k)!} = 3 \cdot 2 = 6.$$

E.3 Combinazioni semplici

Avendo trovato il modo per contare il numero delle disposizioni semplici di n elementi distinti della classe k, dobbiamo ora trasformare la (E.2) in modo da ignorare l'ordine degli elementi di ciascun sottoinsieme. Le *combinazioni semplici* di n elementi a k a k ($k \leq n$) sono tutti i sottoinsiemi di k elementi di un dato insieme di n elementi, tutti distinti tra loro. Le combinazioni semplici differiscono dalle disposizioni semplici per il fatto che le disposizioni semplici tengono conto dell'ordine di estrazione mentre nelle combinazioni semplici si considerano distinti solo i raggruppamenti che differiscono almeno per un elemento.

Gli elementi di ciascuna combinazione di k oggetti possono essere ordinati tra loro in $k!$ modi diversi, per cui il numero delle combinazioni semplici è dato dal numero di disposizioni semplici $D_{n,k}$ diviso per il numero di permutazioni semplici P_k dei k elementi. Il numero delle com-

binazioni semplici di n elementi distinti della classe k è dunque uguale a

$$C_{n,k} = \frac{D_{n,k}}{P_k} = \frac{n!}{k!(n-k)!}. \tag{E.3}$$

Il numero delle combinazioni semplici $C_{n,k}$ è spesso detto coefficiente binomiale e indicato con il simbolo $\binom{n}{k}$ che si legge "n su k".

Per l'insieme $A = \{a, b, c\}$, le combinazioni semplici di classe 2 sono

$$\{a, b\},\ \{a, c\},\ \{b, c\},$$

Il numero di combinazioni semplici di classe 2 è dunque uguale a tre:

$$C_{n,k} = \binom{n}{k} = \binom{3}{2} = 3.$$

F

Esponenziali e logaritmi

Potenze ad esponente reale

Per un qualsiasi numero razionale $\frac{m}{n}$ (in cui $n > 0$) si ha

$$a^{\frac{m}{n}} = \sqrt[n]{a^m}$$

per numeri a reali positivi.

Proprietà

Se a, b sono reali positivi ed x, y reali qualsiasi, si ha

- $a^0 = 1$ e $a^{-x} = \frac{1}{a^x}$,
- $a^x a^y = a^{x+y}$ e $\frac{a^x}{a^y} = a^{x-y}$,
- $a^x b^x = (ab)^x$ e $\frac{a^x}{b^x} = \left(\frac{a}{b}\right)^x$,
- $(a^x)^y = a^{xy}$.

F.1 Funzione esponenziale

Definizione F.1. La funzione esponenziale con base a è

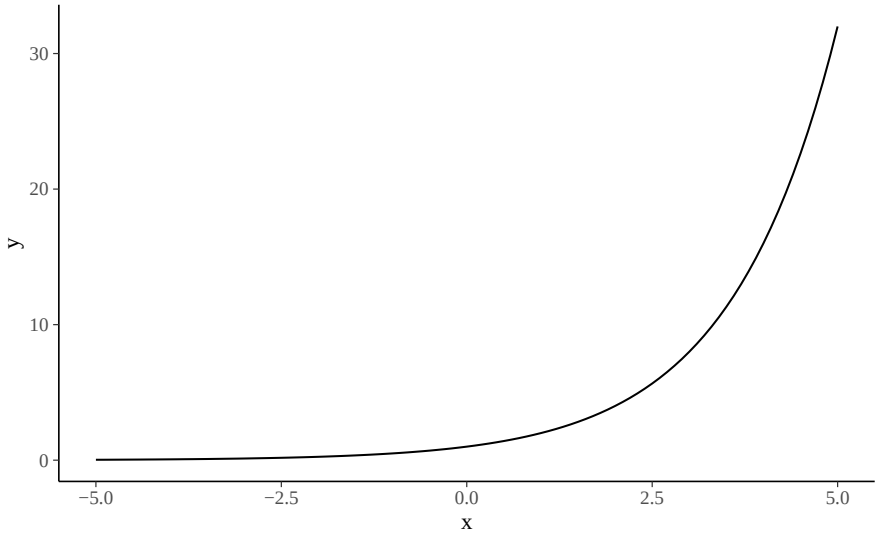
$$f(x) = a^x \tag{F.1}$$

dove $a > 0$, $a \neq 1$ e x è qualsiasi numero reale.

La base $a = 1$ è esclusa perché produce $f(x) = 1^x = 1$, la quale è una costante, non una funzione esponenziale.

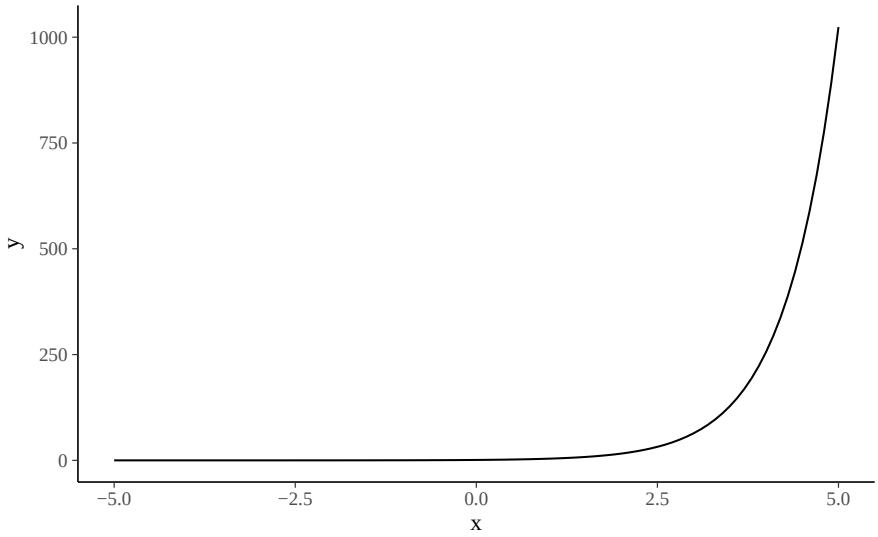
Per esempio, un grafico della funzione esponenziale di base 2 si trova con

```
exp_base2 = function(x){2^x}
tibble(x = c(-5, 5)) %>%
ggplot(aes(x = x)) +
  stat_function(fun = exp_base2)
```



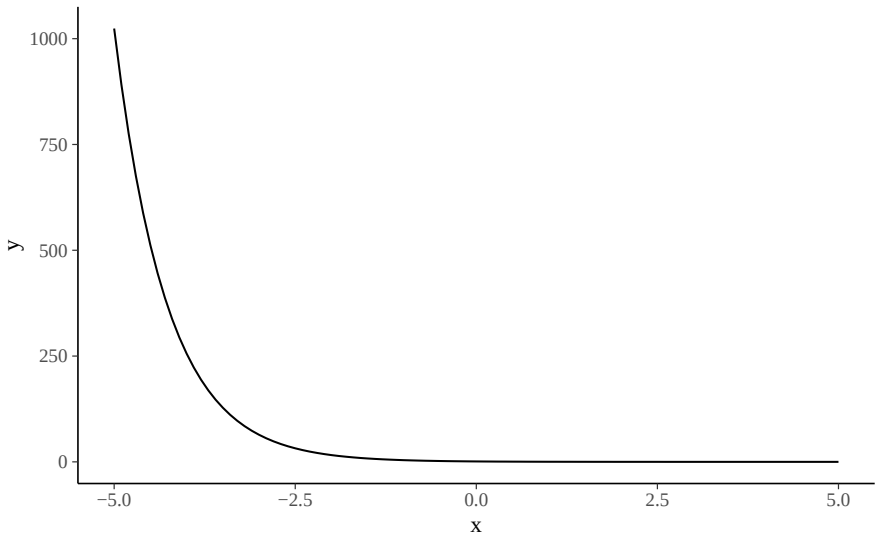
Se usiamo la base 4 troviamo

```
exp_base4 = function(x){4^x}
tibble(x = c(-5, 5)) %>%
ggplot(aes(x = x)) +
  stat_function(fun = exp_base4)
```



Oppure

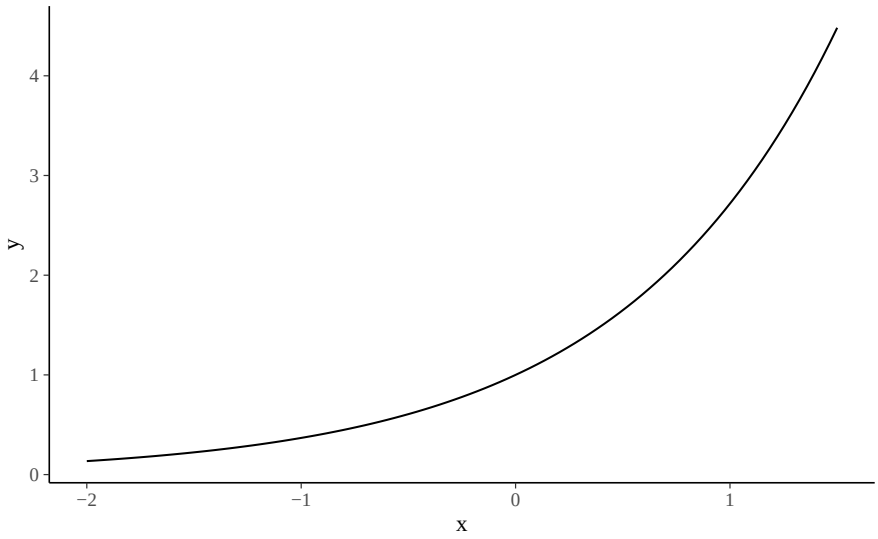
```
exp_base4 = function(x){4^-{x}}
tibble(x = c(-5, 5)) %>%
ggplot(aes(x = x)) +
  stat_function(fun = exp_base4)
```



In molte applicazioni la scelta più conveniente per la base è il numero irrazionale $e = 2.718281828\ldots$. Questo numero è chiamato la *base naturale*. La funzione $f(x) = e^x$ è chiamata *funzione esponenziale naturale*.

Per esempio, abbiamo

```
exp_base_e= function(x){exp(x)}
tibble(x = c(-2, 1.5)) %>%
ggplot(aes(x = x)) +
  stat_function(fun = exp_base_e)
```



Logaritmi

Dati due numeri reali $b > 0$ e $a > 0$ con $a \neq 1$, l'equazione esponenziale $a^x = b$ ammette sempre una ed una sola soluzione. Tale soluzione è detta *logaritmo in base a di b* ed è indicata con la scrittura $\log_a b$, dove b è detto *argomento* del logaritmo. In altri termini, per definizione si ha

$$x = \log_a b \iff a^x = b$$

dove deve essere $a > 0,\ a \neq 1,\ b > 0$.

Quando valutiamo i logaritmi, dobbiamo ricordare che un logaritmo è un esponente: il logaritmo in base a di b, $\log_a b$, è l'esponente da attribuire alla base a per ottenere l'argomento b. Le seguenti equazioni sono dunque equivalenti:

$$y = \log_a x \qquad x = a^y.$$

La prima equazione è in forma logaritmica e la seconda è in forma esponenziale. Ad esempio, l'equazione logaritmica $2 = \log_3 9$ può essere riscritta in forma esponenziale come $9 = 3^2$.

Esempio F.1. Scrivendo l'argomento come potenza della base si ottiene

- $\log_2 8 = \log_2 2^3 = 3$
- $\log_3 \sqrt[7]{3^{20}} = \log_3 3^{\frac{20}{7}} = \frac{20}{7}$
- $\log_{0.1} 0.01 = \log_{\frac{1}{10}} \frac{1}{100} = \log_{\frac{1}{10}} \left(\frac{1}{10}\right)^2 = 2$

Proprietà

Nell'operare con i logaritmi si procede spesso mediante le loro proprietà, che costituiscono una rilettura in termini di logaritmi delle proprietà delle potenze: se a, b sono numeri reali positivi diversi da 1 ed x, y reali positivi qualunque, allora

- $\log_a(xy) = \log_a x + \log_a y$,
- $\log_a \left(\frac{x}{y}\right) = \log_a x - \log_a y$,
- $\log_a (x^\alpha) = \alpha \log_a x, \quad \forall \alpha$ reale,
- $\log_a x = \frac{\log_b x}{\log_b a}$ (cambiamento di base).

Esempio F.2.

$$\begin{aligned} \log_a(x+1) - \log_a x - 2\log_a 2 &= \log_a(x+1) - (\log_a x + \log_a 2^2) \\ &= \log_a(x+1) - \log_a 4x \\ &= \log_a \frac{x+1}{4x}. \end{aligned}$$

F.2 Funzione logaritmica

La funzione logaritmica è la funzione inversa della funzione esponenziale.

Definizione F.2. Siano $a > 0$, $a \neq 1$. Per $x > 0$

$$y = \log_a x \quad \text{se e solo se } x = a^y. \tag{F.2}$$

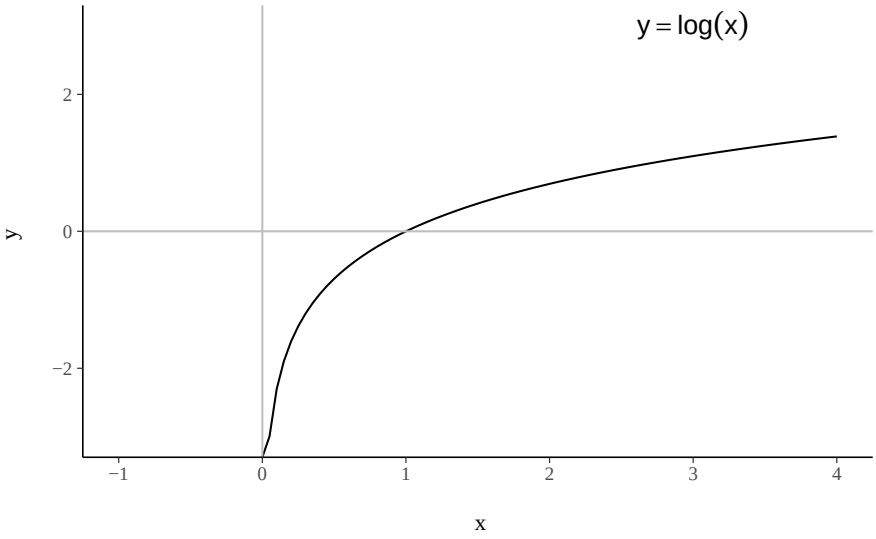
La funzione data da

$$f(x) = \log_a x \tag{F.3}$$

è chiamata funzione logaritmica.

Per esempio, abbiamo

```
log_funct <- function(x){
  log(x)
}
ggplot(tibble(x = c(-0.5, 4)), aes(x = x)) +
  stat_function(fun = log_funct) +
  xlim(c(-1, 4)) +
  ylim(-3, 3) +
  labs(x = "\n x", y = "y \n") +
  annotate("text", x = 3, y = 3, parse = TRUE, size = 5, fontface = "bold",
           label="y == log(x)") +
  geom_hline(yintercept = 0, colour = "gray") +
  geom_vline(xintercept = 0, colour = "gray")
```


y = log(x)
2
0
−2
y
−1
0
1
2
3
4
x

G

La Normale motivata dal metodo dei minimi quadrati

La distribuzione Normale fu scoperta da Gauss nel 1809 e, nella derivazione di Gauss, è intimamente legata al metodo dei minimi quadrati. Vediamo come Gauss arrivò alla definizione della densità Normale.

Tra il 1735 e il 1754 l'Accademia di Francia effettuò quattro misurazioni della lunghezza di un arco di meridiano a latitudini diverse con lo scopo di determinare la figura della Terra.[1] Papa Benedetto XIV volle contribuire a questo progetto e nel 1750 incaricò Roger Joseph Boscovich (1711—1787) e il gesuita inglese Christopher Maire di misurare un arco di meridiana nei pressi di Roma e contemporaneamente di costruire una nuova mappa dello Stato Pontificio. Il loro rapporto fu pubblicato nel 1755.

La relazione tra lunghezza d'arco e latitudine per archi piccoli è approssimativamente $y = \alpha + \beta x$, dove y è la lunghezza dell'arco e $x = sin^2 L$, dove L è la latitudine del punto medio dell'arco. Il problema di Boscovich era quello di stimare α e β da cinque osservazioni di (x, y).

Nel 1757 pubblicò una sintesi del rapporto del 1755 in cui proponeva di risolvere il problema di riconciliare le relazioni lineari inconsistenti mediante la minimizzazione della somma dei valori assoluti dei residui, sotto il vincolo che la somma dei residui fosse uguale a zero. In altre parole, Boscovich propose di minimizzare la quantità $\sum \mid y_i - a - bx_i \mid$ rispetto ad a e b sotto il vincolo $(y_i - a - bx_i) = 0$. Boscovich fu il primo a formulare un metodo per adattare una retta ai dati descritti da un diagramma a dispersione, laddove l'orientamento della retta dipende dalla minimizzazione di una funzione dei residui. La formulazione e la soluzione di Boscovich erano puramente verbali ed era accompagnata da un diagramma che spiegava il metodo di minimizzazione.

[1] L'espressione "figura della Terra" è utilizzata in geodesia per indicare la precisione con cui sono definite la dimensione e la forma della Terra.

Nella *Mécanique Céleste*, Laplace (1749, 1827) ritornò sul problema di Boscovich e mostrò in maniera formale come sia possibile minimizzare la quantità $\sum w_i \mid y_i - a - bx_i \mid$. Il metodo della minimizzazione del valore assoluto degli scarti presentava degli svantaggi rispetto al metodo dei minimi quadrati: (1) la stima della pendenza della retta era complicata da calcolare e (2) il metodo era limitato a una sola variabile indipendente. Il metodo scomparve quindi dalla pratica statistica fino alla seconda metà del XX secolo quando venne riproposto nel contesto della discussione della robustezza delle stime.

In seguito, tale problema venne ripreso da Legendre. Il suo *Nouvelle methods pour la determinazione des orbites des comètes* contiene un'appendice (pp. 72-80) intitolata *Sur la méthode des moindres carrés*, in cui per la prima volta il metodo dei minimi quadrati viene presentato come un metodo algebrico per l'adattamento di un modello lineare ai dati. Legendre scrive *"Tra tutti i princìpi che si possono proporre a questo scopo, credo che non ce ne sia uno più generale, più esatto e più facile da applicare di quello di cui ci siamo serviti nelle precedenti ricerche, e che consiste nel minimizzare la somma dei quadrati degli errori. In questo modo si stabilisce una sorta di equilibrio tra gli errori, che impedisce agli estremi di prevalere e ben si presta a farci conoscere lo stato del sistema più vicino alla verità."*

La somma dei quadrati degli errori è

$$\sum_{i=1}^{n} e_i^2 = (y_i - a - b_1 x_{i1} - \cdots - b_m x_{im})^2.$$

Per trovare il minimo di tale funzione, Legendre pone a zero le derivate della funzione rispetto ad $a, b_1, \ldots, b_m$, il che conduce a quelle che in seguito sono state chiamate le "equazioni normali". Risolvendo il sistema di equazioni normali rispetto $a, b_1, \ldots, b_m$, si determinano le stime dei minimi quadrati dei parametri del modello di regressione.

Tutto questo è rilevante per la derivazione della Normale perché, in questo contesto, Legendre osservò che la media aritmetica, quale caso speciale dei minimi quadrati, si ottiene minimizzando $\sum (y_i - b)^2$. In precedenza, Laplace si era posto il problema di mostrare che la media aritmetica è la migliore stima possibile della tendenza centrale di una distribuzione di errori di misurazione, ma non ci era riuscito perché aveva minimizzato il valore assoluto degli scarti, il che portava ad identificare la mediana

quale migliore stimatore della tendenza centrale della distribuzione degli errori, non la media.

Nel 1809, Gauss riformulò il problema ponendosi le seguenti domande. Che forma deve avere la densità della distribuzione degli errori? Quale quantità deve essere minimizzata per fare in modo che la media aritmetica risulti la miglior stima possibile della tendenza centrale della distribuzione degli errori? *"Si è soliti considerare come un assioma l'ipotesi che se una qualsiasi grandezza è stata determinata da più osservazioni dirette, fatte nelle stesse circostanze e con uguale cura, la media aritmetica dei valori osservati dà il valore più probabile, se non rigorosamente, eppure con una grade approssimazione, così che è sempre più sicuro utilizzare tale valore."*

Basandosi sul risultato di Legendre (ovvero, che è necessario minimizzare il quadrato degli scarti dalla tendenza centrale, non il valore assoluto degli scarti), Gauss derivò la formula della densità Normale quale modello teorico della distribuzione degli errori di misurazione. La Normale ha infatti la proprietà desiderata: il valore atteso della distribuzione corrisponde alla media aritmetica.

La scoperta della distribuzione normale segna l'inizio di una nuova era nella statistica. La distribuzione Normale è importante, in primo luogo, perché molti fenomeni naturali hanno approssimativamente le caratteristiche descritte dall'esempio precedente. In secondo luogo, è importante perché molti modelli statistici assumono che il fenomeno aleatorio di interesse abbia una distribuzione Normale.

Nella derivazione della Normale, Gauss fornì una giustificazione probabilistica al metodo dei minimi quadrati basata sull'ipotesi che le osservazioni siano distribuite normalmente e che la distribuzione a priori del parametro di tendenza centrale sia uniforme. Si noti come la discussione sia formulata in termini bayesiani.

La derivazione formale della Normale è troppo complessa per gli scopi presenti. Il Paragrafo **??** illustra invece come si possa giungere alla Normale mediante una simulazione. La motivazione del presente escursus storico è stata quella di mostrare come la Normale sia fortemente legata, in un contesto storico, al modello lineare e al metodo dei minimi quadrati.

H

La stima di massima verosimiglianza

H.1 La s.m.v. per una proporzione

La s.m.v. della proporzione di successi θ in una sequenza di prove Bernoulliane è uguale data dalla proporzione di successi campionari. Questo risultato può essere dimostrato come segue.

Dimostrazione. Per n prove Bernoulliane indipendenti, le quali producono y successi e $(n-y)$ insuccessi, la funzione nucleo (ovvero, la funzione di verosimiglianza da cui sono state escluse tutte le costanti moltiplicative che non hanno alcun effetto su $\widehat{\theta}$) è

$$\mathcal{L}(p \mid y) = \theta^y (1-\theta)^{n-y}.$$

La funzione nucleo di log-verosimiglianza è

$$\begin{aligned}
\ell(\theta \mid y) &= \log \mathcal{L}(\theta \mid y) \\
&= \log \left(\theta^y (1-\theta)^{n-y}\right) \\
&= \log \theta^y + \log \left((1-\theta)^{n-y}\right) \\
&= y \log \theta + (n-y) \log(1-\theta).
\end{aligned}$$

Per calcolare il massimo della funzione di log-verosimiglianza è necessario differenziare $\ell(\theta \mid y)$ rispetto a θ, porre la derivata a zero e risolvere. La derivata di $\ell(\theta \mid y)$ è:

$$\ell'(\theta \mid y) = \frac{y}{\theta} - \frac{n-y}{1-\theta}.$$

Ponendo l'equazione uguale a zero e risolvendo otteniamo la s.m.v.:

$$\widehat{\theta} = \frac{y}{n}, \tag{H.1}$$

ovvero la frequenza relativa dei successi nel campione. □

Calcolo numerico

In maniera più semplice, il risultato descritto nel Paragrafo H.1 può essere ottenuto mediante una simulazione in R. Iniziamo a definire un insieme di valori possibili per il parametro incognito θ:

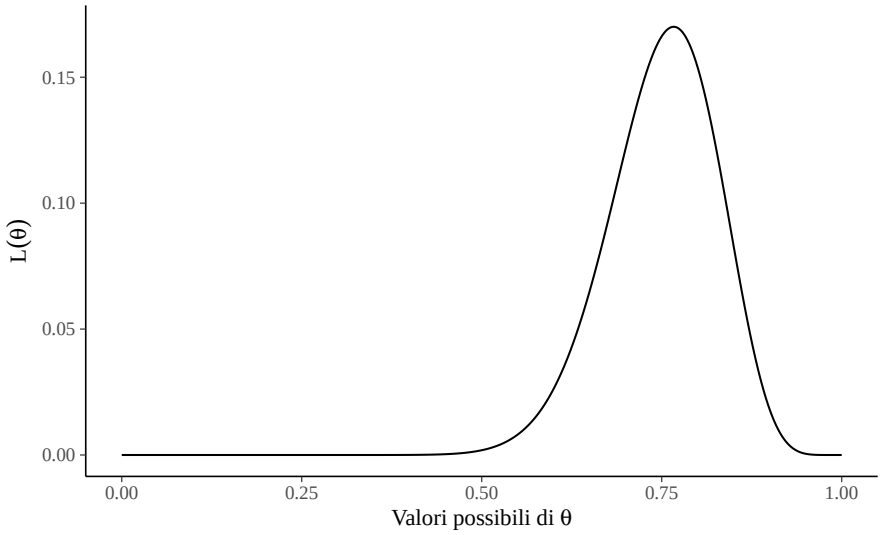
```
theta <- seq(0, 1, length.out = 1e3)
```

Sappiamo che la funzione di verosimiglianza è la funzione di massa di probabilità espressa in funzione del parametro sconosciuto θ assumendo come noti i dati. Questo si può esprimere in R nel modo seguente:

```
like <- dbinom(x = 23, size = 30, prob = theta)
```

Si noti che, nell'istruzione precedente, abbiamo passato alla funzione `dbinom()` i dati, ovvero `x = 23` successi in `size = 30` prove. Inoltre, abbiamo passato alla funzione il vettore `prob = theta` che contiene 1000 valori possibili per il parametro $\theta \in [0, 1]$. Per ciascuno dei valori θ, la funzione `dbinom()` ritorna un valore che corrisopnde all'ordinata della funzione di verosimiglianza, tenendo sempre costanti i dati (ovvero, 6 successi in 9 prove). Un grafico della funzione di verosimiglianza è dato da:

```
tibble(theta, like) %>%
  ggplot(aes(x = theta, y = like)) +
  geom_line() +
  labs(
    y = expression(L(theta)),
    x = expression('Valori possibili di' ~ theta)
  )
```



Nella simulazione, il valore θ che massimizza la funzione di verosimiglianza può essere trovato nel modo seguente:

```
theta[which.max(like)]
#> [1] 0.7668
```

Il valore così trovato è uguale al valore definito dalla (H.1).

H.2 La s.m.v. del modello Normale

Ora che abbiamo capito come costruire la funzione verosimiglianza di una binomiale è relativamente semplice fare un passo ulteriore e considerare la verosimiglianza del caso di una funzione di densità, ovvero nel caso di una variabile casuale continua. Consideriamo qui il caso della Normale.

Dimostrazione. La densità di una distribuzione Normale di parametri μ e σ è

$$f(y \mid \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{-\frac{1}{2\sigma^2}(y-\mu)^2\right\}.$$

Poniamoci il problema di trovare la s.m.v. dei parametri sconosciuti μ e σ nel caso in cui le n osservazioni $y = (y_1, \dots, y_n)$ sono realizzazioni indipendenti ed identicamente distribuite (di seguito, i.i.d.) della medesima variabile casuale $Y \sim \mathcal{N}(\mu, \sigma)$. Per semplicità, scriveremo $\theta = \{\mu, \sigma\}$.

Il campione osservato è un insieme di eventi, ciascuno dei quali corrisponde alla realizzazione di una variabile casuale — possiamo pensare ad uno di tali eventi come all'estrazione casuale di un valore dalla "popolazione" $\mathcal{N}(\mu, \sigma)$. Se le variabili casuali sono i.i.d., la loro densità congiunta è data da:

$$\begin{aligned} f(y \mid \theta) &= f(y_1 \mid \theta) \cdot f(y_2 \mid \theta) \cdot \ \dots \ \cdot f(y_n \mid \theta) \\ &= \prod_{i=1}^{n} f(y_i \mid \theta), \end{aligned} \tag{H.2}$$

laddove la funzione $f(\cdot)$ è la (H.2). Tenendo costanti i dati y, la funzione di verosimiglianza è:

$$\mathcal{L}(\theta \mid y) = \prod_{i=1}^{n} f(y_i \mid \theta). \tag{H.3}$$

L'obiettivo è quello di massimizzare la funzione di verosimiglianza per trovare i valori θ ottimali. Usando la notazione matematica questo si esprime dicendo che cerchiamo l'argmax della (H.3) rispetto a θ, ovvero

$$\hat{\theta} = \text{argmax}_\theta \prod_{i=1}^{n} f(y_i \mid \theta).$$

Questo problema si risolve calcolando le derivate della funzione rispetto a θ, ponendo le derivate uguali a zero e risolvendo. Saltando tutti i passaggi algebrici di questo procedimento, per μ troviamo

$$\hat{\mu} = \frac{1}{n} \sum_{i=1}^{n} y_i \tag{H.4}$$

e per σ abbiamo

$$\hat{\sigma} = \sqrt{\sum_{i=1}^{n} \frac{1}{n} (y_i - \mu)^2}. \tag{H.5}$$

In altri termini, la s.m.v. del parametro μ è la media del campione e la s.m.v. del parametro σ è la deviazione standard del campione. □

Calcolo numerico

Consideriamo ora un esempio che utilizza dei dati reali. I dati corrispondono ai valori BDI-II dei trenta soggetti del campione clinico di Zetsche et al. (2019):

```
d <- tibble(
  y = c(
    26, 35, 30, 25, 44, 30, 33, 43, 22, 43, 24, 19, 39, 31, 25, 28, 35, 30,
    26, 31, 41, 36, 26, 35, 33, 28, 27, 34, 27, 22)
)
```

Ci poniamo l'obiettivo di creare la funzione di verosimiglianza per questi dati, supponendo, in base ai risultati di ricerche precedenti, di sapere che i punteggi BDI-II si distribuiscono secondo una legge Normale.

Per semplificare il problema, assumeremo di conoscere σ (lo porremo uguale alla deviazione standard del campione) in modo da avere un solo parametro sconosciuto, cioè μ. Il problema è dunque quello di trovare la funzione di verosimiglianza per il parametro μ, date le 30 osservazioni del campione e dato $\sigma = s = 6.61$.

Per una singola osservazione, la funzione di verosimiglianza è la densità Normale espressa in funzione dei parametri. Per un campione di osservazioni i.i.d., ovvero $y = (y_1, y_2, \dots, y_n)$, la verosimiglianza è la funzione di densità congiunta $f(y \mid \mu, \sigma)$ espressa in funzione dei parametri, ovvero $\mathcal{L}(\mu, \sigma \mid y)$. Dato che le osservazioni sono i.i.d., la densità congiunta è data dal prodotto delle densità delle singole osservazioni. Per semplicità, assumiamo σ noto e uguale alla deviazione standard del campione:

```
true_sigma <- sd(d$y)
true_sigma
#> [1] 6.607
```

Avendo posto $\sigma = 6.61$, per una singola osservazione y_i abbiamo

$$f(y_i \mid \mu, \sigma) = \frac{1}{6.61\sqrt{2\pi}} \exp\left\{-\frac{(y_i - \mu)^2}{2 \cdot 6.61^2}\right\},$$

dove il pedice i specifica l'osservazione y_i tra le molteplici osservazioni y, e μ è il parametro sconosciuto che deve essere determinato (nell'esempio, $\sigma = s$). La densità congiunta è dunque

$$f(y \mid \mu, \sigma) = \prod_{i=1}^{n} f(y_i \mid \mu, \sigma)$$

e, alla luce dei dati osservati, la verosimiglianza diventa

$$\begin{aligned}\mathcal{L}(\mu, \sigma \mid y) = \prod_{i=1}^{n} f(y_i \mid \mu, \sigma) = \\ &\frac{1}{6.61\sqrt{2\pi}} \exp\left\{-\frac{(26 - \mu)^2}{2 \cdot 6.61^2}\right\} \times \\ &\frac{1}{6.61\sqrt{2\pi}} \exp\left\{-\frac{(35 - \mu)^2}{2 \cdot 6.61^2}\right\} \times \\ &\vdots \\ &\frac{1}{6.61\sqrt{2\pi}} \exp\left\{-\frac{(22 - \mu)^2}{2 \cdot 6.61^2}\right\}.\end{aligned}$$

Poniamoci ora il problema di rappresentare graficamente la funzione di verosimiglianza per il parametro μ. Avendo un solo parametro sconosciuto, possiamo rappresentare la verosimiglianza con una curva. In R, definiamo la funzione di log-verosimiglianza nel modo seguente:

```
log_likelihood <- function(y, mu, sigma = true_sigma) {
  sum(dnorm(y, mu, sigma, log = TRUE))
}
```

Nella funzione `log_likelihood()`, `y` è un vettore che, nel caso presente contiene $n = 30$ valori. Per ciascuno di questi valori, la funzione `dnorm()` trova la densità Normale utilizzando il valore μ che passato a `log_likelihood()` e il valore σ uguale a 6.61 — nell'esempio, questo parametro viene assunto come noto. L'argomento `log = TRUE` specifica che deve essere preso il logaritmo. La funzione `dnorm()` è un argomento della

funzione `sum()`. Ciò significa che i 30 valori così trovati, espressi su scala logaritmica, verranno sommati — sommare logaritmi è equivalente a fare il prodotto dei valori sulla scala originaria.

Se applichiamo questa funzione ad un solo valore μ otteniamo l'ordinata della funzione di log-verosimiglianza in corrispondenza del valore μ (si veda la figura (H.2)). Si noti che, per trovare un tale valore, abbiamo utilizzato le seguenti informazioni:

- i 30 dati del campione,
- il valore $\sigma = s$ fissato a 6.61,
- il singolo valore μ passato alla funzione `log_likelihood()`.

Avendo trovato un singolo punto della funzione di log-verosimiglianza, dobbiamo ripetere i calcoli precedenti per tutti i possibili valori che μ può assumere. Nel seguente ciclo `for()` viene calcolata la log-verosimiglianza di 100,000 valori possibili del parametro μ:

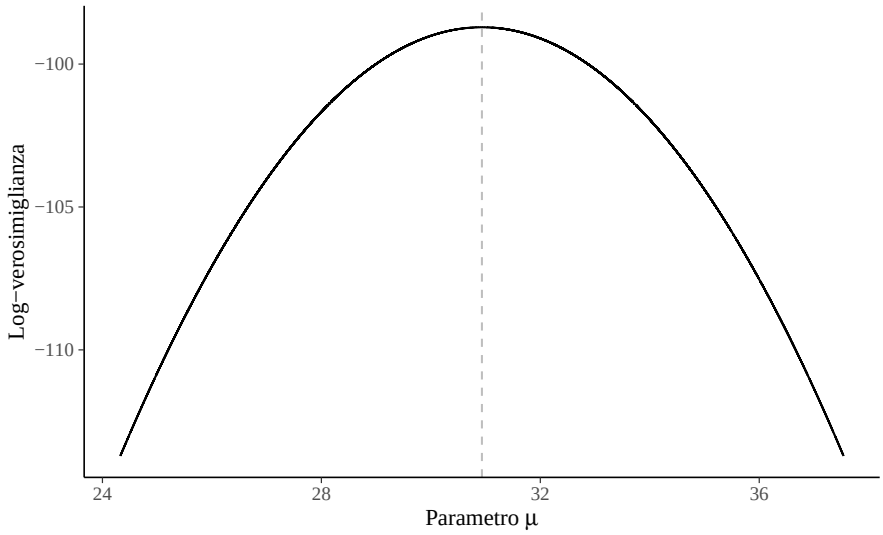
```
nrep <- 1e5
mu <- seq(
  mean(d$y) - sd(d$y),
  mean(d$y) + sd(d$y),
  length.out = nrep
)

ll <- rep(NA, nrep)
for (i in 1:nrep) {
  ll[i] <- log_likelihood(d$y, mu[i], true_sigma)
}
```

Il vettore `mu` contiene 100,000 possibili valori del parametro μ; tali valori sono stati scelti nell'intervallo $\bar{y} \pm s$. Per ciascuno di questi valori la funzione `log_likelihood()` calcola il valore di log-verosimiglianza. I 100,000 risultati vengono salvati nel vettore `ll`.

I vettori `mu` e `ll` possono dunque essere usati per disegnare il grafico della funzione di log-verosimiglianza per il parametro μ:

```
tibble(mu, ll) %>%
ggplot(aes(x = mu, y = ll)) +
  geom_line() +
```

```
  vline_at(mean(d$y), color = "gray", linetype = "dashed") +
  labs(
    y = "Log-verosimiglianza",
    x = expression("Parametro"~mu)
  )
```



Dalla figura notiamo che, per i dati osservati, il massimo della funzione di log-verosimiglianza calcolata per via numerica, ovvero 30.93, è identico alla media dei dati campionari e corrisponde al risultato teorico della (H.4).

Considerazioni conclusive

La verosimiglianza viene utilizzata sia nell'inferenza bayesiana che in quella frequentista. In entrambi i paradigmi di inferenza, il suo ruolo è quantificare la forza con la quale i dati osservati supportano i possibili valori dei parametri sconosciuti.

Nella funzione di verosimiglianza i dati (osservati) vengono trattati come fissi, mentre i valori del parametro (o dei parametri) θ vengono variati: la verosimiglianza è una funzione di θ per il dato fisso y. Pertanto, la funzio-

ne di verosimiglianza riassume i seguenti elementi: un modello statistico che genera stocasticamente i dati (in questo capitolo abbiamo esaminato due modelli statistici: quello binomiale e quello Normale), un intervallo di valori possibili per θ e i dati osservati y.

Nella statistica frequentista l'inferenza si basa solo sui dati a disposizione e qualunque informazione fornita dalle conoscenze precedenti non viene presa in considerazione. Nello specifico, nella statistica frequentista l'inferenza viene condotta massimizzando la funzione di (log) verosimiglianza, condizionatamente ai valori assunti dalle variabili casuali campionarie. Nella statistica bayesiana, invece, l'inferenza statistica viene condotta combinando la funzione di verosimiglianza con le distribuzioni a priori dei parametri incogniti θ.

La differenza fondamentale tra inferenza bayesiana e frequentista è dunque che i frequentisti non ritengono utile descrivere in termini probabilistici i parametri: i parametri dei modelli statistici vengono concepiti come fissi ma sconosciuti. Nell'inferenza bayesiana, invece, i parametri sconosciuti sono intesi come delle variabili casuali e ciò consente di quantificare in termini probabilistici il nostro grado di intertezza relativamente al loro valore.

I

Verosimiglianza marginale

I.1 Derivazione analitica della costante di normalizzazione

Riportiamo di seguito la derivazione analitica per la costante di normalizzazione discussa nella Sezione **??**, ovvero dell'integrale (**??**).

Dimostrazione. Sia la distribuzione a priori $\theta \sim \text{Beta}(a, b)$ e sia $y = \{y_1, \dots, y_n\} \sim \text{Bin}(\theta, n)$. Scrivendo la *funzione beta* come

$$\text{B}(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a + b)},$$

la verosimiglianza marginale diventa

$$\begin{aligned} p(y) &= \int p(y \mid \theta) p(\theta) \, \mathrm{d}\theta \\ &= \int_0^1 \binom{n}{y} \theta^y (1-\theta)^{n-y} \frac{1}{\text{B}(a, b)} \theta^{a-1} (1-\theta)^{b-1} \, \mathrm{d}\theta \\ &= \binom{n}{y} \frac{1}{\text{B}(a, b)} \int_0^1 \theta^{y+a-1} (1-\theta)^{n-y+b-1} \, \mathrm{d}\theta \\ &= \binom{n}{y} \frac{\text{B}(y + a, n - y + b)}{\text{B}(a, b)}, \end{aligned} \tag{I.1}$$

in quanto

$$\int_0^1 \frac{1}{\mathrm{B}(a,b)} \theta^{a-1}(1-\theta)^{b-1}\, \mathrm{d}\theta = 1$$
$$\frac{1}{\mathrm{B}(a,b)} \int_0^1 \theta^{a-1}(1-\theta)^{b-1}\, \mathrm{d}\theta = 1$$
$$\int_0^1 \theta^{a-1}(1-\theta)^{b-1}\, \mathrm{d}\theta = \mathrm{B}(a,b).$$

In conclusione, nel caso di una verosimiglianza binomiale $y =\sim \mathrm{Bin}(\theta, n)$ e di una distribuzione a priori $\theta \sim \mathrm{Beta}(a,b)$, la verosimiglianza marginale diventa uguale alla (I.1). □

Esercizio I.1. Si verifichi la (I.1) mediante di dati di Zetsche et al. (2019).

Per replicare mediante la (I.1) il risultato trovato per via numerica nella Sezione **??** assumiamo una distribuzione a priori uniforme, ovvero $\mathrm{Beta}(1,1)$. I valori del problema dunque diventano i seguenti:

```
a <- 1
b <- 1
y <- 23
n <- 30
```

Definiamo

```
B <- function(a, b) {
  (gamma(a) * gamma(b)) / gamma(a + b)
}
```

Il risultato cercato è

```
choose(30, 23) * B(y + a, n - y + b) / B(a, b)
#> [1] 0.03226
```

J

Aspettative degli individui depressi

Per fare pratica, applichiamo il metodo basato su griglia ad un campione di dati reali. Zetsche et al. (2019) si sono chiesti se gli individui depressi manifestino delle aspettative accurate circa il loro umore futuro, oppure se tali aspettative siano distorte negativamente. Esamineremo qui i 30 partecipanti dello studio di Zetsche et al. (2019) che hanno riportato la presenza di un episodio di depressione maggiore in atto. All'inizio della settimana di test, a questi pazienti è stato chiesto di valutare l'umore che si aspettavano di esperire nei giorni seguenti della settimana. Mediante una app, i partecipanti dovevano poi valutare il proprio umore in cinque momenti diversi di ciascuno dei cinque giorni successivi. Lo studio considera diverse emozioni, ma qui ci concentriamo solo sulla tristezza.

Sulla base dei dati forniti dagli autori, abbiamo calcolato la media dei giudizi relativi al livello di tristezza raccolti da ciascun partecipante tramite la app. Tale media è stata poi sottratta dall'aspettativa del livello di tristezza fornita all'inizio della settimana. La discrepanza tra aspettative e realtà è stata considerata come un evento dicotomico: valori positivi di tale differenza indicano che le aspettative circa il livello di tristezza erano maggiori del livello di tristezza effettivamente esperito — ciò significa che le aspettative future risultano negativamente distorte (evento codificato con "1"). Viceversa, si ha che le aspettative risultano positivamente distorte se la differenza descritta in precedenza assume un valore negativo (evento codificato con "0").

Nel campione dei 30 partecipanti clinici di Zetsche et al. (2019), le aspettative future di 23 partecipanti risultano distorte negativamente e quelle di 7 partecipanti risultano distorte positivamente. Chiameremo θ la probabilità dell'evento "le aspettative del partecipante sono distorte negativamente". Ci poniamo il problema di ottenere una stima a posteriori di θ usando il metodo basato su griglia.

J.1 La griglia

Fissiamo una griglia di $n = 50$ valori equispaziati nell'intervallo [0, 1] per il parametro θ:

```
n_points <- 50
p_grid <- seq(from = 0, to = 1, length.out = n_points)
p_grid
#>  [1] 0.00000 0.02041 0.04082 0.06122 0.08163 0.10204
#>  [7] 0.12245 0.14286 0.16327 0.18367 0.20408 0.22449
#> [13] 0.24490 0.26531 0.28571 0.30612 0.32653 0.34694
#> [19] 0.36735 0.38776 0.40816 0.42857 0.44898 0.46939
#> [25] 0.48980 0.51020 0.53061 0.55102 0.57143 0.59184
#> [31] 0.61224 0.63265 0.65306 0.67347 0.69388 0.71429
#> [37] 0.73469 0.75510 0.77551 0.79592 0.81633 0.83673
#> [43] 0.85714 0.87755 0.89796 0.91837 0.93878 0.95918
#> [49] 0.97959 1.00000
```

J.2 Distribuzione a priori

Supponiamo di avere scarse credenze a priori sulla tendenza di un individuo clinicamente depresso a manifestare delle aspettative distorte negativamente circa il suo umore futuro. Imponiamo quindi una distribuzione non informativa sulla distribuzione a priori di θ — ovvero, una distribuzione uniforme nell'intervallo [0, 1]. Dato che consideriamo soltanto $n = 50$ valori possibili per il parametro θ, creiamo un vettore di 50 elementi che conterrà i valori della distribuzione a priori scalando ciascun valore del vettore per n in modo tale che la somma di tutti i valori sia uguale a 1.0:

```
prior1 <- dbeta(p_grid, 1, 1) / sum(dbeta(p_grid, 1, 1))
prior1
#>  [1] 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
#> [11] 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
```

```
#> [21] 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
#> [31] 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
#> [41] 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02 0.02
```

Verifichiamo:

```
sum(prior1)
#> [1] 1
```

La distribuzione a priori così costruita è rappresentata nella figura J.1.

```
p1 <- data.frame(p_grid, prior1) %>%
  ggplot(aes(x=p_grid, xend=p_grid, y=0, yend=prior1)) +
  geom_line() +
  geom_segment() +
  ylim(0, 0.17) +
  labs(
    x = "Parametro \U03B8",
    y = "Probabilità a priori",
    title = "50 punti"
  )
p1
```

J.3 Funzione di verosimiglianza

Calcoliamo ora la funzione di verosimiglianza utilizzando i 50 valori θ definiti in precedenza. A ciascuno dei valori della griglia applichiamo la formula binomiale, tendendo costanti i dati (ovvero 23 "successi" in 30 prove). Ad esempio, in corrispondenza del valore $\theta = 0.816$, l'ordinata della funzione di verosimiglianza diventa

$$\binom{30}{23} \cdot 0.816^{23} \cdot (1 - 0.816)^7 = 0.135.$$

Per $\theta = 0.837$, l'ordinata della funzione di verosimiglianza sarà

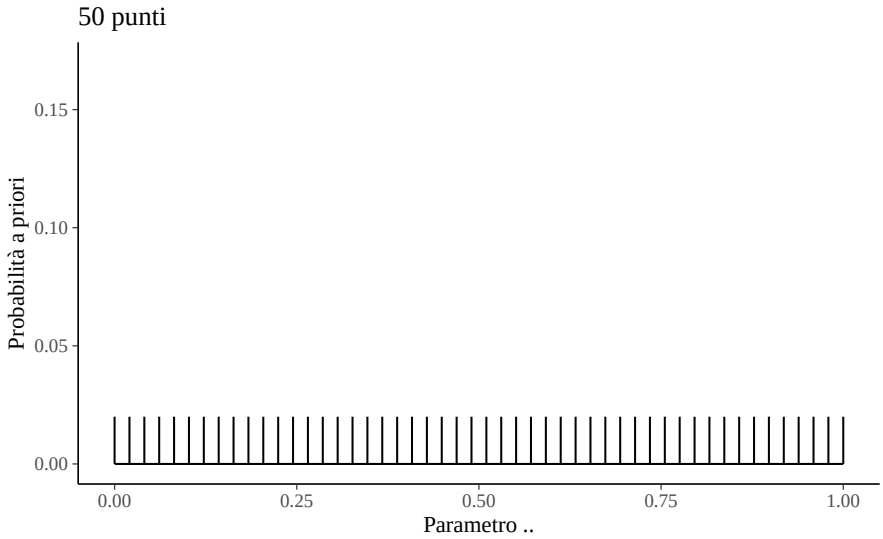


Figura J.1: *Rappresentazione grafica della distribuzione a priori per il parametro heta, ovvero la probabilità di aspettative future distorte negativamente.*

$$\binom{30}{23} \cdot 0.837^{23} \cdot (1 - 0.837)^{7} = 0.104.$$

Dobbiamo svolgere questo calcolo per tutti gli elementi della griglia. Usando R, tale risultato si trova nel modo seguente:

```
likelihood <- dbinom(x = 23, size = 30, prob = p_grid)
likelihood
#>  [1] 0.000e+00 2.353e-33 1.703e-26 1.644e-22 1.054e-19
#>  [6] 1.525e-17 8.602e-16 2.528e-14 4.607e-13 5.819e-12
#> [11] 5.499e-11 4.106e-10 2.520e-09 1.311e-08 5.919e-08
#> [16] 2.362e-07 8.457e-07 2.749e-06 8.197e-06 2.260e-05
#> [21] 5.799e-05 1.393e-04 3.149e-04 6.721e-04 1.359e-03
#> [26] 2.612e-03 4.779e-03 8.340e-03 1.390e-02 2.214e-02
#> [31] 3.372e-02 4.910e-02 6.830e-02 9.068e-02 1.147e-01
#> [36] 1.378e-01 1.568e-01 1.682e-01 1.689e-01 1.575e-01
#> [41] 1.349e-01 1.044e-01 7.133e-02 4.166e-02 1.973e-02
#> [46] 6.937e-03 1.535e-03 1.473e-04 1.868e-06 0.000e+00
```

La funzione `dbinom(x, size, prob)` richiede che vengano specificati tre parametri: il numero di "successi", il numero di prove e la probabilità di successo. Nella chiamata precedente, `x` (numero di successi) e `size` (numero di prove bernoulliane) sono degli scalari e `prob` è il vettore `p_grid`. In tali circostanze, l'output di `dbinom()` è il vettore che abbiamo chiamato `likelihood`. Gli elementi di tale vettore sono stati calcolati applicando la formula della distribuzione binomiale a ciascuno dei 50 elementi della griglia, tenendo sempre costanti i dati [ovvero, `x` (il numero di successi) e `size` (numero di prove bernoulliane)]; ciò che varia è il valore `prob`, che assume valori diversi (`p_grid`) in ciascuna cella della griglia.

La chiamata a `dbinom()` produce dunque un vettore i cui valori corrispondono all'ordinata della funzione di verosimiglianza per per ciascun valore θ specificato in `p_grid`. La verosimiglianza discretizzata così ottenuta è riportata nella figura J.2.

```
p2 <- data.frame(p_grid, likelihood) %>%
  ggplot(aes(x=p_grid, xend=p_grid, y=0, yend=likelihood)) +
  geom_segment() +
  ylim(0, 0.17) +
  labs(
    x = "Parametro \U03B8",
    y = "Verosimiglianza"
  )
p2
```

J.4 Distribuzione a posteriori

L'approssimazione discretizzata della distribuzione a posteriori $p(\theta \mid y)$ si ottiene facendo il prodotto della verosimiglianza e della distribuzione a priori per poi scalare tale prodotto per una costante di normalizzazione. Il prodotto $p(\theta)\mathcal{L}(y \mid \theta)$ produce la distribuzione a posteriori *non standardizzata*.

Nel caso di una distribuzione a priori non informativa (ovvero una distribuzione uniforme), per ottenere la funzione a posteriori non standardizzata è sufficiente moltiplicare ciascun valore della funzione di vero-

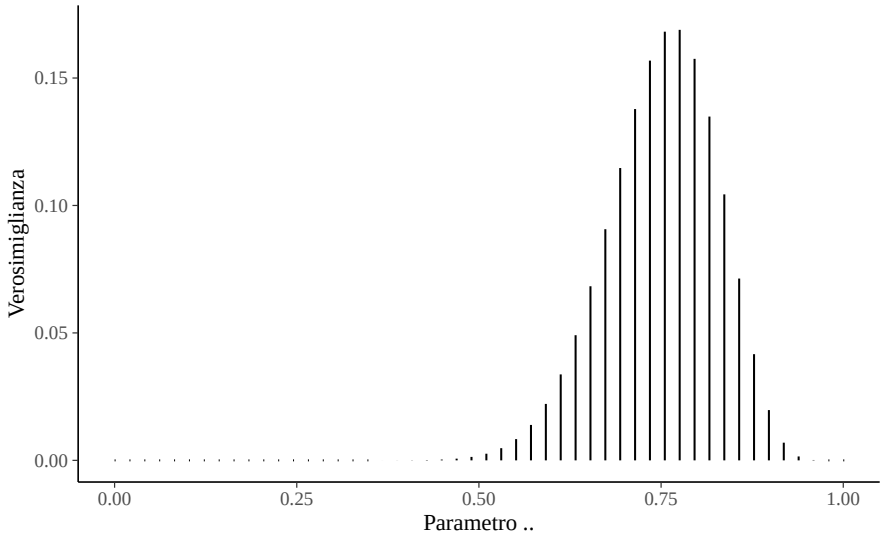


Figura J.2: *Rappresentazione della funzione di verosimiglianza per il parametro θ, ovvero la probabilità di aspettative future distorte negativamente.*

simiglianza per 0.02. Per esempio, per il primo valore della funzione di verosimiglianza usato quale esempio poco sopra, abbiamo $0.135 \cdot 0.02$; per il secondo valore dell'esempio abbiamo $0.104 \cdot 0.02$; e così via. Possiamo svolgere tutti i calcoli usando R nel modo seguente:[1]

```
unstd_posterior <- likelihood * prior1
unstd_posterior
#>  [1] 0.000e+00 4.705e-35 3.406e-28 3.288e-24 2.107e-21
#>  [6] 3.050e-19 1.720e-17 5.057e-16 9.214e-15 1.164e-13
#> [11] 1.100e-12 8.211e-12 5.040e-11 2.622e-10 1.184e-09
#> [16] 4.724e-09 1.691e-08 5.499e-08 1.639e-07 4.519e-07
#> [21] 1.160e-06 2.786e-06 6.297e-06 1.344e-05 2.718e-05
#> [26] 5.224e-05 9.558e-05 1.668e-04 2.780e-04 4.428e-04
#> [31] 6.744e-04 9.820e-04 1.366e-03 1.814e-03 2.294e-03
#> [36] 2.756e-03 3.136e-03 3.363e-03 3.378e-03 3.150e-03
```

[1] Ricordiamo il principio dell'aritmetica vettorializzata: i vettori `likelihood` e `prior1` sono entrambi costituiti da 50 elementi. Se facciamo il prodotto tra i due vettori otteniamo un vettore di 50 elementi, ciascuno dei quali uguale al prodotto dei corrispondenti elementi dei vettori `likelihood` e `prior1`.

```
#> [41] 2.697e-03 2.087e-03 1.427e-03 8.331e-04 3.945e-04
#> [46] 1.387e-04 3.070e-05 2.947e-06 3.736e-08 0.000e+00
```

Avendo calcolato i valori della funzione a posteriori non standardizzata è poi necessario dividere per una costante di normalizzazione. Nel caso discreto, trovare il denominatore del teorema di Bayes è facile: esso è uguale alla somma di tutti i valori della distribuzione a posteriori non normalizzata. Per i dati presenti, tale costante di normalizzazione è uguale a 0.032:

```
sum(unstd_posterior)
#> [1] 0.03161
```

La standardizzazione dei due valori usati come esempio è data da: $0.135 \cdot 0.02/0.032$ e da $0.104 \cdot 0.02/0.032$. Usiamo R per svolgere questo calcolo su tutti i 50 valori di `unstd_posterior` così che la somma dei 50 i valori di `posterior` sia uguale a 1.0:

```
posterior <- unstd_posterior / sum(unstd_posterior)
posterior
#>  [1] 0.000e+00 1.488e-33 1.077e-26 1.040e-22 6.666e-20
#>  [6] 9.649e-18 5.442e-16 1.600e-14 2.915e-13 3.681e-12
#> [11] 3.479e-11 2.597e-10 1.594e-09 8.295e-09 3.745e-08
#> [16] 1.494e-07 5.350e-07 1.739e-06 5.186e-06 1.430e-05
#> [21] 3.669e-05 8.814e-05 1.992e-04 4.252e-04 8.599e-04
#> [26] 1.652e-03 3.023e-03 5.276e-03 8.794e-03 1.401e-02
#> [31] 2.133e-02 3.106e-02 4.321e-02 5.737e-02 7.256e-02
#> [36] 8.719e-02 9.922e-02 1.064e-01 1.069e-01 9.966e-02
#> [41] 8.533e-02 6.602e-02 4.513e-02 2.635e-02 1.248e-02
#> [46] 4.389e-03 9.712e-04 9.321e-05 1.182e-06 0.000e+00
```

Verifichiamo:

```
sum(posterior)
#> [1] 1
```

La distribuzione a posteriori così trovata non è altro che la versione normalizzata della funzione di verosimiglianza: questo avviene perché

la distribuzione a priori uniforme non ha aggiunto altre informazioni oltre a quelle che erano già fornite dalla funzione di verosimiglianza. L'approssimazione discretizzata di $p(\theta \mid y)$ che abbiamo appena trovato è riportata nella figura J.3.

```
p3 <- data.frame(p_grid, posterior) %>%
  ggplot(aes(x=p_grid, xend=p_grid, y=0, yend=posterior)) +
  geom_segment() +
  ylim(0, 0.17) +
  labs(
    x = "Parametro \U03B8",
    y = "Probabilità a posteriori"
  )
p3
```

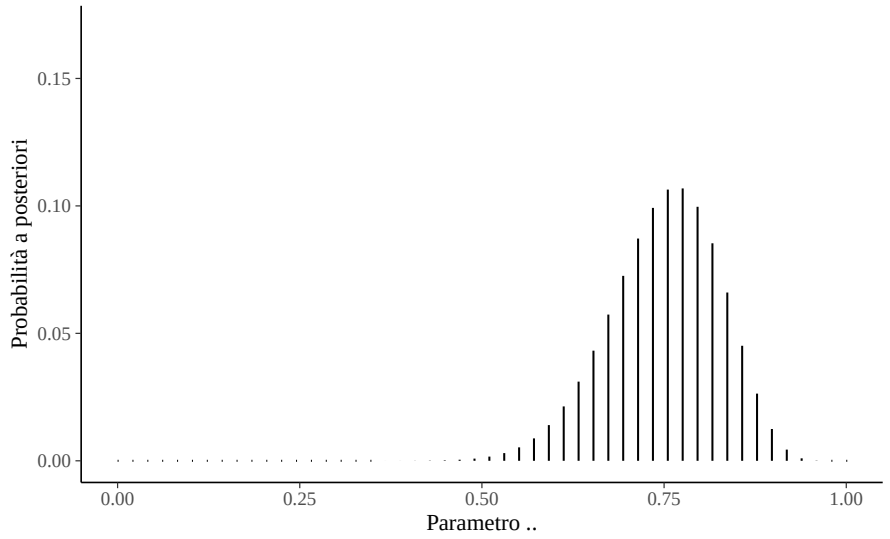


Figura J.3: *Rappresentazione della distribuzione a posteriori per il parametro θ, ovvero la probabilità di aspettative future distorte negativamente.*

I grafici delle figure J.1, J.2 e J.3 sono state calcolati utilizzando una griglia di 50 valori equi-spaziati per il parametro θ. I segmenti verticali rappresentano l'intensità della funzione in corrispondenza di ciascuna modalità parametro θ. Nella figura J.1 e nella figura J.3 la somma delle lunghezze dei segmenti verticali è uguale a 1.0; ciò non si verifica, invece,

nel caso della figura J.3 (la funzione di verosimiglianza non è mai una funzione di probabilità, né nel caso discreto né in quello continuo).

J.5 La stima della distribuzione a posteriori (versione 2)

Continuiamo l'analisi di questi dati esaminiamo l'impatto di una distribuzione a priori informativa sulla distribuzione a posteriori. Una distribuzione a priori informativa riflette un alto grado di certezza a priori sui valori dei parametri del modello. Un ricercatore utilizza una distribuzione a priori informativa per introdurre nel processo di stima informazioni pre-esistenti alla raccolta dei dati, introducendo così delle restrizioni sulla possibile gamma di valori del parametro.

Nel caso presente, supponiamo che la letteratura psicologica fornisca delle informazioni su θ (la probabilità che le aspettative future di un individuo clinicamente depresso siano distorte negativamente). Per fare un esempio, supponiamo (irrealisticamente) che tali conoscenze pregresse possano essere rappresentate da una Beta di parametri $\alpha = 2$ e $\beta = 10$. Tali ipotetiche conoscenze pregresse ritengono molto plausibili valori θ bassi e considerano implausibili valori $\theta > 0.5$. Questo è equivalente a dire che ci aspettiamo che le aspettative relative all'umore futuro siano distorte negativamente solo per pochissimi individui clinicamente depressi — ovvero, ci aspettiamo che la maggioranza degli individui clinicamente depressi sia inguaribilmente ottimista. Questa è, ovviamente, una credenza a priori del tutto irrealistica. La esamino qui, non perché abbia alcun senso nel contesto dei dati di Zetsche et al. (2019), ma soltanto per fare un esempio nel quale risulta chiaro come la distribuzione a posteriori sia una sorta di "compromesso" tra la distribuzione a priori e la verosimiglianza.

Con calcoli del tutto simili a quelli descritti sopra si giunge alla distribuzione a posteriori rappresentata nella figura J.4. Useremo ora una griglia di 100 valori per il parametro θ:

```
n_points <- 100
p_grid <- seq(from = 0, to = 1, length.out = n_points)
```

Per la distribuzione a priori scegliamo una Beta(2, 10):

```
alpha <- 2
beta <- 10
prior2 <- dbeta(p_grid, alpha, beta) / sum(dbeta(p_grid, alpha, beta))
sum(prior2)
#> [1] 1
```

Tale distribuzione a priori è rappresentata nella figura J.4:

```
plot_df <- data.frame(p_grid, prior2)
p4 <- plot_df %>%
  ggplot(aes(x=p_grid, xend=p_grid, y=0, yend=prior2)) +
  geom_segment() +
  ylim(0, 0.17) +
  labs(
    x = "",
    y = "Probabilità a priori"
  )
p4
```

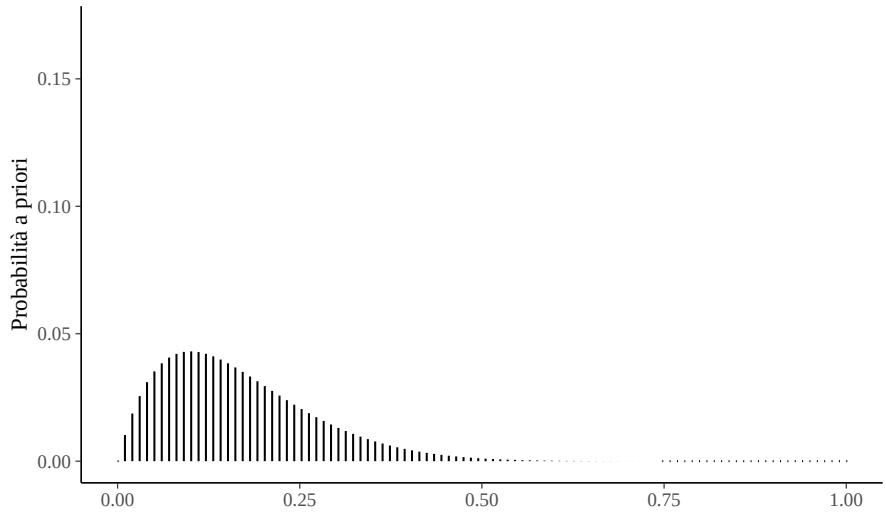


Figura J.4: *Rappresentazione di una funzione a priori informativa per il parametro θ.*

Calcoliamo il valore di verosimiglianza per ciascun punto della griglia:

```
likelihood <- dbinom(23, size = 30, prob = p_grid)
```

Per ciascun punto della griglia, il prodotto tra la verosimiglianza e distribuzione a priori è dato da:

```
unstd_posterior2 <- likelihood * prior2
```

È necessario normalizzare la distribuzione a posteriori discretizzata:

```
posterior2 <- unstd_posterior2 / sum(unstd_posterior2)
```

Verifichiamo:

```
sum(posterior2)
#> [1] 1
```

La nuova funzione a posteriori discretizzata è rappresentata nella figura J.5:

```
plot_df <- data.frame(p_grid, posterior2)
p5 <- plot_df %>%
  ggplot(aes(x = p_grid, xend = p_grid, y = 0, yend = posterior2)) +
  geom_segment() +
  ylim(0, 0.17) +
  labs(
    x = "Parametro \U03B8",
    y = "Probabilità a posteriori"
  )
p5
```

Facendo un confronto tra le figure J.4 e J.5 notiamo una notevole differenza tra la distribuzione a priori e la distribuzione a posteriori. In particolare, la distribuzione a posteriori risulta spostata verso destra su posizioni più vicine a quelle della verosimiglianza [figura J.2]. Si noti inoltre che, a causa dell'effetto della distribuzione a priori, le distribuzioni a posteriori delle figure J.3 e J.5 sono molto diverse tra loro.

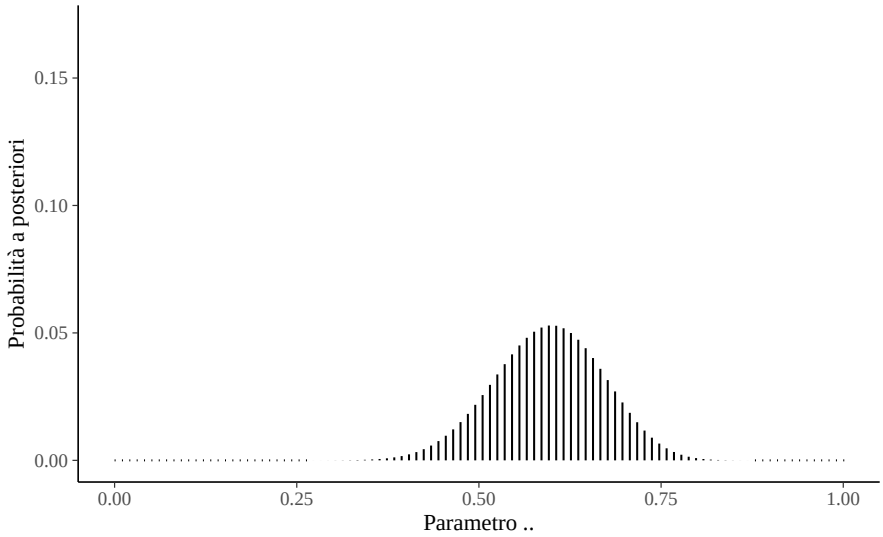


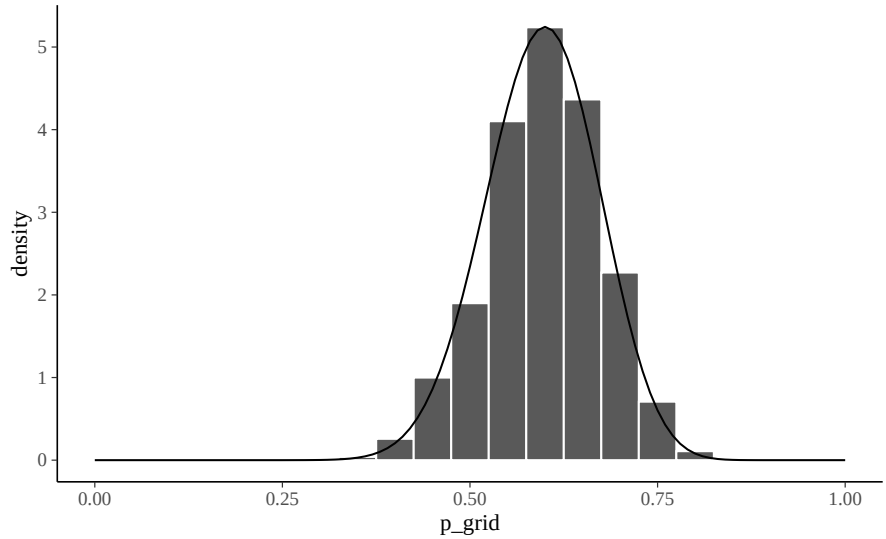
Figura J.5: *Rappresentazione della funzione a posteriori per il parametro θ calcolata utilizzando una distribuzione a priori informativa.*

Campioniamo ora 10,000 punti dall'approssimazione discretizzata della distribuzione a posteriori:

```
# Set the seed
set.seed(84735)

df <- data.frame(
  p_grid,
  posterior2
)
# Step 4: sample from the discretized posterior
post_samples <- df %>%
  slice_sample(
  n = 1e5,
  weight_by = posterior2,
  replace = TRUE
)
```

Una rappresentazione grafica del campione casuale estratto dalla distribuzione a posteriori $p(\theta \mid y)$ è data da:

```
post_samples %>%
  ggplot(aes(x = p_grid)) +
  geom_histogram(
    aes(y = ..density..),
    color = "white",
    binwidth = 0.05
  ) +
  stat_function(fun = dbeta, args = list(25, 17)) +
  lims(x = c(0, 1))
```



All'istogramma è stata sovrapposta la corretta distribuzione a posteriori, ovvero una Beta di parametri 25 ($y + \alpha = 23 + 2$) e 17 ($n - y + \beta = 30$ - 23 + 10).

La stima della moda a posteriori si ottiene con

```
df$p_grid[which.max(df$posterior2)]
#> [1] 0.596
```

e corrisponde a

$$\text{Mo} = \frac{\alpha - 1}{\alpha + \beta - 2} = \frac{25 - 1}{25 + 17 - 2} = 0.6.$$

La stima della media a posteriori si ottiene con

```
mean(post_samples$p_grid)
#> [1] 0.5953
```

e corrisponde a

$$\bar{\theta} = \frac{\alpha}{\alpha + \beta} = \frac{25}{25 + 17} \approx 0.5952.$$

La stima della mediana a posteriori si ottiene con

```
median(post_samples$p_grid)
#> [1] 0.596
```

e corrisponde a

$$\text{Me} = \frac{\alpha - \frac{1}{3}}{\alpha + \beta - \frac{2}{3}} \approx 0.5968.$$

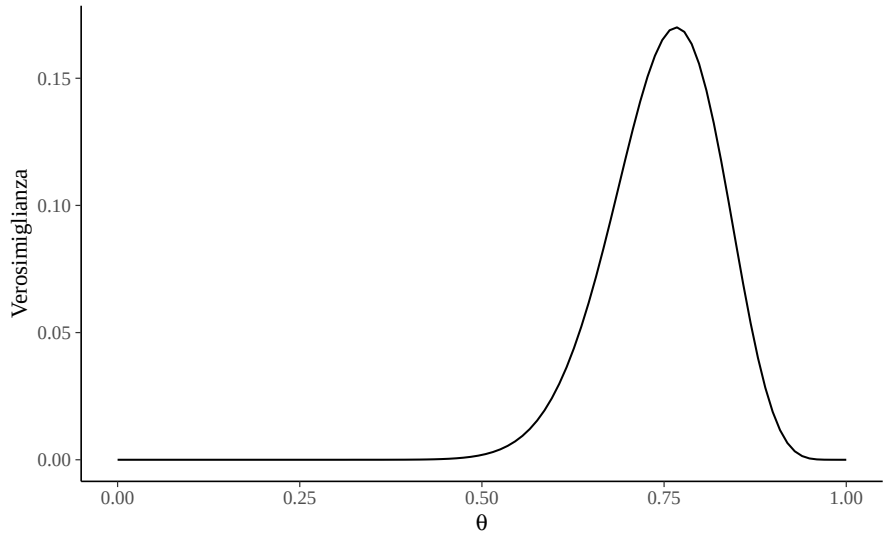
J.6 Versione 2

Possiamo semplificare i calcoli precedenti definendo le funzioni `likelihood()`, `prior()` e `posterior()`.

Per calcolare la funzione di verosimiglianza per i 30 valori di Zetsche et al. (2019) useremo la funzione `likelihood()`:

```
x <- 23
N <- 30
param <- seq(0, 1, length.out = 100)

likelihood <- function(param, x = 23, N = 30) {
  dbinom(x, N, param)
}

tibble(
```

```
  x = param,
  y = likelihood(param)
) %>%
  ggplot(aes(x, y)) +
  geom_line() +
  labs(
    x = expression(theta),
    y = "Verosimiglianza"
  )
```

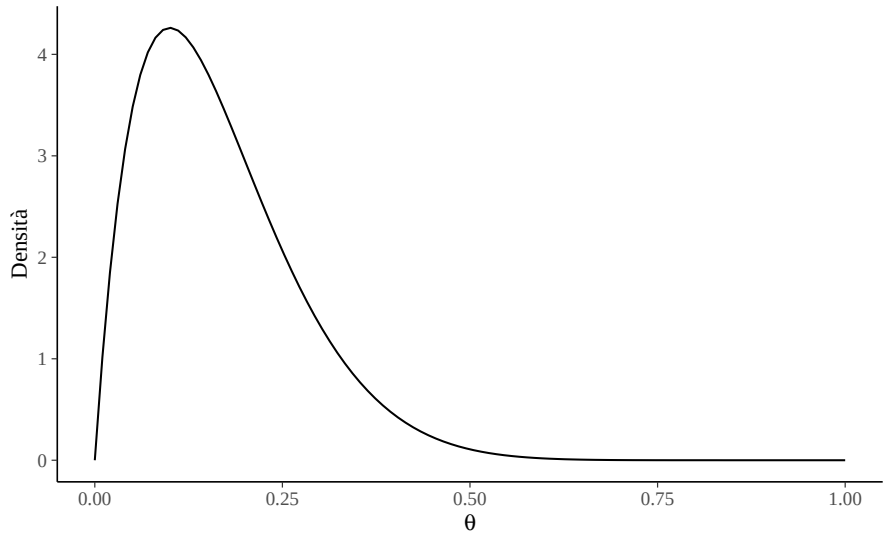


La funzione `likelihood()` ritorna l'ordinata della verosimiglianza binomiale per ciascun valore del vettore `param` in input.

Quale distribuzione a priori utilizzeremo una Beta$(2, 10)$ che è implementata nella funzione `prior()`:

```
prior <- function(param, alpha = 2, beta = 10) {
  param_vals <- seq(0, 1, length.out = 100)
  dbeta(param, alpha, beta) # / sum(dbeta(param_vals, alpha, beta))
}

tibble(
  x = param,
```

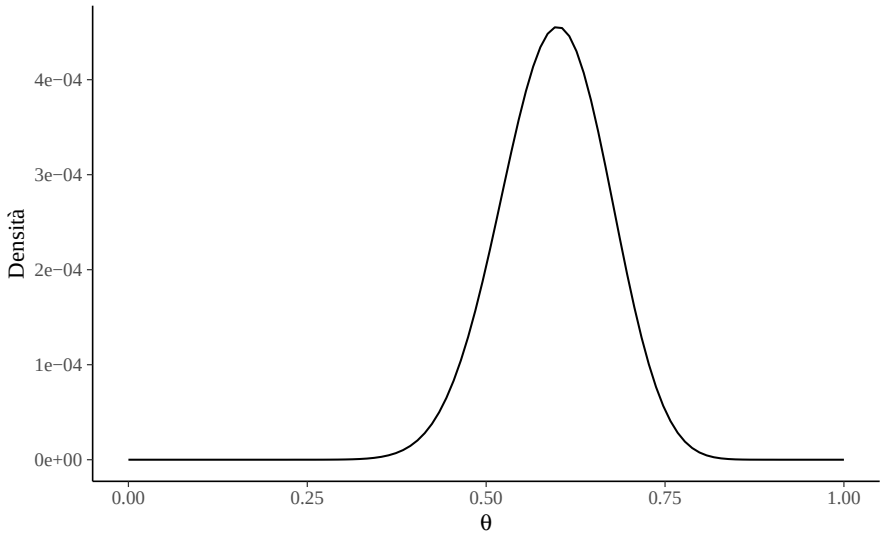
```
  y = prior(param)
) %>%
  ggplot(aes(x, y)) +
  geom_line() +
  labs(
    x = expression(theta),
    y = "Densità"
  )
```



La funzione `posterior()` ritorna il prodotto della densità a priori e della verosimiglianza:

```
posterior <- function(param) {
  likelihood(param) * prior(param)
}

tibble(
  x = param,
  y = posterior(param)
) %>%
  ggplot(aes(x, y)) +
  geom_line() +
```

```
labs(
  x = expression(theta),
  y = "Densità"
)
```



La distribuzione a posteriori non normalizzata mostrata nella figura replica il risultato ottenuto con il codice utilizzato nella prima parte di questo Capitolo. Per l'implementazione dell'algoritmo di Metropolis non è necessaria la normalizzazione della distribuzione a posteriori.

K

Integrazione di Monte Carlo

Il termine Monte Carlo si riferisce al fatto che la computazione fa ricorso ad un ripetuto campionamento casuale attraverso la generazione di sequenze di numeri casuali. Una delle sue applicazioni più potenti è il calcolo degli integrali mediante simulazione numerica. Sia l'integrale da calcolare

$$\int_a^b h(y)dy.$$

Se decomponiamo $h(y)$ nel prodotto di una funzione $f(y)$ e una funzione di densità di probabilità $p(y)$ definita nell'intervallo (a, b) avremo:

$$\int_a^b h(y)dy = \int_a^b f(y)p(y)dy = \mathbb{E}[f(y)],$$

così che l'integrale può essere espresso come una funzione di aspettazione $f(y)$ sulla densità $p(y)$. Se definiamo un gran numero di variabili casuali $y_1, y_2, \dots, y_n$ appartenenti alla densità di probabilità $p(y)$ allora avremo

$$\int_a^b h(y)dy = \int_a^b f(y)p(y)dy = \mathbb{E}[f(y)] \approx \frac{1}{n}\sum_{i=1}^{n} f(y)$$

che è l'integrale di Monte Carlo.

L'integrazione con metodo Monte Carlo trova la sua giustificazione nella *Legge forte dei grandi numeri*. Data una successione di variabili casuali $Y_1, Y_2, \dots, Y_n, \dots$ indipendenti e identicamente distribuite con media μ, ne segue che

$$P\left(\lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} Y_i = \mu\right) = 1.$$

Ciò significa che, al crescere di n, la media delle realizzazioni di $Y_1, Y_2, \dots, Y_n, \dots$ converge con probabilità 1 al vero valore μ.

Possiamo fornire un esempio intuitivo della legge forte dei grandi numeri facendo riferimento ad una serie di lanci di una moneta dove $Y = 1$ significa "testa" e $Y = 0$ significa "croce". Per la legge forte dei grandi numeri, nel caso di una moneta equilibrata la proporzione di eventi "testa" converge alla vera probabilità dell'evento "testa"

$$\frac{1}{n}\sum_{i=1}^{n} Y_i \to \frac{1}{2}$$

con probabilità di uno.

Quello che è stato detto sopra non è che un modo sofisticato per dire che, se vogliamo calcolare un'approssimazione del valore atteso di una variabile casuale, non dobbiamo fare altro che la media aritmetica di un grande numero di realizzazioni della variabile casuale. Come è facile intuire, l'approssimazione migliora al crescere del numero di dati che abbiamo a disposizione.

L'integrazione di Monte Carlo può essere usata per approssimare la distribuzione a posteriori richiesta da una analisi Bayesiana: una stima di $p(\theta \mid y)$ può essere ottenuta mediante un grande numero di campioni casuali della distribuzionea a posteriori.

L

Le catene di Markov

Per introdurre il concetto di catena di Markov, supponiamo che una persona esegua una passeggiata casuale sulla retta dei numeri naturali considerando solo i valori 1, 2, 3, 4, 5, 6.[1] Se la persona è collocata su un valore interno dei valori possibili (ovvero, 2, 3, 4 o 5), nel passo successivo è altrettanto probabile che rimanga su quel numero o si sposti su un numero adiacente. Se si muove, è ugualmente probabile che si muova a sinistra o a destra. Se la persona si trova su uno dei valori estremi (ovvero, 1 o 6), nel passo successivo è altrettanto probabile che rimanga rimanga su quel numero o si sposti nella posizione adiacente.

Questo è un esempio di una catena di Markov discreta. Una catena di Markov descrive il movimento probabilistico tra un numero di stati. Nell'esempio ci sono sei possibili stati, da 1 a 6, i quali corrispondono alle possibili posizioni della passeggiata casuale. Data la sua posizione corrente, la persona si sposterà nelle altre posizioni possibili con delle specifiche probabilità. La probabilità che si sposti in un'altra posizione dipende solo dalla sua posizione attuale e non dalle posizioni visitate in precedenza.

È possibile descrivere il movimento tra gli stati nei termini delle cosiddette *probabilità di transizione*, ovvero le probabilità di movimento tra tutti i possibili stati in un unico passaggio di una catena di Markov. Le probabilità di transizione sono riassunte in una *matrice di transizione* P:

```
p <- c(0, 0, 1, 0, 0, 0)

P <- matrix(
  c(.5, .5, 0, 0, 0, 0,
    .25, .5, .25, 0, 0, 0,
```

[1] Seguiamo qui la presentazione fornita da Bob Carpenter[2].

```
    0, .25, .5, .25, 0, 0,
    0, 0, .25, .5, .25, 0,
    0, 0, 0, .25, .5, .25,
    0, 0, 0, 0, .5, .5
    ),
  nrow = 6, ncol = 6, byrow = TRUE)

kableExtra::kable(P)
```

0.50	0.50	0.00	0.00	0.00	0.00
0.25	0.50	0.25	0.00	0.00	0.00
0.00	0.25	0.50	0.25	0.00	0.00
0.00	0.00	0.25	0.50	0.25	0.00
0.00	0.00	0.00	0.25	0.50	0.25
0.00	0.00	0.00	0.00	0.50	0.50

La prima riga della matrice di transizione P fornisce le probabilità di passare a ciascuno degli stati da 1 a 6 in un unico passaggio a partire dalla posizione 1; la seconda riga fornisce le probabilità di transizione in un unico passaggio dalla posizione 2 e così via. Per esempio, il valore $P[1, 1]$ ci dice che, se la persona è nello stato 1, avrà una probabilità di 0.5 di rimanere in quello stato; $P[1, 2]$ ci dice che c'è una probabilità di 0.5 di passare dallo stato 1 allo stato 2. Gli altri elementi della prima riga sono 0 perché, in un unico passaggio, non è possibile passare dallo stato 1 agli stati 3, 4, 5 e 6. Il valore $P[2, 1]$ ci dice che, se la persona è nello stato 1 (seconda riga), avrà una probabilità di 0.25 di passare allo stato 1; avra una probabilità di 0.5 di rimanere in quello stato, $P[2, 2]$; e avrà una probabilità di 0.25 di passare allo stato 3, $P[2, 3]$; eccetera.

Si notino alcune importanti proprietà di questa particolare catena di Markov.

- È possibile passare da ogni stato a qualunque altro stato in uno o più passaggi: una catena di Markov con questa proprietà si dice *irriducibile*.
- Dato che la persona si trova in un particolare stato, se può tornare a questo stato solo a intervalli regolari, si dice che la catena di Markov è *periodica*. In questo esempio la catena è *aperiodica* poiché la passeggiata casuale non può eitornare allo stato attuale a intervalli regolari.

Un'importante proprietà di una catena di Markov irriducibile e aperiodica è che il passaggio ad uno stato del sistema dipende unicamente dallo stato immediatamente precedente e non dal come si è giunti a tale stato (dalla storia). Per questo motivo si dice che un processo markoviano è senza memoria. Tale "assenza di memoria" può essere interpretata come la proprietà mediante cui è possibile ottenere un insieme di campioni casuali da una distribuzione di interesse. Nel caso dell'inferenza bayesiana, la distribuzione di interesse è la distribuzione a posteriori, $p(\theta \mid y)$. Le catene di Markov consentono di stimare i valori di aspettazione di variabili rispetto alla distribuzione a posteriori.

La matrice di transizione che si ottiene dopo un enorme numero di passi di una passeggiata casuale markoviana si chiama *distribuzione stazionaria*. Se una catena di Markov è irriducibile e aperiodica, allora ha un'unica distribuzione stazionaria w. La distribuzione limite di una tale catena di Markov, quando il numero di passi tende all'infinito, è uguale alla distribuzione stazionaria w.

L.1 Simulare una catena di Markov

Un metodo per dimostrare l'esistenza della distribuzione stazionaria di una catena di Markov è quello di eseguire un esperimento di simulazione. Iniziamo una passeggiata casuale partendo da un particolare stato, diciamo la posizione 3, e quindi simuliamo molti passaggi della catena di Markov usando la matrice di transizione P. Al crescere del numero di passi della catena, le frequenze relative che descrivono il passaggio a ciascuno dei sei possibili nodi della catena approssimano sempre meglio la distribuzione stazionaria w.

Senza entrare nei dettagli della simulazione, la figura L.1 mostra i risultati ottenuti in 10,000 passi di una passeggiata casuale markoviana. Si noti che, all'aumentare del numero di iterazioni, le frequenze relative approssimano sempre meglio le probabilità nella distribuzione stazionaria $w = (0.1, 0.2, 0.2, 0.2, 0.2, 0.1)$.

```
set.seed(123)
s <- vector("numeric", 10000)
s[1] <- 3
```

```r
for (j in 2:10000) {
  s[j] <- sample(1:6, size = 1, prob = P[s[j - 1], ])
}
S <- data.frame(
  Iterazione = 1:10000,
  Location = s
)

S %>%
  mutate(
    L1 = (Location == 1),
    L2 = (Location == 2),
    L3 = (Location == 3),
    L4 = (Location == 4),
    L5 = (Location == 5),
    L6 = (Location == 6)
  ) %>%
  mutate(
    Proporzione_1 = cumsum(L1) / Iterazione,
    Proporzione_2 = cumsum(L2) / Iterazione,
    Proporzione_3 = cumsum(L3) / Iterazione,
    Proporzione_4 = cumsum(L4) / Iterazione,
    Proporzione_5 = cumsum(L5) / Iterazione,
    Proporzione_6 = cumsum(L6) / Iterazione
  ) %>%
  select(
    Iterazione, Proporzione_1, Proporzione_2, Proporzione_3,
    Proporzione_4, Proporzione_5, Proporzione_6
  ) -> S1

gather(S1, Outcome, Probability, -Iterazione) -> S2

ggplot(S2, aes(Iterazione, Probability)) +
  geom_line() +
  facet_wrap(~Outcome, ncol = 3) +
  ylim(0, .4) +
  ylab("Frequenza relativa") +
  # theme(text=element_text(size=14))  +
```

```
scale_x_continuous(breaks = c(0, 3000, 6000, 9000))
```

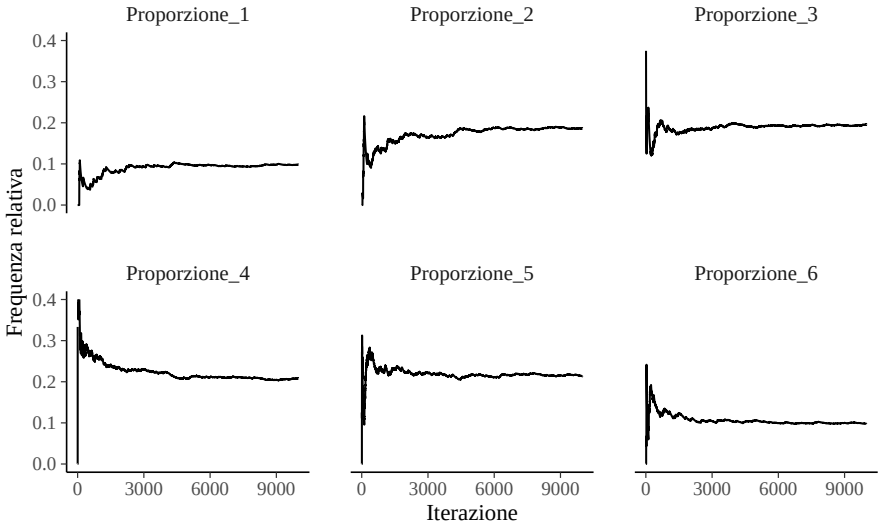


Figura L.1: *Frequenze relative degli stati da 1 a 6 in funzione del numero di iterazioni per la simulazione di una catena di Markov.*

Il metodo di campionamento utilizzato dagli algoritmi MCMC consente di creare una catena di Markov irriducibile e aperiodica, la cui distribuzione stazionaria equivale alla distribuzione a posteriori $p(\theta \mid y)$.

M

Programmare in Stan

M.1 Che cos'è Stan?

STAN[1] è un linguaggio di programmazione probabilistico usato per l'inferenza bayesiana (Carpenter et al., 2017). Prende il nome da uno dei creatori del metodo Monte Carlo, Stanislaw Ulam (Eckhardt, 1987). Stan consente di generare campioni da distribuzioni di probabilità basati sulla costruzione di una catena di Markov avente come distribuzione di equilibrio (o stazionaria) la distribuzione desiderata.

È possibile accedere al linguaggio Stan tramite diverse interfacce:

- `CmdStan`: eseguibile da riga di comando,
- `RStan` - integrazione con il linguaggio R;
- `PyStan` - integrazione con il linguaggio di programmazione Python;
- `MatlabStan` - integrazione con MATLAB;
- `Stan.jl` - integrazione con il linguaggio di programmazione Julia;
- `StataStan` - integrazione con Stata.

Inoltre, vengono fornite interfacce di livello superiore con i pacchetti che utilizzano Stan come backend, principalmente in Linguaggio R:

- `shinystan`: interfaccia grafica interattiva per l'analisi della distribuzione a posteriori e le diagnostiche MCMC;
- `bayesplot`: insieme di funzioni utilizzabili per creare grafici relativi all'analisi della distribuzione a posteriori, ai test del modello e alle diagnostiche MCMC;
- `brms`: fornisce un'ampia gamma di modelli lineari e non lineari specificando i modelli statistici mediante la sintassi usata in R;

[1] http://mc-stan.org/

- `rstanarm`: fornisce un sostituto per i modelli frequentisti forniti da base R e `lme4` utilizzando la sintassi usata in R per la specificazione dei modelli statistici;
- `edstan`: modelli Stan per la Item Response Theory;
- `cmdstanr`, un'interfaccia R per `CmdStan`.

M.2 Interfaccia `cmdstanr`

Negli esempi di questa dispensa verrà usata l'interfaccia `cmdstanr`. Il pacchetto `cmdstanr` non è ancora disponibile su CRAN, ma può essere installato come indicato su questo link[2]. Una volta che è stato installato, il pacchetto `cmdstanr` può essere caricato come un qualsiasi altro pacchetto R.

Si noti che `cmdstanr` richiede un'installazione funzionante di `CmdStan`, l'interfaccia shell per Stan. Se `CmdStan` non è installato, `cmdstanr` lo installerà automaticamente se il computer dispone di una *Toolchain* adatta. Stan richiede infatti che sul computer su cui viene installato siano presenti alcuni strumenti necessari per gestire i file C++. Tra le altre ragioni, questo è dovuto al fatto che il codice Stan viene tradotto in codice C++ e compilato. Il modo migliore per ottenere il software necessario per un computer Windows o Mac è quello di installare `RTools`. Per un computer Linux, è necessario installare `build-essential` e una versione recente dei compilatori g++ o clang++. I requisiti sono descritti nella Guida di CmdStan[3].

Per verificare che la Toolchain sia configurata correttamente è possibile utilizzare la funzione `check_cmdstan_toolchain()`:

```
check_cmdstan_toolchain()
```

Se la toolchain è configurata correttamente, `CmdStan` può essere installato mediante la funzione `install_cmdstan()`:

[2] https://mc-stan.org/docs/2_27/cmdstan-guide/cmdstan-installation.html
[3] https://mc-stan.org/docs/cmdstan-guide/cmdstan-installation.html

```
# do not run!
# install_cmdstan(cores = 2)
```

La versione installata di `CmdStan` si ottiene con:

```
cmdstan_version()
#> [1] "2.28.2"
```

M.3 Codice Stan

Qualunque sia l'interfaccia che viene usata, i modelli sottostanti sono sempre scritti nel linguaggio Stan, il che significa che lo stesso codice Stan è valido per tutte le interfacce possibili. Il codice Stan è costituito da una serie di blocchi che vengono usati per specificare un modello statistico. In ordine, questi blocchi sono: `data`, `transformed data`, `parameters`, `transformed parameters`, `model`, e `generated quantities`.

M.3.1 "Hello, world" – Stan

Quando si studia un nuovo linguaggio di programmazione si utilizza spesso un programma "Hello, world". Questo è un modo semplice, spesso minimo, per dimostrare alcune delle sintassi di base del linguaggio. In Python, il programme "Hello, world" program è:

```
print("Hello, world.")
```

Qui presentiamo Stan e scriviamo un programma "Hello, world" per Stan.

Prima di scrivere il nostro primo programma "Hello, world" per Stan (che estrarrà campioni dalla distribuzione a posteriori di un modello gaussiano) spendiamo due parole per spiegare cosa fa Stan. Un utente scrive un modello usando il linguaggio Stan. Questo è solitamente memorizzato in un file di testo `.stan`. Il modello viene compilato in due passaggi. Innanzitutto, Stan traduce il modello nel file `.stan` in codice C++. Quindi, quel codice C++ viene compilato in codice macchina. Una volta creato

il codice macchina, l'utente può, tramite l'interfaccia CmdStan, campionare la distribuzione definita dal modello ed eseguire altri calcoli con il modello. I risultati del campionamento vengono scritti su disco come file CSV e txt. Come mostrato di seguito, l'utente accede a questi file utilizzando varie funzioni R, senza interagire direttamente con loro.

Per iniziare, possiamo dire che un programma Stan contiene tre "blocchi" obbligatori: blocco `data`, blocco `parameters`, blocco `model`.

M.3.2 Blocco `data`

Qui vengono dichiarate le variabili che saranno passate a Stan. Devono essere elencati i nomi delle variabili che saranno utilizzate nel programma, il *tipo di dati* da registrare per ciascuna variabile, per esempio:

- *int* = intero,
- *real* = numeri reali (ovvero, numeri con cifre decimali),
- *vector* = sequenze ordinate di numeri reali unidimensionali,
- *matrix* = matrici bidimensionali di numeri reali,
- *array* = sequenze ordinate di dati multidimensionali.

Devono anche essere dichiarate le dimensioni delle variabili e le eventuali restrizioni sulle variabili (es. `upper = 1 lower = 0`, che fungono da controlli per Stan). Tutti i nomi delle variabili assegnate qui saranno anche usati negli altri blocchi del programma.

Per esempio, l'istruzione seguente dichiaria la variabile `Y` – la quale rappresenta, ad esempio, l'altezza di 10 persone – come una variabile di tipo `real[10]`. Ciò significa che specifichiamo un array di lunghezza 10, i cui elementi sono variabili continue definite sull'intervallo dei numeri reali $[-\infty, +\infty]$.

```
data {
  real Y[10]; // heights for 10 people
}
```

Invece, con l'istruzione

```
data {
  int Y[10]; // qi for 10 people
}
```

dichiariamo la variabile `Y` – la quale rappresenta, ad esempio, il QI di 10 persone – come una variabile di tipo `int[10]`, ovvero un array di lunghezza 10, i cui elementi sono numeri naturali, cioè numeri interi non negativi $\{0, +1, +2, +3, +4, ...\}$.

Un altro esempio è

```
data {
  real<lower=0, upper=1> Y[10]; // 10 proportions
}
```

nel quale viene specificato un array di lunghezza 10, i cui elementi sono delle variabili continue definite sull'intervallo dei numeri reali $[0, 1]$ — per esempio, delle proporzioni.

Si noti che i tipi `vector` e `matrix` contengono solo elementi di tipo `real`, ovvero variabili continue, mentre gli `array` possono contenere dati di qualsiasi tipo. I dati passati a Stan devono essere contenuti in un oggetto del tipo `list`.

M.3.3 Blocco `parameters`

I parametri che vengono stimati sono dichiarati nel blocco `parameters`. Per esempio, l'istruzione

```
parameters {
  real mu; // mean height in population
  real<lower=0> sigma; // sd of height distribution
}
```

dichiaria la variabile `mu` che codifica l'altezza media nella popolazione, che è una variabile continua in un intervallo illimitato di valori, e la deviazione standard `sigma`, che è una variabile continua non negativa. Avremmo anche potuto specificare un limite inferiore di zero su `mu` perché deve essere non negativo.

Per una regressione lineare semplice, ad esempio, devono essere dichiarate le variabili corrispondenti all'intercetta (`alpha`), alla pendenza (`beta`) e alla deviazione standard degli errori attorno alla linea di regressione (`sigma`). In altri termini, nel blocco `parameters` devono essere elencati tutti i parametri che dovranno essere stimati dal modello. Si noti che para-

metri discreti non sono possibili. Infatti, Stan attualmente non supporta i parametri con valori interi, almeno non direttamente.

M.3.4 Blocco `model`

Nel blocco `model` vengono elencate le dichiarazioni relative alla verosimiglianza dei dati e alle distribuzioni a priori dei parametri, come ad esempio, nelle istruzioni seguenti.

```
model {
  for(i in 1:10) {
    Y[i] ~ normal(mu, sigma);
  }
  mu ~ normal(170, 15); // prior for mu
  sigma ~ cauchy(0, 20); // prior for sigma
}
```

Mediante l'istruzione all'interno del ciclo `for`, ciascun valore dell'altezza viene concepito come una variable casuale proveniente da una distribuzione Normale di parametri μ e σ (i parametri di interesse nell'inferenza). Il ciclo `for` viene ripetuto 10 volte perché i dati sono costituiti da un array di 10 elementi (ovvero, il campione è costituito da 10 osservazioni).

Le due righe che seguno il ciclo `for` specificano le distribuzioni a priori dei parametri su cui vogliamo effettuare l'inferenza. Per μ assumiamo una distribuzione a priori Normale di parametri $\mu = 170$ e $\sigma = 15$; per σ assumiamo una distribuzione a priori Cauchy(0, 20).

Se non viene definita alcuna distribuzione a priori, Stan utilizzerà la distribuzione a priori predefinita $Unif(-\infty, +\infty)$. Raccomandazioni sulle distribuzioni a priori sono fornite in questo link[4].

La precedente notazione di campionamento può anche essere espressa usando la seguente notazione alternativa:

```
  for(i in 1:10) {
    target += normal_lpdf(Y[i] | mu, sigma);
  }
```

[4] https://github.com/stan-dev/stan/wiki/Prior-Choice-Recommendations

Questa notazione rende trasparente il fatto che, in pratica, Stan esegue un campionamento nello spazio

$$\log p(\theta \mid y) \propto \log p(y \mid \theta) + \log p(\theta) = \sum_{i=1}^{n} \log p(y_i \mid \theta) + \log p(\theta).$$

Per ogni passo MCMC, viene ottenuto un nuovo valore di μ e σ eviene valutata la log densità a posteriori non normalizzata. Ad ogni passo MCMC, Stan calcola un nuovo valore della densità a posteriori su scala logaritmica partendo da un valore di 0 e incrementandola ogni volta che incontra un'istruzione ~. Quindi, le istruzioni precedenti aumentano la log-densità di una quantità pari a $\log(p(Y[i])) \propto -\frac{1}{2}\log(\sigma^2) - (Y[i] - \mu)^2/2\sigma^2$ per le altezze si ciascuno degli $i = 1 \dots, 10$ individui – laddove la formula esprime, in termini logaritmici, la densità Normale da cui sono stati esclusi i termini costanti.

Oppure, in termini vettorializzati, il modello descritto sopra può essere espresso come

```
model {
  Y ~ normal(mu, sigma);
}
```

dove il termine a sinistra di ~ è un array. Questa notazione più compatta è anche la più efficiente.

M.3.5 Blocchi opzionali

Ci sono inoltre tre blocchi opzionali:

- Il blocco `transformed data` consente il pre-processing dei dati. È possibile trasformare i parametri del modello; solitamente ciò viene fatto nel caso dei modelli più avanzati per consentire un campionamento MCMC più efficiente.
- Il blocco `transformed parameters` consente la manipolazione dei parametri prima del calcolo della distribuzione a posteriori.
- Il blocco `generated quantities` consente il post-processing riguardante qualsiasi quantità che non fa parte del modello ma può essere calcolata a partire dai parametri del modello, per ogni iterazione dell'algoritmo. Esempi includono la generazione dei campioni a posteriori e le dimensioni degli effetti.

M.3.6 Sintassi

Si noti che il codice Stan richiede i punti e virgola (;) alla fine di ogni istruzione di assegnazione. Questo accade per le dichiarazioni dei dati, per le dichiarazioni dei parametri e ovunque si acceda ad un elemento di un tipo `data` e lo si assegni a qualcos'altro. I punti e virgola non sono invece richiesti all'inizio di un ciclo o di un'istruzione condizionale, dove non viene assegnato nulla.

In STAN, qualsiasi stringa che segue `//` denota un commento e viene ignorata dal programma.

Stan è un linguaggio estremamente potente e consente di implementare quasi tutti i modelli statistici, ma al prezzo di un certo sforzo di programmazione. Anche l'adattamento di semplici modelli statistici mediante il linguaggio STAN a volte può essere laborioso. Per molti modelli comunemente usati, come i modelli di regressione e multilivello, tale processo può essere semplificato usando le funzioni del pacchetto `brms`. D'altra parte, per modelli veramente complessi, non ci sono molte alternative all'uso di STAN. Per chi è curioso, il manuale del linguaggio Stan è accessibile al seguente link[5].

M.4 Workflow

Se usiamo `cmdstanr`, dobbiamo prima scrivere il codice con il modello statistico in un file in formato Stan. È necessario poi "transpile" quel file, ovvero tradurre il file in C++ e compilarlo. Ciò viene fatto mediante la funzione `cmdstan_model()`. Possiamo poi eseguire il campionamento MCMC con il metodo `$sample()`. Infine è possibile creare un sommario dei risultati usando, per esempio, usando il metodo `$summary()`.

M.5 Ciao, Stan

Scriviamo ora il nostro programma Stan "Hello, world" per generare campioni da una distribuzione Normale standard (con media zero e varianza

[5] https://mc-stan.org/docs/2_27/stan-users-guide/index.html

unitaria).

```
modelString = "
parameters {
  real x;
}
model {
  x ~ normal(0, 1);
}
"
writeLines(modelString, con = "code/hello_world.stan")
```

Si noti che ci sono solo due blocchi in questo particolare codice Stan, il blocco parametri e il blocco modello. Questi sono due dei sette blocchi possibili in un codice Stan. Nel blocco parametri, abbiamo i nomi e i tipi di parametri per i quali vogliamo ottenere i campioni. In questo caso, vogliamo ottenere campioni di numeri reale che chiamiamo `x`. Nel blocco modello, abbiamo il nostro modello statistico. Specifichiamo che x, il parametro di cui vogliamo ottenere i campioni, è normalmente distribuito con media zero e deviazione standard unitaria. Ora che abbiamo il nostro codice (che è stato memorizzato in un file chiamato `hello_world.stan`), possiamo usare CmdStan per compilarlo e ottenere `mod`, che è un oggetto R che fornisce l'accesso all'eseguibile Stan compilato.

Leggiamo il file in cui abbiamo salvato il codice Stan

```
file <- file.path("code", "hello_world.stan")
```

compiliamo il modello

```
mod <- cmdstan_model(file)
```

ed eseguiamo il campionamento MCMC:

```
fit <- mod$sample(
  iter_sampling = 4000L,
  iter_warmup = 2000L,
  seed = SEED,
```

```
  chains = 4L,
  refresh = 0,
  thin = 1
)
```

Tasformiamo l'oggetto `fit` nel formato `stanfit` per manipolarlo più facilmente:

```
stanfit <- rstan::read_stan_csv(fit$output_files())
```

Lo esaminiamo

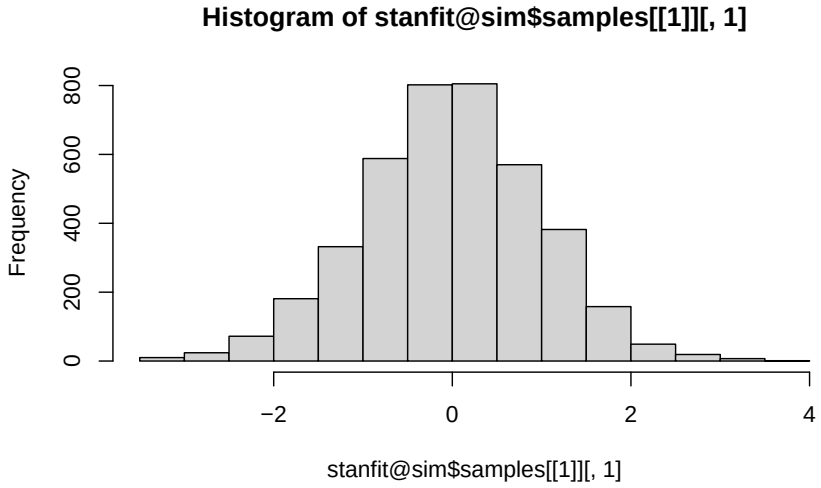
```
length(stanfit@sim$samples)
#> [1] 4
```

Quello che abbiamo ottenuto sono 4 catene di 4000 osservazioni ciascuna, le quali contengono valori casuali estratti dalla gaussiana standardizzata:

```
head(stanfit@sim$samples[[1]])
```

Verifichiamo

```
hist(stanfit@sim$samples[[1]][, 1])
```


Histogram of stanfit@sim$samples[[1]][, 1]
Frequency
0
200
400
600
800
−2
0
2
4
stanfit@sim$samples[[1]][, 1]

N

Inferenza su una proporzione con Stan

Il Capitolo **??** discute il codice Stan necessario per calcolare $p(y^{rep} \mid y)$ nel caso dell'inferenza su una proporzione. Questa Appendice approfondisce alcuni aspetti di tale analisi statistica.

Assumiamo che il codice Stan descritto nel Capitolo **??** abbia prodotto l'oggetto `fit`.

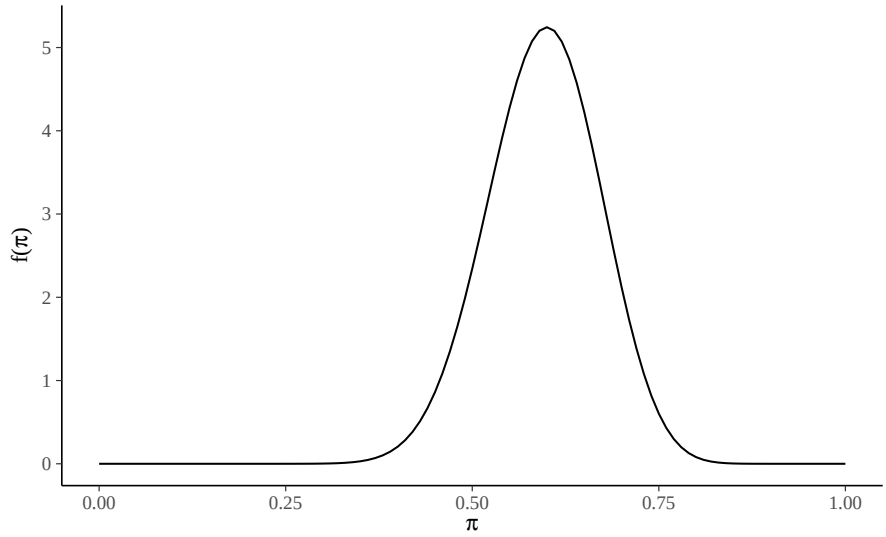
Trasformiamo `fit` in un oggetto di classe `stanfit`:

```
stanfit <- rstan::read_stan_csv(fit$output_files())
```

e esaminiamo il risultato ottenuto. Per i dati dell'esempio, l'esatta distribuzione a posteriori è una Beta(25, 17):

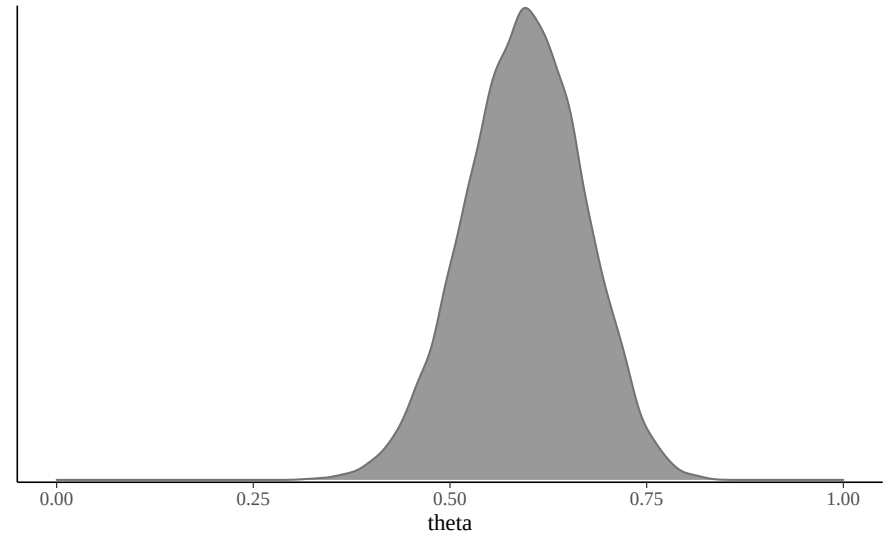
```
summarize_beta_binomial(alpha = 2, beta = 10, y = 23, n = 30)
#>       model alpha beta   mean mode      var      sd
#> 1     prior     2   10 0.1667  0.1 0.010684 0.10336
#> 2 posterior    25   17 0.5952  0.6 0.005603 0.07485
```

```
plot_beta(alpha = 25, beta = 17) +
  lims(x = c(0, 1))
```



L'approssimazione della distribuzione a posteriori per θ ottenuta mediante la simulazione MCMC è

```
mcmc_dens(stanfit, pars = "theta") +
  lims(x = c(0, 1))
```



La funzione `tidy()` nel pacchetto `broom.mixed` fornisce alcune utili statistiche per i 16000 valori della catena Markov memorizzati in `stanfit`:

```
broom.mixed::tidy(
  stanfit,
  conf.int = TRUE,
  conf.level = 0.95,
  pars = "theta"
)
#> # A tibble: 1 x 5
#>   term  estimate std.error conf.low conf.high
#>   <chr>    <dbl>     <dbl>    <dbl>     <dbl>
#> 1 theta    0.596    0.0746    0.445     0.734
```

laddove, per esempio, la media esatta della corretta distribuzione a posteriori è

```
25 / (25 + 17)
#> [1] 0.5952
```

La funzione `tidy()` non consente di calcolare altre statistiche descrittive, oltre alla media. Ma questo risultato può essere ottenuto direttamente utilizzando i valori delle catene di Markov. Iniziamo ad esaminare il contenuto dell'oggetto `stanfit`:

```
list_of_draws <- extract(stanfit)
print(names(list_of_draws))
#> [1] "theta"   "y_rep"   "log_lik" "lp__"
```

Possiamo estrarre i campioni della distribuzione a posteriori nel modo seguente:

```
head(list_of_draws$theta)
#> [1] 0.5841 0.6536 0.7287 0.6072 0.6657 0.6527
```

Creiamo un data.frame con le stime a posteriori $\hat{\theta}$:

```
df <- data.frame(
  theta = list_of_draws$theta
)
```

Le statistiche descrittive della distribuzione a posteriori possono ora essere ottenute usando direttamente i valori $\hat{\theta}$:

```
df %>%
  summarize(
    post_mean = mean(theta),
    post_median = median(theta),
    post_mode = sample_mode(theta),
    lower_95 = quantile(theta, 0.025),
    upper_95 = quantile(theta, 0.975)
  )
#>   post_mean post_median post_mode lower_95 upper_95
#> 1    0.5946      0.5958    0.5955   0.4451   0.7345
```

È anche possibile calcolare, ad esempio, la probabilità di $\hat{\theta} > 0.5$:

```
df %>%
  mutate(exceeds = theta > 0.50) %>%
  janitor::tabyl(exceeds)
#>  exceeds     n percent
#>    FALSE  1706  0.1066
#>     TRUE 14294  0.8934
```

O

Minimi quadrati

Nella trattazione classica del modello di regressione, $y_i = \alpha + \beta x_i + e_i$, i coefficienti $a = \hat{\alpha}$ e $b = \hat{\beta}$ vengono stimati in modo tale da minimizzare i residui

$$e_i = y_i - \hat{\alpha} - \hat{\beta} x_i. \tag{O.1}$$

In altri termini, il residuo i-esimo è la differenza fra l'ordinata del punto $(x_i,\ y_i)$ e quella del punto di ascissa x_i sulla retta di regressione campionaria.

Per determinare i coefficienti a e b della retta $y_i = a + bx_i + e_i$ non è sufficiente minimizzare la somma dei residui $\sum_{i=1}^{n} e_i$, in quanto i residui possono essere sia positivi che negativi e la loro somma può essere molto prossima allo zero anche per differenze molto grandi tra i valori osservati e la retta di regressione. Infatti, ciascuna retta passante per il punto $(\bar{x}, \bar{y})$ ha $\sum_{i=1}^{n} e_i = 0$.

Una retta passante per il punto $(\bar{x}, \bar{y})$ soddisfa l'equazione $\bar{y} = a + b\bar{x}$. Sottraendo tale equazione dall'equazione $y_i = a + bx_i + e_i$ otteniamo

$$y_i - \bar{y} = b(x_i - \bar{x}) + e_i.$$

Sommando su tutte le osservazioni, si ha che

$$\sum_{i=1}^{n} e_i = \sum_{i=1}^{n} (y_i - \bar{y}) - b \sum_{i=1}^{n} (x_i - \bar{x}) = 0 - b(0) = 0. \tag{O.2}$$

Questo problema viene risolto scegliendo i coefficienti a e b che minimizzano, non tanto la somma dei residui, ma bensì l'*errore quadratico*, cioè la somma dei quadrati degli errori:

$$S(a,b) = \sum_{i=1}^{n} e_i^2 = \sum (y_i - a - bx_i)^2. \tag{O.3}$$

Il metodo più diretto per determinare quelli che vengono chiamati i *coefficienti dei minimi quadrati* è quello di trovare le derivate parziali della funzione $S(a,b)$ rispetto ai coefficienti a e b:

$$\begin{aligned} \frac{\partial S(a,b)}{\partial a} &= \sum (-1)(2)(y_i - a - bx_i), \\ \frac{\partial S(a,b)}{\partial b} &= \sum (-x_i)(2)(y_i - a - bx_i). \end{aligned} \tag{O.4}$$

Ponendo le derivate uguali a zero e dividendo entrambi i membri per -2 si ottengono le *equazioni normali*

$$\begin{aligned} an + b\sum x_i &= \sum y_i, \\ a\sum x_i + b\sum x_i^2 &= \sum x_i y_i. \end{aligned} \tag{O.5}$$

I coefficienti dei minimi quadrati a e b si trovano risolvendo le (O.5) e sono uguali a:

$$a = \bar{y} - b\bar{x}, \tag{O.6}$$

$$b = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2}. \tag{O.7}$$

O.0.1 Massima verosimiglianza

Se gli errori del modello lineare sono indipendenti e distribuiti secondo una Normale, così che $y_i \sim \mathcal{N}(\alpha + \beta x, \sigma^2)$ per ciascun i, allora le stime dei minimi quadrati di α e β corrispondono alla stima di massima verosimiglianza. La funzione di verosimiglianza del modello di regressione è definita come la funzione di densità di probabilità dei dati, dati i parametri e i predittori:

$$p(y \mid \alpha, \beta, \sigma, x) = \prod_{i=1}^{n} \mathcal{N}(y_i \mid \alpha, \beta x_i, \sigma^2). \tag{O.8}$$

Massimizzare la (O.8) conduce alle stime dei minimi quadrati (O.7).

P

Introduzione al linguaggio R

In questa sezione della dispensa saranno presentate le caratteristiche di base e la filosofia dell'ambiente `R`, passando poi a illustrare le strutture dati e le principali strutture di controllo. Verranno introdotte alcune funzioni utili per la gestione dei dati e verranno forniti i rudimenti per realizzare semplici funzioni. Verranno introdotti i tipi di file editabili in RStudio (script, markdown, ...). Nello specifico, dopo aver accennato alcune caratteristiche del sistema `tidyverse`, verranno illustrate le principali funzionalità dell'IDE RStudio e dei pacchetti `dplyr` e `ggplot2`. Sul web sono disponibili tantissime introduzioni all'uso di `R` come, ad esempio, Hands-On Programming with R[1], R for Data Science[2], Data Science for Psychologists[3], e Introduction to Data Science[4].

P.1 Prerequisiti

Al fine di utilizzare `R` è necessario eseguire le seguenti tre operazioni nell'ordine dato:

1. Installare `R`;
2. Installare RStudio;
3. Installare R-Packages (se necessario).

Di seguito viene descritto come installare `R` e RStudio.

[1] https://rstudio-education.github.io/hopr/
[2] https://r4ds.had.co.nz
[3] https://bookdown.org/hneth/ds4psy/
[4] https://bookdown.org/hneth/i2ds/

P.1.1 Installare R e RStudio

R è disponibile gratuitamente ed è scaricabile dal sito `http://www.rproject.org/`. Dalla pagina principale del sito `r-project.org` andiamo sulla sezione `Download` e scegliamo un server a piacimento per scaricare il software d'installazione. Una volta scaricato l'installer, lo installiamo come un qualsiasi software, cliccando due volte sul file d'istallazione. Esistono versioni di R per tutti i più diffusi sistemi operativi (Windows, Mac OS X e Linux).

Il R Core Development Team lavora continuamente per migliorare le prestazioni di R, per correggere errori e per consentire l'uso di con nuove tecnologie. Di conseguenza, periodicamente vengono rilasciate nuove versioni di R. Informazioni a questo proposito sono fornite sulla pagina web `https://www.r-project.org/`. Per installare una nuova versione di R si segue la stessa procedura che è stata seguita per la prima installazione.

Insieme al software si possono scaricare dal sito principale sia manuali d'uso che numerose dispense per approfondire diversi aspetti di R. In particolare, nel sito `http://cran.r-project.org/other-docs.html` si possono trovare anche numerose dispense in italiano (sezione "Other languages").

Dopo avere installato R è opportuno installare anche RStudio. RStudio si può scaricare da `https://www.rstudio.com/`. Anche RStudio è disponibile per tutti i più diffusi sistemi operativi.

P.1.2 Utilizzare RStudio per semplificare il lavoro

Possiamo pensare ad R come al motore di un automobile e a RStudio come al cruscotto di un automobile. Più precisamente, R è un linguaggio di programmazione che esegue calcoli mentre RStudio è un ambiente di sviluppo integrato (IDE) che fornisce un'interfaccia grafica aggiungendo una serie di strumenti che facilitano la fase di sviluppo e di esecuzione del codice. Utilizzeremo dunque R mediante RStudio. In altre parole,

non aprite

aprite invece

L'ambiente di lavoro di RStudio è costituito da quattro finestre: la finestra del codice (scrivere-eseguire script), la finestra della console (riga di comando - output), la finestra degli oggetti (elenco oggetti-cronologia dei comandi) e la finestra dei pacchetti-dei grafici-dell'aiuto in linea.

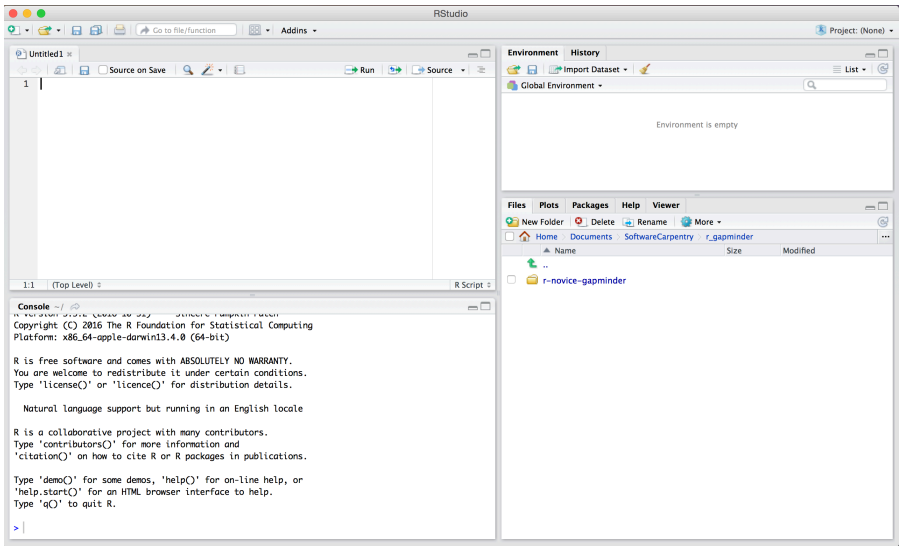


Figura P.1: *La console di RStudio.*

P.1.3 Eseguire il codice

Mediante il menu a tendina di RStudio, scegliendo il percorso

`File > New File > R Notebook`

oppure

`File > New File > R Script`

l'utente può aprire nella finestra del codice (in alto a destra) un R Notebook o un R script dove inserire le istruzioni da eseguire.

In un R script, un blocco di codice viene eseguito selezionando un insieme di righe di istruzioni e digitando la sequenza di tasti `Command` + `Invio` sul Mac, oppure `Control` + `Invio` su Windows. In un R Notebook, un

blocco di codice viene eseguito schiacciando il bottone con l'icona ▶ ("Run current chunk") posizionata a destra rispetto al codice.

P.2 Installare `cmdstan`

È possibile installare `cmdstan` in almento tre modi. Per informazioni dettagliate, si vedano le istruzioni CmdStan Installation[5].

Prima di installare `cmdstan`, tre raccomandazioni generali:

- usare la versione più recente del sistema operativo;
- usare la versione più recente di RStudio;
- usare la versione più recente di R.

Se i tre vincoli precedenti sono soddisfatti, l'installazione di `cmdstan` dovrebbe procedere senza intoppi. Altrimenti si possono creare dei problemi di non facile soluzione.

Il modo più semplice per installare `cmdstan` è quello di installare prima cmdstanr[6] per poi utilizzare le funzionalità di quel pacchetto per l'installazione di `cmdstan`.

Un secondo metodo (che è quello che io uso normalmente) è quello di installare dal sorgente, seguendo le istruzioni riportante su CmdStan Installation[7].

Un terzo metodo (che richiede una minima comprensione delle funzionalità della shell e di Python) richiede, avendo prima installato Anaconda[8], di digitare sulla console del proprio computer (la shell) le seguenti istruzioni:

```
conda create -n stan-env -c conda-forge cmdstan
conda activate stan-env
```

Su `macos`, prima di installare `cmdstan`, è necessario installare la versione più recente di Xcode. Dopo avere installato Xcode, aprire la app. Verrà chiesto all'utente se si vogliono istallare delle componenti aggiuntive.

[5] https://mc-stan.org/docs/2_28/cmdstan-guide/cmdstan-installation.html
[6] https://mc-stan.org/cmdstanr/
[7] https://mc-stan.org/docs/2_28/cmdstan-guide/cmdstan-installation.html
[8] https://www.anaconda.com/products/individual

Questo passaggio è cruciale, perché senza queste componenti aggiuntive `cmdstan` non funzionerà. Dopo avere installato le componenti aggiuntive, aprire Xcode e, in caso, accettare i termini della licenza. A quel punto si può chiudere Xcode. Ogni volta che Xcode viene aggiornato (deve sempre essere aggiornato quando un aggiornamento è disponibile), queste operazioni vanno ripetute.

P.3 Sintassi di base

`R` è un linguaggio di programmazione orientato all'analisi dei dati, il calcolo e la visualizzazione grafica. È disponibile su Internet una vasta gamma di materiali utile per avvicinarsi all'ambiente `R` e aiutare l'utente nell'apprendimento di questo software statistico. Cercheremo qui di fornire alcune indicazioni e una breve descrizione delle risorse di base di `R`.

Aggiungo qui sotto alcune considerazioni che ho preso, pari pari, da un testo che tratta di un altro linguaggio di programmazione, ma che si applicano perfettamente anche al caso nostro. *"Come in ogni linguaggio, per parlare in R è necessario seguire un insieme di regole. Come in tutti i linguaggi di programmazione, queste regole sono del tutto inflessibili e inderogabili. In R, un enunciato o è sintatticamente corretto o è incomprensibile all'interprete, che lo segnalerà all'utente. Questo aspetto non è esattamente amichevole per chi non è abituato ai linguaggi di programmazione, e si trova così costretto ad una precisione di scrittura decisamente poco"analogica". Tuttavia, ci sono due aspetti positivi nello scrivere codice, interrelati tra loro. Il primo è lo sforzo analitico necessario, che allena ad un'analisi precisa del problema che si vuole risolvere in modo da poterlo formalizzare linguisticamente. Il secondo concerne una forma di autoconsapevolezza specifica: salvo "bachi" nel linguaggio (rarissimi sebbene possibili), il mantra del programmatore è "Se qualcosa non ti funziona, è colpa tua"* (testo adattato da Andrea Valle).

A chi preferisce un approccio più "giocoso" posso suggerire il seguente link[9].

[9] https://tinystats.github.io/teacups-giraffes-and-statistics/01_introToR.html

P.3.1 Utilizzare la console R come calcolatrice

La console di RStudio contiene un cursore rappresentato dal simbolo ">" (linea di comando) dove si possono inserire i comandi e le funzioni – in realtà è sempre meglio utilizzare un R Notebook anziché la console, ma per ora esaminiamo il funzionamento di quest'ultima.

La console di RStudio può essere utilizzata come semplice calcolatrice. I comandi elementari consistono di espressioni o di assegnazioni. Le operazioni aritmetiche vengono eseguite mediante simboli "standard:" +, *, -, /, `sqrt()`, `log()`, `exp()`, …

I comandi sono separati da un carattere di nuova linea (si immette un carattere di nuova linea digitando il tasto `Invio`). Se un comando non è completo alla fine della linea, R darà un prompt differente che per default è il carattere + sulla linea seguente e continuerà a leggere l'input finché il comando non è sintatticamente completo. Ad esempio,

```
4 -
+
+ 1
#> [1] 3
```

R è un ambiente interattivo, ossia i comandi producono una risposta immediata. Se scriviamo `2 + 2` e premiamo il tasto di invio, comparirà nella riga successiva il risultato:

```
2 + 2
#> [1] 4
```

Il risultato è preceduto da `[1]`, il che significa che il risultato dell'operazione che abbiamo appena eseguito è il primo valore di questa linea. Alcune funzioni ritornano più di un singolo numero e, in quel caso, l'informazione fornita da R è più utile. Per esempio, l'istruzione `100:130` ritorna 31 valori, ovvero i numeri da 100 a 130:

```
100:130
#>  [1] 100 101 102 103 104 105 106 107 108 109 110 111
#> [13] 112 113 114 115 116 117 118 119 120 121 122 123
#> [25] 124 125 126 127 128 129 130
```

In questo caso, sul mio computer, `[24]` indica che il valore 123 è il ventiquattresimo numero che è stato stampato sulla console – su un altro computer le cose possono essere diverse in quanto il risultato, credo, dipende dalla grandezza dello schermo.

P.3.2 Espressioni

In questo corso, cercheremo di evitare i numeri nei nomi R, così come le lettere maiuscole e .. Useremo quindi nomi come: my_data, anova_results, square_root, ecc.

Un'espressione in `R` è un enunciato finito e autonomo del linguaggio: una frase conclusa, si potrebbe dire. Si noti che le espressioni in `R` non sono delimitate dal `;` come succede in alcuni linguaggi di programmazione. L'ordine delle espressioni è l'ordine di esecuzione delle stesse.

L'a capo non è rilevante per `R`. Questo permette di utilizzare l'a capo per migliorare la leggibilità del codice.

P.3.3 Oggetti

`R` è un linguaggio di programmazione a oggetti, quindi si basa sulla creazione di oggetti e sulla possibilità di salvarli nella memoria del programma. `R` distingue tra maiuscole e minuscole come la maggior parte dei linguaggi basati su UNIX, quindi `A` e `a` sono nomi diversi e fanno riferimento a oggetti diversi.

I comandi elementari di `R` consistono in espressioni o assegnazioni.

Se un'espressione viene fornita come comando, viene valutata, stampata sullo schermo e il valore viene perso, come succedeva alle operazioni aritmetiche che abbiamo presentato sopra discutendo l'uso della console `R` come calcolatrice.

Un'assegnazione crea un oggetto oppure valuta un'espressione e passa il valore a un oggetto, ma il risultato non viene stampato automaticamente sullo schermo. Per l'operazione di assegnazione si usa il simbolo `<-`. Ad esempio, per creare un oggetto che contiene il risultato dell'operazione `2 + 2` procediamo nel modo seguente:

```
res_sum <- 2 + 2
res_sum
#> [1] 4
```

L'operazione di assegnazione (`<-`) copia il contenuto dell'operando destro (detto `r-value`) nell'operando sinistro detto (`l-value`). Il valore dell'espressione assegnazione è `r-value`. Nell'esempio precedente, `res_sum` (`l-value`) assume il valore di 4.

P.3.4 Variabili

L'oggetto `res_sum` è una *variabile*. Una spiegazione di ciò che questo significa è riportata qui sotto. *"Una variabile è un segnaposto. Tutte le volte che si memorizza un dato lo si assegna ad una variabile. Infatti, se il dato è nella memoria, per potervi accedere, è necessario conoscere il suo indirizzo, la sua "etichetta" (come in un grande magazzino in cui si va a cercare un oggetto in base alla sua collocazione). Se il dato è memorizzato ma inaccessibile (come nel caso di un oggetto sperso in un magazzino), allora non si può usare ed è soltanto uno spreco di spazio. La teoria delle variabili è un ambito molto complesso nella scienza della computazione. Ad esempio, una aspetto importante può concernere la tipizzazione delle variabili. Nei linguaggi "tipizzati" (ad esempio C), l'utente dichiara che userà quella etichetta (la variabile) per contenere solo ed esclusivamente un certo tipo di oggetto (ad esempio, un numero intero), e la variabile non potrà essere utilizzata per oggetti diversi (ad esempio, una stringa). In questo caso, prima di usare una variabile se ne dichiara l'esistenza e se ne specifica il tipo. I linguaggi non tipizzati non richiedono all'utente di specificare il tipo, che viene inferito in vario modo (ad esempio, in funzione dell'assegnazione del valore alla variabile). Alcuni linguaggi (ad esempio Python) non richiedono neppure la dichiarazione della variabile, che viene semplicemente usata. È l'interprete che inferisce che quella stringa è una variabile. La tipizzazione impone vincoli d'uso sulle variabili e maggiore scrittura del codice, ma assicura una chiara organizzazione dei dati. In assenza di tipizzazione, si lavora in maniera più rapida e snella, ma potenzialmente si può andare incontro a situazioni complicate, come quando si cambia il tipo di una variabile "in corsa" senza accorgersene" (Andrea Valle).*

`R` è un linguaggio non tipicizzato, come Python. In `R` non è necessario

dichiarare le variabili che si intendono utilizzare, né il loro tipo.

P.3.5 R console

La console di RStudio fornisce la possibilità di richiamare e rieseguire i comandi. I tasti freccia verticale, ↑ e ↓, sulla tastiera possono essere utilizzati per scorrere avanti e indietro i comandi già immessi. Appena trovato il comando che interessa, lo si può modificare, ad esempio, con i tasti freccia orizzontali, immettendo nuovi caratteri o cancellandone altri.

Se viene digitato un comando che `R` non riconosce, sulla console viene visualizzato un messaggio di errore; ad esempio,

```
3 % 9
Errore: unexpected input in "3 % 9"
```

P.3.6 Parentesi

Le parentesi in `R` (come in generale in ogni linguaggio di programmazione) assegnano un significato diverso alle porzioni di codice che delimitano.

- Le parentesi tonde funzionano come nell'algebra. Per esempio

```
2 + 3 * 4
#> [1] 14
```

non è equivalente a

```
(2 + 3) * 4
#> [1] 20
```

Le due istruzioni precedenti producono risultati diversi perché, se la sequenza delle operazioni algebriche non viene specificata dalle parentesi, `R` assegna alle operazioni algebriche il seguente ordine di priorità decrescente: esponenziazione, moltiplicazione / divisione, addizione / sottrazione, confronti logici (`<, >, <=, >=, ==, !=`). È sempre una buona idea rendere esplicito l'ordine delle operazioni algebriche che si vuole eseguire mediante l'uso delle parentesi tonde.

Le parentesi tonde vengono anche utilizzate per le funzioni, come vedremo nei prossimi paragrafi. Tra le parentesi tonde avremo dunque l'oggetto a cui vogliamo applicare la funzione e gli argomenti passati alla funzione.

- Le parentesi graffe sono destinate alla programmazione. Un blocco tra le parentesi graffe viene letto come un oggetto unico che può contenere una o più istruzioni.
- Le parentesi quadre vengono utilizzate per selezionare degli elementi, per esempio all'interno di un vettore, o di una matrice, o di un data.frame. L'argomento entro le parentesi quadre può essere generato da espressioni logiche.

P.3.7 I nomi degli oggetti

Le entità create e manipolate da `R` si chiamano 'oggetti'. Tali oggetti possono essere variabili (come nell'esempio che abbiamo visto sopra), array di numeri, caratteri, stringhe, funzioni, o più in generale strutture costruite a partire da tali componenti. Durante una sessione di R gli oggetti sono creati e memorizzati attraverso opportuni nomi.

I nomi possono contenere un qualunque carattere alfanumerico e come carattere speciale il trattino basso (`_`) o il punto. R fornisce i seguenti vincoli per i nomi degli oggetti: i nomi degli oggetti non possono mai iniziare con un carattere numerico e non possono contenere i seguenti simboli: `$`, `@`, `!`, `^`, `+`, `-`, `/`, `*`. È buona pratica usare nomi come `ratio_of_sums`. È fortemente sconsigliato utilizzare nei nomi degli oggetti caratteri accentati o, ancora peggio, apostrofi. Per questa ragione è sensato creare i nomi degli oggetti utilizzando la lingua inglese. È anche bene che i nomi degli oggetti non coincidano con nomi di funzioni. Ricordo nuovamente che `R` è *case sensitive*, cioè `A` e `a` sono due simboli diversi e identificano due oggetti differenti.

In questo corso cercheremo di evitare i numeri nei nomi degli oggetti `R`, così come le lettere maiuscole e il punto. Useremo quindi nomi come: `my_data`, `regression_results`, `square_root`, ecc.

P.3.8 Permanenza dei dati e rimozione di oggetti

Gli oggetti vengono salvati nello "spazio di lavoro" (*workspace*). Il comando `ls()` può essere utilizzato per visualizzare i nomi degli oggetti

che sono in quel momento memorizzati in R.

Per eliminare oggetti dallo spazio di lavoro è disponibile la funzione `rm()`; ad esempio

```
rm(x, y, z, ink, junk, temp, foo, bar)
```

cancella tutti gli oggetti indicati entro parentesi. Per eliminare tutti gli oggetti presenti nello spazio di lavoro si può utilizzare la seguente istruzione:

```
rm(list = ls())
```

P.3.9 Chiudere R

Quando si chiude RStudio il programma ci chiederà se si desidera salvare l'area di lavoro sul computer. Tale operazione è da evitare in quanto gli oggetti così salvati andranno ad interferire con gli oggetti creati in un lavoro futuro. Si consiglia dunque di rispondere negativamente a questa domanda.

- In RStudio, selezionare `Preferences` dal menu a tendina e, in `R General Workspace`, deselezionare l'opzione `Restore .RData into workspace at start- up` e scegliere l'opzione `Never` nella finestra di dialogo `Save workspace to .RData on exit`.
- In R, selezionare `Preferences` dal menu a tendina e, in `Startup`, selezionare l'opzione `No` in corrispondenza dell'item `Save workspace on exit from R`.

P.3.10 Creare ed eseguire uno script R con un editore

È molto più facile interagire con R manipolando uno script con un editore piuttosto che inserendo direttamente le istruzioni nella console. R fornisce il Text Editor dove è possibile inserire il codice (File → New Script). Per salvare il file basta utilizzare l'apposito menù a tendina (estensione `.R`). Tale file potrà poi essere riaperto ed utilizzato in un momento successivo.

L'editore comunica con R nel modo seguente: dopo avere selezionato la porzione di codice che si vuole eseguire, si digita un'apposita sequenza di tasti (`Command + Enter` su Mac OS X e `ctrl + r` in Windows). `ctrl +`

`r` significa premere il tasto `ctrl` e, tenendolo premuto, premere il tasto `r` della tastiera. Così facendo, `R` eseguirà le istruzioni selezionate e l'output verrà stampato sulla console. Il Text Editor fornito da `R` è piuttosto primitivo: è fortemente consigliato utilizzare RStudio.

P.3.11 Commentare il codice

Un "commento" è una parte di codice che l'interprete non tiene in considerazione. Quando l'interprete arriva ad un segnalatore di commento salta fino al segnalatore di fine commento e di lì riprende il normale processo esecutivo.

I commenti sono parole in linguaggio naturale (nel nostro caso l'italiano), che permettono agli utilizzatori di capire il flusso logico del codice e a chi lo ha scritto di ricordare il perché di determinate istruzioni.

In `R`, le parole dopo il simbolo `#` sono considerate commenti e sono ignorate; ad esempio:

```
# Questo e' un commento
```

P.3.12 Cambiare la cartella di lavoro

Quando si inizia una sessione di lavoro, `R` sceglie una cartella quale "working directory". Sarà in tale cartella che andrà a cercare gli script definiti dall'utilizzatore e i file dei dati. È possibile determinare quale sia la corrente "working directory" digitando sulla console di RStudio l'istruzione:

```
getwd()
```

Per cambiare la cartella di lavoro (in maniera tale che corrisponda alla cartella nella quale sono stati salvati i dati e gli script da eseguire) si sceglie la voce `Set Working Directory` sul menù a tendina di RStudio e si selezione la voce `Choose Directory…` Nella finestra che compare, si cambia la cartella con quella che si vuole.

P.3.13 L'oggetto base di R: il vettore

R opera su strutture di dati; la più semplice di tali strutture è il vettore numerico, che consiste in un insieme ordinato di numeri; ad esempio:

```
x <- c(7.0, 10.2, -2.9, 21.4)
```

Nell'istruzione precedente, `c()` è una funzione. In R gli argomenti sono passati alle funzioni inserendoli all'interno delle parentesi tonde. Si noti che gli argomenti (in questo caso, i numeri $7.0, 10.2, -2.9, 21.4$) sono separati a virgole. La funzione `c()` può prendere un numero arbitrario di argomenti e genera un vettore concatenando i suoi argomenti. L'operatore `<-` assegna un nome al vettore che è stato creato. Nel caso presente, digitando `x` possiamo visualizzare il vettore che abbiamo creato:

```
x
#> [1]  7.0 10.2 -2.9 21.4
```

Se invece eseguiamo l'istruzione

```
c(7.0, 10.2, -2.9, 21.4)
#> [1]  7.0 10.2 -2.9 21.4
```

senza assegnazione, il valore dell'espressione sarà visualizzato nella console, ma il vettore non potrà essere utilizzato in nessun altro modo.

P.3.14 Operazioni vettorializzate

Molte operazioni in R sono vettorializzate, il che significa che esse sono eseguite in parallelo in determinati oggetti. Ciò consente di scrivere codice che sia efficiente, conciso e più facile da leggere rispetto al codice che contiene istruzioni non vettorializzate.

P.3.15 Vettori aritmetici

L'esempio più semplice che illustra come si svolgono le operazioni vettorializzate riguarda le operazioni algebriche applicate ai vettori. I vettori, infatti, possono essere utilizzati in espressioni numeriche nelle quali le operazioni algebriche vengono eseguite "elemento per elemento".

Per illustrare questo concetto, definiamo il vettore `die` che contiene i possibili risultati del lancio di un dado:

```
die <- c(1, 2, 3, 4, 5, 6)
die
#> [1] 1 2 3 4 5 6
```

Supponiamo di volere sommare 10 a ciascun elemento del vettore `die`. Dato che le operazioni sui vettori sono eseguite elemento per elemento, per ottenere questo risultato è sufficiente eseguire l'istruzione:

```
die + 10
#> [1] 11 12 13 14 15 16
```

Si noti come la costante 10 sia stata sommata a ciascun elemento del vettore. In maniera corrispondente, l'istruzione

```
die - 1
#> [1] 0 1 2 3 4 5
```

sottrarrà un'unità da ciascuno degli elementi del vettore `die`.

Se l'operazione aritmetica coinvolge due o più vettori, R allinea i vettori ed esegue una sequenza di operazioni elemento per elemento. Per esempio, l'istruzione

```
die * die
#> [1]  1  4  9 16 25 36
```

fa sì che i due vettori vengano disposti l'uno di fianco all'altro per poi moltiplicare gli elementi corrispondenti: il primo elemento del primo vettore per il primo elemento del secondo vettore e così via. Il vettore risultante avrà la stessa dimensione dei due vettori che sono stati moltiplicati, come indicato qui sotto:

1	×	1	→	1
2	×	2	→	4
3	×	3	→	9
4	×	4	→	16
5	×	5	→	25
6	×	6	→	36
`die`	∗	`die`	=	

Oltre agli operatori aritmetici elementari `+`, `-`, `*`, `/`, e `^` per l'elevamento a potenza, sono disponibili le più comuni funzioni matematiche: `log()`, `exp()`, `sin()`, `cos()`, `tan()`, `sqrt()`, `max()`, `min()` e così via. Altre funzioni di uso comune sono: `range()` che restituisce un vettore `c(min(x), max(x))`; `sort()` che restituisce un vettore ordinato; `length(x)` che restituisce il numero di elementi di `x`; `sum(x)` che dà la somma degli elementi di `x`, mentre `prod(x)` dà il loro prodotto. Due funzioni statistiche di uso comune sono `mean(x)`, la media aritmetica, e `var(x)`, la varianza.

P.3.16 Generazione di sequenze regolari

`R` possiede un ampio numero di funzioni per generare sequenze di numeri. Ad esempio, `c(1:10)` è il vettore `c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`. L'espressione `c(30:1)` può essere utilizzata per generare una sequenza all'indietro.

La funzione `seq()` genera un vettore che contiene una sequenza regolare di numeri, generata in base a determinate regole. Può avere 5 argomenti: i primi due rappresentano l'inizio (`from`) e la fine (`to`) della sequenza, il terzo specifica l'ampiezza del passo (`by`), il quarto la lunghezza della sequenza (`length.out`) e infine il quinto (`along.with`), che se utilizzato deve essere l'unico parametro presente, è il nome di un vettore, ad esempio `x`, creando in tal modo la sequenza 1, 2, ..., `length(x)`. Esempi di utilizzo della funzione `seq()` sono i seguenti:

```
seq(from = 1, to = 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
seq(-5, 5, by = 2.5)
#> [1] -5.0 -2.5  0.0  2.5  5.0
seq(from = 1, to = 7, length.out = 4)
#> [1] 1 3 5 7
```

```
seq(along.with = die)
#> [1] 1 2 3 4 5 6
```

Altra funzione utilizzata per generare sequenze è `rep()` che può essere utilizzata per replicare un oggetto in vari modi. Ad esempio:

```
die3 <- rep(die, times = 3)
die3
#>  [1] 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6
```

metterà tre copie di `die` nell'oggetto `die3`.

P.3.17 Generazione di numeri casuali

La funzione `sample()` è una delle tante funzioni che possono essere usate per generare numeri casuali. Per esempio, la seguente istruzione simula dieci lanci di un dado a sei facce:

```
roll <- sample(1:6, 10, replace = TRUE)
roll
#>  [1] 1 3 3 6 3 1 2 2 6 6
```

Il primo argomento di `sample()` è il vettore da cui la funzione estrarrà degli elementi a caso; il secondo argomento specifica che dovranno essere effettuate 10 estrazioni casuali; il terzo argomento specifica che le estrazioni sono con rimessa (cioè, lo stesso elemento può essere estratto più di una volta).

Scegliere un elemento a caso dal vettore $\{1, 2, 3, 4, 5, 6\}$ è equivalente a lanciare un dado e osservare la faccia che si presenta. L'istruzione precedente corrisponde dunque alla simulazione di dieci lanci di un dado a sei facce.

P.3.18 Vettori logici

Quando si manipolano i vettori, talvolta si vogliono trovare gli elementi che soddisfano determinate condizioni logiche. Per esempio, in dieci lanci di un dado, quante volte è uscito 5? Per rispondere a questa domanda si

possono usare gli operatori logici `<`, `>` e `==` per le operazioni di "minore di," "maggiore di" e "uguale a". Se scriviamo

```
roll == 5
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [9] FALSE FALSE
```

creiamo un vettore costituito da elementi `TRUE/FALSE` i quali identificano gli elementi del vettore che soddisfano la condizione logica specificata.

Possiamo trattare tale vettore come se fosse costituito da elementi di valore 0 e 1. Sommando gli elementi di tale vettore, infatti, possiamo contare il numero di "5":

```
sum(roll == 5)
#> [1] 0
```

P.3.19 Dati mancanti

Quando si è in presenza di un dato mancante, R assegna il valore speciale `NA`, che sta per *Not Available*. In generale, un'operazione su un `NA` dà come risultato un `NA`. Nell'uso delle funzioni che operano sui dati sarà dunque necessario specificare che, qualunque operazione venga effettuata, gli `NA` devono essere esclusi.

P.3.20 Vettori di caratteri e fattori

I vettori di caratteri si creano formando una sequenza di caratteri delimitati da doppie virgolette e possono essere concatenati in un vettore attraverso la funzione `c()`. Successivamente, si può applicare la funzione `factor()`, che definisce automaticamente le modalità della variabile categoriale. Ad esempio,

```
soc_status <- factor(
  c("low", "high", "medium", "high", "low", "medium", "high")
)
levels(soc_status)
#> [1] "high"   "low"    "medium"
```

Talvolta l'ordine dei livelli del fattore non importa, mentre altre volte l'ordine è importante, per esempio, quando una variable categoriale viene rappresentata in un grafico. Per specificare l'ordine dei livelli del fattore si usa la seguente sintassi:

```
soc_status <-
  factor(soc_status, levels = c("low", "medium", "high"))
levels(soc_status)
#> [1] "low"    "medium" "high"
```

P.3.21 Funzioni

R offre la possibilità di utilizzare un'enorme libreria di funzioni che permettono di svolgere operazioni complicate, quali ad esempio, il campionamento casuale. Esaminiamo ora con più attenzione le proprietà delle funzioni di R utilizzando ancora l'esempio del lancio di un dado. Abbiamo visto in precedenza come il lancio di un dado possa essere simulato da R con la funzione `sample()`. La funzione `sample()` prende tre argomenti: il nome di un vettore, un numero chiamato `size` e un argomento chiamato `replace`. La funzione `sample()` ritorna un numero di elementi del vettore pari a `size`. Ad esempio

```
sample(die, 2, replace = TRUE)
#> [1] 3 4
```

Assegnando `TRUE` all'argomento `replace` specifichiamo che vogliamo un campionamento con rimessa.

Se volgiamo eseguire una serie di lanci indipendenti di un dado, eseguiamo ripetutamente la funzione `sample()` ponendo `size` uguale a 1:

```
sample(die, 1, replace = TRUE)
#> [1] 3
sample(die, 1, replace = TRUE)
#> [1] 2
sample(die, 1, replace = TRUE)
#> [1] 5
```

Come si fa a sapere quanti e quali argomenti sono richiesti da una funzione? Tale informazione viene fornita dalla funzione `args()`. Nel nostro caso

```
args(sample)
#> function (x, size, replace = FALSE, prob = NULL)
#> NULL
```

ci informa che il primo argomento è un vettore chiamato `x`, il secondo argomento è chiamato `size` ed ha il significato descritto sopra, il terzo argomento, `replace`, specifica se il campionamento è eseguito con o senza reimmissione, e il quarto argomento, `prob`, assegna delle probabilità agli elementi del vettore. Il significato degli argomenti viene spiegato nel file di help della funzione. Si noti che agli ultimi due argomenti sono stati assegnati dei valori, detti di default. Ciò significa che, se l'utilizzatore non li cambia, verranno usati da . La specificazione `replace = FALSE` significa che il campionamento viene eseguito senza reimmissione. Se desideriamo un campionamento con reimmissione, basta specificare `replace = TRUE` (nel caso di una singola estrazione è ovviamente irrilevante). Ad esempio, l'istruzione seguente simula i risultati di 10 lanci indipendenti di un dado:

```
sample(die, 10, replace = TRUE)
#>  [1] 3 6 1 3 3 4 3 6 3 4
```

Infine, `prob = NULL` specifica che non viene alterata la probabilità di estrazione degli elementi del vettore. In generale, gli argomenti di una funzione possono essere oggetti come vettori, matrici, altre funzioni, parametri o operatori logici.

`R` ha un sistema di help interno in formato HTML che si richiama con `help.start()`. Per avere informazioni su qualche funzione specifica, per esempio la funzione `sample()`, il comando da utilizzare è `help(sample)` oppure `?sample`.

P.3.22 Scrivere proprie funzioni

Abbiamo visto in precedenza come sia possibile simulare i risultati prodotti da dieci lanci di un dado o, in maniera equivalente, dal singolo lancio di dieci dadi. Possiamo replicare questo processo digitando ripetutamente le stesse istruzioni nella console. Otterremo ogni volta risultati

diversi perché, ad ogni ripetizione, il generatore di numeri pseudo-casuali di R dipende dal valore ottenuto dal clock interno della macchina. La funzione `set.seed()` ci permette di replicare esattamente i risultati della generazione di numeri casuali. Per ottenere questo risultato, basta assegnare al seed un numero arbitrario, es. `set.seed(12345)`. Tuttavia, questa procedura è praticamente difficile da perseguire se il numero di ripetizioni è alto. In tal caso è vantaggioso scrivere una funzione contenente il codice che specifica il numero di ripetizioni. In questo modo, per trovare il risultato cercato basterà chiamare la funzione una sola volta.

Le funzioni utilizzate da `R` sono costituite da tre elementi: il nome, il blocco del codice e una serie di argomenti. Per creare una funzione è necessario immagazzinare in R questi tre elementi e `function()` consente di ottenere tale risultato usando la sintassi seguente:

```
nome_funzione <- function(arg1, arg2, ...) {
  espressione1
  espressione2
  return(risultato)
}
```

Una chiamata di funzione è poi eseguita nel seguente modo:

```
nome_funzione(arg1, arg2, ...)
```

Per potere essere utilizzata, una funzione deve essere presente nella memoria di lavoro di `R`. Le funzioni salvate in un file possono essere richiamate utilizzando la funzione `source()`, ad esempio, `source("file_funzioni.R")`.

Consideriamo ora la funzione `two_rolls()` che ritorna la somma dei punti prodotti dal lancio di due dadi non truccati:

```
two_rolls <- function() {
  die <- 1:6
  res <- sample(die, size = 2, replace = TRUE)
  sum_res <- sum(res)
  return(sum_res)
}
```

La funzione `two_rolls()` inizia con il creare il vettore `die` che contiene sei elementi: i numeri da 1 a 6. Viene poi utilizzata la funzione `sample()` con gli gli argomenti, `die`, `size = 2` e `replace = TRUE`. Tale funzione restituisce il risultato del lancio di due dadi. Il risultato fornito da `sample(die, size = 2, replace = TRUE)` viene assegnato all'oggetto `res`. L'oggetto `res` corrisponde dunque ad un vettore di due elementi. L'istruzione `sum(res)` somma gli elementi del vettore `res` e attribuisce il risultato di questa operazione a `sum_res`. Infine, la funzione `return()` ritorna il contenuto dell'oggetto `sum_res`. Invocando la funzione `two_rolls()` si ottiene dunque la somma del lancio di due dadi. In generale, la funzione `two_rolls()` produrrà un risultato diverso ogni volta che viene usata:

```
two_rolls()
#> [1] 9
two_rolls()
#> [1] 4
two_rolls()
#> [1] 7
```

La formattazione del codice mediante l'uso di spazi e rientri non è necessaria ma è altamente raccomandata per minimizzare la probabilità di compiere errori.

P.3.23 Pacchetti

Le funzioni di `R` sono organizzate in pacchetti, i più importanti dei quali sono già disponibili quando si accede al programma.

P.3.24 Istallazione e upgrade dei pacchetti

Alcuni pacchetti non sono presenti nella release di base di `R`. Per installare un pacchetto non presente è sufficiente scrivere nella console:

```
install.packages("nome_pacchetto")
```

Ad esempio,

```
install.packages("ggplot2")
```

La prima volta che si usa questa funzione durante una sessione di lavoro si dovrà anche selezionare da una lista il sito *mirror* da cui scaricare il pacchetto.

Gli autori dei pacchetti periodicamente rilasciano nuove versioni dei loro pacchetti che contengono miglioramenti di varia natura. Per eseguire l'upgrade dei pacchetti `ggplot2` e `dplyr`, ad esempio, si usa la seguente istruzione:

```
update.packages(c("ggplot2", "dplyr"))
```

Per eseguire l'upgrade di tutti i pacchetti l'istruzione è

```
update.packages()
```

P.3.25 Caricare un pacchetto in R

L'istallazione dei pacchetti non rende immediatamente disponibili le funzioni in essi contenute. L'istallazione di un pacchetto semplicemente copia il codice sul disco rigido della macchina in uso. Per potere usare le funzioni contenute in un pacchetto installato è necessario caricare il pacchetto in . Ciò si ottiene con il comando:

```
library("ggplot2")
```

se si vuole caricare il pacchetto `ggplot2`. A questo punto diventa possibile usare le funzioni contenute in `ggplot2`. Queste operazioni si possono anche eseguire usando dal menu a tendina di RStudio.

Per sapere quali sono i pacchetti già presenti nella release di `R` con cui si sta lavorando, basta scrivere:

```
library()
```

P.4 Strutture di dati

Solitamente gli psicologi raccolgono grandi quantità di dati. Tali dati vengono codificati in R all'interno di oggetti aventi proprietà diverse. Intuitivamente, in R un oggetto è qualsiasi cosa a cui è possibile assegnare un valore. I dati possono essere di tipo numerico o alfanumerico. Di conseguenza, Rdistingue tra oggetti aventi *modi* diversi. Inoltre, i dati possono essere organizzati in righe e colonne in base a diversi tipi di strutture che R chiama *classi*.

P.4.1 Classi e modi degli oggetti

Gli oggetti R si distinguono a seconda della loro classe (*class*) e del loro modo (*mode*). La classe definisce il tipo di oggetto. In R, vengono utilizzate cinque strutture di dati che corrispondono a cinque classi differenti: `vector`, `matrix`, `array`, `list` e `data.frame`. Un'altra classe di oggetti R è `function` (ad essa appartengono le funzioni).

La classe di appartenenza di un oggetto si stabilisce usando le funzioni `class()`, oppure `is.list()`, `is.function()`, `is.logical()`, e così via. Queste funzioni restituisco `TRUE` e `FALSE` in base all'appartenenza o meno dell'argomento a quella determinata classe.

Gli oggetti R possono anche essere classificati in base al loro 'modo'. I modi 'atomici' degli oggetti sono: `numeric`, `complex`, `character` e `logical`. Per esempio,

```
x <- c(4, 9)
mode(x)
#> [1] "numeric"
cards <- c("9 of clubs", "10 of hearts", "jack of hearts")
mode(cards)
#> [1] "character"
```

Nel seguito verranno esaminate le cinque strutture di dati utilizzate da R.

P.4.2 Vettori

I vettori sono la classe di oggetto più importante in R. Un vettore può essere creato usando la funzione `c()`:

```
y <- c(2, 1, 6, -3, 9)
y
#> [1]  2  1  6 -3  9
```

Le dimensioni di un vettore presente nella memoria di lavoro possono essere trovare con la funzione `length()`; ad esempio,

```
length(y)
#> [1] 5
```

ci dice che `y` è un vettore costituito da cinque elementi. La somma, il minimo e il massimo degli elementi contenuti in un vettore si trovano con le seguenti istruzioni:

```
sum(y)
#> [1] 15
min(y)
#> [1] -3
max(y)
#> [1] 9
```

Mentre ci sono sei 'tipi' di vettori 'atomici' in R, noi ci focalizzeremo sui tipi seguenti: 'numeric' ('integer': *e.g.*, 5; 'double': *e.g.*, 5.5), 'character' (*e.g.*, 'pippo') e 'logical' (*e.g.*, `TRUE`, `FALSE`). Usiamo la funzione `typeof()` per determinare il 'tipo' di un vettore atomico. Tutti gli elementi di un vettore atomico devono essere dello stesso tipo. La funzione `str()` rende visibile in maniera compatta la struttura interna di un oggetto.

P.4.3 Matrici

Una matrice è una collezione di vettori. Il comando per generare una matrice è `matrix()`:

```
X <- matrix(1:20, nrow = 4, byrow = FALSE)
X
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    5    9   13   17
#> [2,]    2    6   10   14   18
#> [3,]    3    7   11   15   19
#> [4,]    4    8   12   16   20
```

Il primo argomento è il vettore i cui elementi andranno a disporsi all'interno della matrice. È poi necessario specificare le dimensioni della matrice e il modo in cui `R` dovrà riempire la matrice. Date le dimensioni del vettore, la specificazione del numero di righe (secondo argomento) è sufficiente per determinare le dimensioni della matrice. L'argomento `byrow = FALSE` è il default. In tal caso, `R` riempie la matrice per colonne. Se vogliamo che `R` riempia la matrice per righe, usiamo `byrow = TRUE`:

```
Y <- matrix(1:20, nrow = 4, byrow = TRUE)
Y
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
#> [2,]    6    7    8    9   10
#> [3,]   11   12   13   14   15
#> [4,]   16   17   18   19   20
```

Le dimensioni di una matrice presente nella memoria di lavoro possono essere trovare con la funzione `dim()`; ad esempio,

```
dim(Y)
#> [1] 4 5
```

ci dice che `Y` è una matrice con quattro righe e cinque colonne.

P.4.4 Array

Un array è una collezione di matrici (si veda la Figura 1.1). Per costruire un array con la funzione `array()` è necessario specificare un vettore come primo argomento e un vettore di dimensioni, chiamato `dim`, quale secondo argomento:

```
ar <- array(
  c(11:14, 21:24, 31:34),
  dim = c(2, 2, 3)
)
```

Un sottoinsieme di questi dati può essere selezionato, per esempio, nel modo seguente:

```
ar[, , 3]
#>      [,1] [,2]
#> [1,]   31   33
#> [2,]   32   34
```

P.4.5 Operazioni aritmetiche su vettori, matrici e array

P.4.5.1 Operazioni aritmetiche su vettori

I vettori e le matrici (o gli array) possono essere utilizzati in espressioni aritmetiche. Il risultato è un vettore o una matrice (o un array) formato dalle operazioni fatte elemento per elemento sui vettori o sulle matrici. Ad esempio,

```
y + 3
#> [1]  5  4  9  0 12
```

restituisce un vettore di dimensioni uguali alle dimensioni di `y`, i cui elementi sono dati dalla somma tra ciascuno degli elementi originari di `y` e la costante "3".

Ovviamente, ad un vettore possono essere applicate tutte le altre operazioni algebriche, sempre elemento per elemento. Ad esempio,

```
3 * y
#> [1]  6  3 18 -9 27
```

restituisce un vettore i cui elementi sono uguali agli elementi di `y` moltiplicati per 3.

Se sono costituiti dallo stesso numero di elementi, due vettori possono essere sommati, sottratti, moltiplicati e divisi, laddove queste operazioni algebriche vengono eseguite elemento per elemento. Per esempio,

```
x <- c(1, 1, 2, 1, 3)
y <- c(2, 1, 6, 3, 9)
x + y
#> [1]  3  2  8  4 12
x - y
#> [1] -1  0 -4 -2 -6
x * y
#> [1]  2  1 12  3 27
x / y
#> [1] 0.5000 1.0000 0.3333 0.3333 0.3333
```

P.4.5.2 Operazioni aritmetiche su matrici

Le operazioni algebriche elemento per elemento si possono estendere al caso delle matrici. Per esempio, se `X`, `Y` sono entrambe matrici di dimensioni 4×5, allora la seguente operazione

```
M <- 2 * (X + Y) - 3
```

crea una matrice `D` anch'essa di dimensioni 4×5 i cui elementi sono ottenuti dalle operazioni fatte elemento per elemento sulle matrici e sugli scalari:

```
M
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1   11   21   31   41
#> [2,]   13   23   33   43   53
```

```
#> [3,]   25   35   45   55   65
#> [4,]   37   47   57   67   77
```

P.4.5.3 Operazioni aritmetiche su array

Le stesse considerazioni si estendono al caso degli array.

P.4.6 Liste

Le liste assomigliano ai vettori perché raggruppano i dati in un insieme unidimensionale. Tuttavia, le liste non raggruppano elementi individuali ma bensì oggetti di `R`, quali vettori e altre liste. Per esempio,

```
list1 <- list("R", list(TRUE, FALSE), 20:24)
list1
#> [[1]]
#> [1] "R"
#>
#> [[2]]
#> [[2]][[1]]
#> [1] TRUE
#>
#> [[2]][[2]]
#> [1] FALSE
#>
#>
#> [[3]]
#> [1] 20 21 22 23 24
```

Le doppie parentesi quadre identificano l'elemento della lista a cui vogliamo fare riferimento. Per esempio,

```
list1[[3]]
#> [1] 20 21 22 23 24
list1[[3]][2]
#> [1] 21
```

P.4.7 Data frame

I data.frame sono strutture tipo matrice, in cui le colonne possono essere vettori di tipi differenti. La funzione usata per generare un data frame è `data.frame()`, che permette di unire più vettori di uguale lunghezza come colonne del data frame, ognuno dei quali si riferisce ad una diversa variabile. Ad esempio,

```
df <- data.frame(
  face = c("ace", "two", "six"),
  suit = c("clubs", "clubs", "clubs"),
  value = c(1, 2, 3)
)
df
#>   face  suit value
#> 1  ace clubs     1
#> 2  two clubs     2
#> 3  six clubs     3
```

L'estrazione di dati da un data.frame può essere effettuata in maniera simile a quanto avviene per i vettori. Ad esempio, per estrarre la variabile `value` dal data.frame `df` si può indicare l'indice della terza colonna:

```
df[, 3]
#> [1] 1 2 3
```

Dal momento che le colonne sono delle variabili, è possibile estrarle anche indicando nome della variabile, scrivendo `nome_data_frame$nome_variabile`:

```
df$value
#> [1] 1 2 3
```

Per fare un esempio, creiamo un data.frame che contenga tutte le informazioni di un mazzo di carte da poker (Grolemund, 2014). In tale data.frame, ciascuna riga corrisponde ad una carta – in un mazzo da poker ci sono 52 carte, perciò il data.frame avrà 52 righe. Il vettore `face` indica con una stringa di caratteri il valore di ciascuna carta, il vettore `suit` indica il seme e il vettore `value` indica con un numero intero il valore di ciascuna carta. Quindi, il data.frame avrà 3 colonne.

```
deck <- data.frame(
  face = c("king", "queen", "jack", "ten", "nine", "eight",
  "seven", "six", "five", "four", "three", "two", "ace",
  "king", "queen", "jack", "ten", "nine", "eight", "seven",
  "six", "five", "four", "three", "two", "ace", "king",
  "queen", "jack", "ten", "nine", "eight", "seven", "six",
  "five", "four", "three", "two", "ace", "king", "queen",
  "jack", "ten", "nine", "eight", "seven", "six", "five",
  "four", "three", "two", "ace"),
  suit = c("spades", "spades", "spades", "spades",
  "spades", "spades", "spades", "spades", "spades",
  "spades", "spades", "spades", "spades", "clubs", "clubs",
  "clubs", "clubs", "clubs", "clubs", "clubs", "clubs",
  "clubs", "clubs", "clubs", "clubs", "clubs", "diamonds",
  "diamonds", "diamonds", "diamonds", "diamonds",
  "diamonds", "diamonds", "diamonds", "diamonds",
  "diamonds", "diamonds", "diamonds", "diamonds", "hearts",
  "hearts", "hearts", "hearts", "hearts", "hearts",
  "hearts", "hearts", "hearts", "hearts", "hearts",
  "hearts", "hearts"),
  value = c(13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
)
```

Avendo salvato tutte queste informazioni nell'oggetto `deck`, possiamo stamparle sullo schermo semplicemente digitando il nome dell'oggetto che le contiene:

```
deck
#>     face     suit value
#> 1   king   spades    13
#> 2  queen   spades    12
#> 3   jack   spades    11
#> 4    ten   spades    10
#> 5   nine   spades     9
#> 6  eight   spades     8
#> 7  seven   spades     7
#> 8    six   spades     6
#> 9   five   spades     5
```

```
#> 10  four   spades     4
#> 11 three   spades     3
#> 12   two   spades     2
#> 13   ace   spades     1
#> 14  king    clubs    13
#> 15 queen    clubs    12
#> 16  jack    clubs    11
#> 17   ten    clubs    10
#> 18  nine    clubs     9
#> 19 eight    clubs     8
#> 20 seven    clubs     7
#> 21   six    clubs     6
#> 22  five    clubs     5
#> 23  four    clubs     4
#> 24 three    clubs     3
#> 25   two    clubs     2
#> 26   ace    clubs     1
#> 27  king diamonds    13
#> 28 queen diamonds    12
#> 29  jack diamonds    11
#> 30   ten diamonds    10
#> 31  nine diamonds     9
#> 32 eight diamonds     8
#> 33 seven diamonds     7
#> 34   six diamonds     6
#> 35  five diamonds     5
#> 36  four diamonds     4
#> 37 three diamonds     3
#> 38   two diamonds     2
#> 39   ace diamonds     1
#> 40  king   hearts    13
#> 41 queen   hearts    12
#> 42  jack   hearts    11
#> 43   ten   hearts    10
#> 44  nine   hearts     9
#> 45 eight   hearts     8
#> 46 seven   hearts     7
#> 47   six   hearts     6
```

```
#> 48  five   hearts     5
#> 49  four   hearts     4
#> 50 three   hearts     3
#> 51   two   hearts     2
#> 52   ace   hearts     1
```

Si noti che, a schermo, R stampa un numero progressivo che corrisponde al numero della riga.

P.4.7.1 Selezione di elementi

Una volta creato un data.frame, ad esempio quello che contiene un mazzo virtuale di carte (si veda l'esempio

$$exmp : deck_of_cards$$

), è necessario sapere come manipolarlo. La funzione `head()` mostra le prime sei righe del data.frame:

```
head(deck)
#>    face   suit value
#> 1  king spades    13
#> 2 queen spades    12
#> 3  jack spades    11
#> 4   ten spades    10
#> 5  nine spades     9
#> 6 eight spades     8
```

Poniamoci ora il problema di mescolare il mazzo di carte e di estrarre alcune carte dal mazzo. Queste operazioni possono essere eseguite usando il sistema notazionale di R.

Il sistema di notazione di R consente di estrarre singoli elementi dagli oggetti definiti da R. Per estrarre un valore da un data.frame, per esempio, dobbiamo scrivere il nome del data.frame seguito da una coppia di parentesi quadre:

```
deck[, ]
```

All'interno delle parentesi quadre ci sono due indici separati da una virgola. R usa il primo indice per selezionare un sottoinsieme di righe del

data.frame e il secondo indice per selezionare un sottoinsieme di colonne. L'indice è il numero d'ordine che etichetta progressivamente ognuno dei valori del vettore. Per esempio,

```
deck[9, 2]
#> [1] "spades"
```

restituisce l'elemento che si trova nella nella nona riga della seconda colonna di `deck`.

In `R` ci sono sei modi diversi per specificare gli indici di un oggetto: interi positivi, interi negativi, zero, spazi vuoti, valori logici e nomi. Esaminiamoli qui di seguito.

P.4.7.2 Interi positivi

Gli indici i, j possono essere degli interi positivi che identificano l'elemento nella i-esima riga e nella j-esima colonna del data.frame. Per l'esempio relativo al mazzo di carte, l'istruzione

```
deck[1, 1]
#> [1] "king"
```

ritorna il valore nella prima riga e nella prima colonna. Per estrarre più di un valore, usiamo un vettore di interi positivi. Per esempio, la prima riga di `deck` si trova con

```
deck[1, c(1:3)]
#>   face   suit value
#> 1 king spades    13
```

Tale sistema notazionale non si applica solo ai data.frame ma può essere usato anche per gli altri oggetti di `R`.

L'indice usato da `R` inizia da 1. In altri linguaggi di programmazione, per esempio `C`, inizia da 0.

P.4.7.3 Interi negativi

Gli interi negativi fanno l'esatto contrario degli interi positivi: R ritornerà tutti gli elementi tranne quelli specificati dagli interi negativi. Per

esempio, la prima riga del data.frame può essere specificata nel modo seguente

```
deck[-(2:52), 1:3]
#>   face   suit value
#> 1 king spades    13
```

ovvero, escludendo tutte le righe seguenti.

P.4.7.4 Zero

Quando lo zero viene usato come indice, `R` non ritorna nulla dalla dimensione a cui lo zero si riferisce. L'istruzione

```
deck[0, 0]
#> data frame con 0 colonne e 0 righe
```

ritorna un data.frame vuoto. Non molto utile.

P.4.7.5 Spazio ' '

Uno spazio viene usato quale indice per comunicare a `R` di estrarre tutti i valori in quella dimensione. Questo è utile per estrarre intere colonne o intere righe da un data.frame. Per esempio, l'istruzione

```
deck[3, ]
#>   face   suit value
#> 3 jack spades    11
```

ritorna la terza riga del data.frame `deck`.

P.4.7.6 Valori booleani

Se viene fornito un vettore di stringhe `TRUE`, `FALSE`, `R` selezionerà gli elementi riga o colonna corrispondenti ai valori booleani `TRUE` usati quali indici. Per esempio, l'istruzione

```
deck[3, c(TRUE, TRUE, FALSE)]
#>   face   suit
#> 3 jack spades
```

ritorna i valori delle prime due colonne della terza riga di `deck`.

P.4.7.7 Nomi

È possibile selezionare gli elementi del data.frame usando i loro nomi. Per esempio,

```
deck[1, c("face", "suit", "value")]
#>   face   suit value
#> 1 king spades    13
deck[, "value"]
#>  [1] 13 12 11 10  9  8  7  6  5  4  3  2  1 13 12 11 10
#> [18]  9  8  7  6  5  4  3  2  1 13 12 11 10  9  8  7  6
#> [35]  5  4  3  2  1 13 12 11 10  9  8  7  6  5  4  3  2
#> [52]  1
```

P.4.8 Giochi di carte

Avendo presentato le nozioni base del sistema di notazione di `R`, utilizziamo tali conoscenze per manipolare il data.frame. L'istruzione

```
deck[1:52, ]
```

ritorna tutte le righe e tutte e le colonne del data.frame `deck`. Le righe sono identificate dal primo indice, che va da 1 a 52. Permutare in modo casuale l'indice delle righe equivale a mescolare il mazzo di carte. Per fare questo, utilizziamo la funzione `sample()` ponendo `replace=FALSE` e `size` uguale alla dimensione del vettore che contiene gli indici da 1 a 52:

```
random <- sample(1:52, size = 52, replace = FALSE)
random
#>  [1] 41 35 51 22  3 10  2 15 11 16 46 21  7 19 43 17 27
#> [18] 50 39 44 14 18 40 47 31 30 52 37 20 33  6  9  5 49
#> [35] 13  4  8 28 32 45 42 26 36  1 48 23 24 29 34 25 38
#> [52] 12
```

Utilizzando il vettore `random` di indici permutati otteniamo il risultato cercato:

```
deck_shuffled <- deck[random, ]
head(deck_shuffled)
#>     face     suit value
#> 41 queen   hearts    12
#> 35  five diamonds     5
#> 51   two   hearts     2
#> 22  five    clubs     5
#> 3   jack   spades    11
#> 10  four   spades     4
```

Possiamo ora scrivere una funzione che include le precedenti istruzioni:

```
shuffle <- function(cards) {
  random <- sample(1:52, size = 52, replace = FALSE)
  return(cards[random, ])
}
```

Invocando la funzione `shuffle()` possiamo generare un data.frame che rappresenta un mazzo di carte mescolato:

```
deck_shuffled <- shuffle(deck)
```

Se immaginiamo di distribuire le carte di questo mazzo a due giocatori di poker, per il primo giocatore avremo:

```
deck_shuffled[c(1, 3, 5, 7, 9), ]
#>    face     suit value
#> 26  ace    clubs     1
#> 44 nine   hearts     9
#> 29 jack diamonds    11
#> 42 jack   hearts    11
#> 22 five    clubs     5
```

e per il secondo:

```
deck_shuffled[c(2, 4, 6, 8, 10), ]
#>     face     suit value
```

```
#> 24 three    clubs     3
#> 47   six   hearts     6
#> 34   six diamonds     6
#> 51   two   hearts     2
#> 17   ten    clubs    10
```

P.4.9 Variabili locali

Si noti che, nell'esempio precedente, abbiamo passato l'argomento `deck` alla funzione `shuffle()`, perché questo è il nome del data.frame che volevamo manipolare. Nella definizione della funzione `shuffle()`, però, l'argomento della funzione era chiamato `cards`. Il nome degli argomenti è diverso nei due casi. Allora perché l'istruzione `shuffle(deck)` non dà un messaggio d'errore?

La risposta a questa domanda è che nelle funzioni le variabili nascono quando la funzione entra in esecuzione e muoiono al termine dell'esecuzione della funzione. Per questa ragione, sono dette 'locali'. La variabile `cards`, in questo esempio, esiste soltanto all'interno della funzione. Dunque non deve (necessariamente) avere lo stesso nome di un altro oggetto che esiste al di fuori della funzione, nello spazio di lavoro di R (anzi, è meglio se il nome degli oggetti usati all'interno delle funzioni è diverso da quello degli oggetti che esistono fuori dalle funzioni). R sa che l'oggetto `deck` passato a `shuffle()` corrisponde a `cards` all'interno della funzione perché assegna il nome `cards` a qualunque oggetto venga passato alla funzione `shuffle()` come primo (e, in questo caso, unico) argomento.

P.5 Strutture di controllo

In `R` il flusso della computazione segue l'ordine di lettura delle espressioni. I controlli di flusso sono quei costrutti sintattici che possono modificare quest'ordine di computazione. Ad esempio, un ciclo `for` ripete le istruzioni annidate al suo interno per un certo numero di volte, e quindi procede sequenzialmente da lì in avanti, mentre un condizionale `if` valuta una condizione rispetto alla quale il flusso di informazioni si biforca (se è vero / se è falso). Ci limitiamo qui ad introdurre il ciclo `for`.

P.5.1 Il ciclo `for`

Il ciclo `for` è una struttura di controllo iterativa che determina l'esecuzione di una porzione di codice ripetuta per un certo numero noto di volte. Il linguaggio `R` usa la seguente sintassi per il ciclo `for`:

`for` (`indice` in `valori_indice`) { *operazioni* }

il che significa "esegui le operazioni *operazioni* per i diversi valori di `indice` compresi nel vettore `valori_indice`". Per esempio, il seguente ciclo `for` non fa altro che stampare il valore della variabile contatore in ciascuna esecuzione del ciclo:

```
for (i in 1:3) {
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
```

Un esempio (leggermente) più complicato è il seguente:

```
x_list <- seq(1, 9, by = 2)
x_list
#> [1] 1 3 5 7 9
sum_x <- 0
for (x in x_list) {
  sum_x <- sum_x + x
  cat("L'indice corrente e'", x, "\n")
  cat("La frequenza cumulata e'", sum_x, "\n")
}
#> L'indice corrente e' 1
#> La frequenza cumulata e' 1
#> L'indice corrente e' 3
#> La frequenza cumulata e' 4
#> L'indice corrente e' 5
#> La frequenza cumulata e' 9
#> L'indice corrente e' 7
#> La frequenza cumulata e' 16
```

```
#> L'indice corrente e' 9
#> La frequenza cumulata e' 25
```

Per esempio, quanti numeri pari sono contenuti in un vettore? La risposta a questa domanda viene fornita dalla funzione `countEvenNumbers()` che possiamo definire come indicato qui sotto:

```
countEvenNumbers <- function(x) {
  count <- 0
  for (i in 1:length(x)) {
    if (x[i] %% 2 == 0)
      count = count + 1
  }
  count
}
```

Nella funzione `countEvenNumbers()` abbiamo inizializzato la variabile `count` a zero. Prima dell'esecuzione del ciclo `for`, dunque, `count` vale zero. Il ciclo `for` viene eseguito tante volte quanti sono gli elementi che costituiscono il vettore `x`. L'indice `i` dunque assume valori compresi tra 1 e il valore che corrisponde al numero di elementi di `x`. L'operazione modulo, indicato con `%%` dà come risultato il resto della divisione euclidea del primo numero per il secondo. Per esempio, `9 %% 2` dà come risultato 1 perché questo è il resto della divisione 9/2. L'operazione modulo dà come risultato 0 per tutti i numeri pari. In ciascuna esecuzione del ciclo `for` l'operazione modulo viene eseguita, successivamente, su uno degli elementi di `x`. Se l'operazione modulo dà 0 come risultato, ovvero se il valore considerato è un numero pari, allora la variabile `count` viene incrementata di un'unità. L'istruzione `return()` ritorna il numero di valori pari contenuti nel vettore di input alla funzione. Si noti che è necessario usare `return()`: la funzione ritornerà qualunque cosa sia stampato nell'ultima riga della funzione stessa.

Facciamo un esempio:

```
x <- c(1, 2, 1, 4, 6, 3, 9, 12)
countEvenNumbers(x)
#> [1] 4
```

P.6 Input/Output

I dati raccolti dallo psicologo sono contenuti in file aventi formati diversi: solo testo, CSV, Excel, eccetera. `R` prevede diverse funzioni di importazione dei dati. Esamineremo qui la funzione `read.table()` per l'importazione di dati in formato solo testo, ma funzioni analoghe possono essere usate per molti altri formati possibili.

P.6.1 La funzione `read.table()`

Ci sono tanti modi per importare un file dal nostro computer. `R` permette di utilizzare delle funzioni che sono già nella libreria di base, oppure possiamo utilizzare delle funzioni specifiche, a seconda del tipo di file da importare, che sono contenute in pacchetti aggiuntivi. Per leggere i dati da file in `R` è conveniente preliminarmente generare un file di dati in formato ASCII, disponendoli come si farebbe in una matrice di dati, e mettere questo file nella cartella di lavoro corrente. Fatto questo, si può utilizzare la funzione `read.table()` presente nella libreria di base per leggere l'intero dataset. Se la prima riga del file contiene l'intestazione delle variabili, allora `read.table("my_file.txt", header = TRUE)` interpreterà la prima riga del file come una riga dove sono contenuti i nomi delle variabili, assegnando ciascun nome alle variabili del data frame:

```
mydata <- read.table("my_file.txt", header = TRUE)
```

In alternativa, si può impiegare la funzione `read.csv()`, che è adatta a leggere dati salvati in `.csv`. Utilizzando altre funzioni, si possono leggere in `R` i dati contenuti in file aventi formati diversi da quelli considerati qui, quali Excel, SPSS, ecc.

P.6.2 File di dati forniti da R

In `R` esistono comunque oltre 50 insiemi di dati contenuti nel package `base` e altri sono disponibili in altri packages. Per vedere l'elenco degli insiemi di dati disponibili nel package `base` basta usare l'istruzione `data()`; per caricare un particolare insieme di dati, ad esempio `cars`, basta utilizzare l'istruzione

```
data(cars)
```

Nella maggior parte dei casi questo corrisponde a caricare un oggetto, solitamente un data.frame dello stesso nome: per l'esempio considerato si avrebbe un data frame di nome `cars`.

P.6.3 Esportazione di un file

Per esportare un data.frame in formato `.csv` possiamo scrivere il seguente codice

```
write.csv(df_esempio, file = "esempio.csv", row.names = FALSE)
```

dove `df_esempio` è il data.frame da salvare e `esempio.csv` è il file che verrà salvato all'interno della nostra cartellla di lavoro.

P.6.4 Pacchetto `rio`

Un'alternativa più semplice è fornita dalle funzioni fornite dal pacchetto `rio`. Per importare i dati da un file in qualsiasi formato si usa

```
my_data_frame <- rio::import("my_file.csv")
```

Per esportare i dati in un file avente qualsiasi formato si usa invece

```
rio::export(my_data_frame, "my_file.csv")
```

P.6.5 Dove sono i miei file?

Quello che abbiamo detto finora, a proposito dell'importazione ed esportazione dei file, si riferisce a file che si trovano nella cartella di lavoro (*working directory*). Ma non sempre ci troviamo in questa situazione, il che è una buona cosa, perché se dobbiamo gestire un progetto anche leggermente complesso è sempre una buona idea salvare i file che usiamo in cartelle diverse. Per esempio, possiamo usare una cartella chiamata `psicometria` dove salviamo tutto il materiale di questo insegnamento. Nella cartella `psicometria` ci potrà essere una cartella chiamata `scripts` dove salveremo gli script con il codice R utilizzato per i vari esercizi, e

una cartella chiamata `data` dove possiamo salvare i dati. Questa organizzazione minimale ci pone, però, difronte ad un problema: i dati che vogliamo caricare in R non si trovano nella cartella dove sono contenuti gli script. Quando importiamo un file di dati dobbiamo dunque specificare il percorso che identifica la posizione del file sul nostro computer.

Questo problema può essere risolto in due modi: speficicando l'indirizzo assoluto del file, o l'indirizzo relativo. Specificare l'indirizzo assoluto di un file comporta una serie di svantaggi. Il più grande è che non sarà possibile utilizzare quell'istruzione su una macchina diversa. Dunque, è molto più conveniente specificare l'indirizzo dei file in modo relativo. Ma relativo rispetto a cosa? Rispetto alla *working directory* che definirà l'origine del nostro percorso.

È ovvio che la *working directory* cambia da progetto a progetto. Infatti, per ciascun progetto dobbiamo specificare una diversa *working directory*. Per esempio, potremmo avere un progetto relativo all'insegnamento di Psicometria e un progetto relativo alla prova finale.

Per organizzaere il lavoro in questo modo, si procede come segue. Supponiamo di creare una cartella chiamata `psicometria` che contiene, al suo interno, le cartelle `scripts` e `data`:

```
psicometria/
  ├── data
  ├── scripts
```

Supponiamo che queste cartelle contengano i file che ho specificato sopra. Chiudiamo RStudio, se è aperto, e lo riapriamo di nuovo. Dal menu selezioniamo `File -> New Project...` In questo modo si aprirà un menu che ci chiederà, tra le altre cose, se vogliamo creare un nuovo progetto (`New project`). Selezioniamo quell'opzione e navighiamo fino alla cartella `psicometria` e selezioniamo `open`. Questo creerà un file chiamato `psicometria.Rproj` nella cartella `psicometria`.

Chiudiamo ora RStudio. Se vogliamo accedere al progetto "psicometria", che abbiamo appena creato, dobbiamo semplicemente cliccare sul file `psicometria.Rproj`. Questo aprirà RStudio e farà in modo che la *working directory* coincida con la cartalla `psicometria`. Ogni volta che vogliamo lavorare sui dati del progetto "psicometria" chiudiamo dunque RStudio (se è già aperto) e lo riapriamo cliccando sul file `psicometria.Rproj`.

A questo punto possiamo definire l'indirizzo dei file in modo relativo – ovvero, relativo alla cartella `psicometria`. Per fare questo usiamo le funzionalità del pacchetto `here`. Supponiamo di volere caricare un file di dati che si chiama `dati_depressione.txt` e si trova nella cartella `psicometria/data`. Per importare i dati (dopo avere caricato i pacchetti `rio` e `here`) useremo l'istruzione seguente:

```
rio::import(here("data", "dati_depressione.txt"))
```

In altre parole, così facendo specifichiamo il percorso relativo del file `dati_depressione.txt` (in quanto l'origine corrisponde alla cartella `psicometria`). L'istruzione precedente significa che, partendo dalla cartella che coincide con la *working directory* (ovvero, `psicometria`) ci spostiamo nella cartella `data` e lì dentro troviamo il file chiamato `dati_depressione.txt`.

P.7 Manipolazione dei dati

P.7.1 Motivazione

Si chiamano "dati grezzi" quelli che provengono dal mondo circostanze, i dati raccolti per mezzo degli strumenti usati negli esperimenti, per mezzo di interviste, di questionari, ecc. Questi dati (chiamati *dataset*) raramente vengono forniti con una struttura logica precisa. Per potere elaborarli mediante dei software dobbiamo prima trasformarli in maniera tale che abbiano una struttura logica organizzata. La struttura che solitamente si utilizza è quella tabellare (matrice dei dati), ovvero si dispongono i dati in una tabella nella quale a ciascuna riga corrisponde ad un'osservazione e ciascuna colonna corrisponde ad una variabile rilevata. In `R` una tale struttura è chiamata *data frame*.

Utilizzando i pacchetti del `tidyverse` (`tidyverse` è un insieme, o *bundle*, di pacchetti `R`), le operazioni di trasformazione dei dati risultano molto semplificate. Nel `tidyverse` i data frame vengono leggermente modificati e si chiamano `tibble`. Per la manipolazione dei dati vengono usati i seguenti pacchetti del `tidyverse`:

- `dplyr`
- `tidyr` (tibbles, dataframe e tabelle)

- `stringr` (stringhe)

Il pacchetto `dplyr` (al momento uno dei pacchetti più famosi e utilizzati per la gestione dei dati) offre una serie di funzionalità che consentono di eseguire le operazioni più comuni di manipolazione dei dati in maniera più semplice rispetto a quanto succeda quando usiamo le funzioni base di `R`.

P.7.2 Trattamento dei dati con `dplyr`

Il pacchetto `dplyr` include sei funzioni base: `filter()`, `select()`, `mutate()`, `arrange()`, `group_by()` e `summarise()`. Queste sei funzioni costituiscono i *verbi* del linguaggio di manipolazione dei dati. A questi sei verbi si aggiunge il pipe `%>%` che serve a concatenare più operazioni. In particolare, considerando una matrice osservazioni per variabili, `select()` e `mutate()` si occupano di organizzare le variabili, `filter()` e `arrange()` i casi, e `group_by()` e `summarise()` i gruppi.

Per introdurre le funzionalità di `dplyr`, utilizzeremo i dati `msleep` forniti dal pacchetto `ggplot2`. Tali dati descrivono le ore di sonno medie di 83 specie di mammiferi (Savage et al., 2007). Carichiamo il *bundle* `tidyverse` (che contiene `ggplot2`) e leggiamo nella memoria di lavoro l'oggetto `msleep`:

```
library("tidyverse")
data(msleep)
dim(msleep)
#> [1] 83 11
```

P.7.2.1 Operatore pipe

Prima di presentare le funzionalità di `dplyr`, introduciamo l'operatore pipe `%>%` del pacchetto `magrittr` – ma ora presente anche in base `R` nella versione `|>`. L'operatore pipe, `%>%` o `|>`, serve a concatenare varie funzioni insieme, in modo da inserire un'operazione dietro l'altra. Una spiegazione intuitiva dell'operatore pipe è stata fornita in un tweet di `@andrewheiss`. Consideriamo la seguente istruzione in pseudo-codice `R`:

```
leave_house(
  get_dressed(
```

```
    get_out_of_bed(
      wake_up(me, time = "8:00"),
      side = "correct"),
    pants = TRUE,
    shirt = TRUE),
  car = TRUE,
  bike = FALSE
)
```

Il listato precedente descrive una serie di (pseudo) funzioni concatenate, le quali costituiscono gli argomenti di altre funzioni. Scritto così, il codice è molto difficile da capire. Possiamo però ottenere lo stesso risultato utilizzando l'operatore pipe che facilita enormememnte la leggibilità del codice:

```
me %>%
  wake_up(time = "8:00") %>%
  get_out_of_bed(side = "correct") %>%
  get_dressed(pants = TRUE, shirt = TRUE) %>%
  leave_house(car = TRUE, bike = FALSE)
```

In questa seconda versione del (pseudo) codice `R` si capisce molto meglio ciò che vogliamo fare. Il `tibble me` viene passato alla funzione `wake_up()`. La funzione `wake_up()` ha come argomento l'ora del giorno: `time = "8:00"`. Una volta "svegliati" (wake up) dobbiamo scendere dal letto. Quindi l'output di `wake_up()` viene passato alla funzione `get_out_of_bed()` la quale ha come argomento `side = "correct"` perché vogliamo scendere dal letto dalla parte giusta. E così via.

Questo pseudo-codice chiarisce il significato dell'operatore pipe. L'operatore `%>%` è "syntactic sugar" per una serie di chiamate di funzioni concatenate, ovvero, detto in altre parole, consente di definire la relazione tra una serie di funzioni nelle quali il risultato (output) di una funzione viene utilizzato come l'input di una funzione successiva.

P.7.2.2 Estrarre una singola colonna con `pull()`

Ritorniamo ora all'esempio precedente. Iniziamo a trasformare il data frame `msleep` in un `tibble` (che è identico ad un data frame ma viene stampato sulla console in un modo diverso):

```
msleep <- tibble(msleep)
```

Estraiamo da `msleep` la variabile `sleep_total` usando il verbo `pull()`:

```
msleep %>%
  pull(sleep_total)
#>  [1] 12.1 17.0 14.4 14.9  4.0 14.4  8.7  7.0 10.1  3.0
#> [11]  5.3  9.4 10.0 12.5 10.3  8.3  9.1 17.4  5.3 18.0
#> [21]  3.9 19.7  2.9  3.1 10.1 10.9 14.9 12.5  9.8  1.9
#> [31]  2.7  6.2  6.3  8.0  9.5  3.3 19.4 10.1 14.2 14.3
#> [41] 12.8 12.5 19.9 14.6 11.0  7.7 14.5  8.4  3.8  9.7
#> [51] 15.8 10.4 13.5  9.4 10.3 11.0 11.5 13.7  3.5  5.6
#> [61] 11.1 18.1  5.4 13.0  8.7  9.6  8.4 11.3 10.6 16.6
#> [71] 13.8 15.9 12.8  9.1  8.6 15.8  4.4 15.6  8.9  5.2
#> [81]  6.3 12.5  9.8
```

P.7.2.3 Selezionare più colonne con `select()`

Se vogliamo selezionare da `msleep` un insieme di variabili, ad esempio `name`, `vore` e `sleep_total`, possiamo usare il verbo `select()`:

```
dt <- msleep %>%
  dplyr::select(name, vore, sleep_total)
dt
#> # A tibble: 83 x 3
#>   name                        vore  sleep_total
#>   <chr>                       <chr>       <dbl>
#> 1 Cheetah                     carni        12.1
#> 2 Owl monkey                  omni         17
#> 3 Mountain beaver             herbi        14.4
#> 4 Greater short-tailed shrew omni         14.9
#> 5 Cow                         herbi         4
#> 6 Three-toed sloth            herbi        14.4
#> 7 Northern fur seal           carni         8.7
#> 8 Vesper mouse                <NA>          7
#> # ... with 75 more rows
```

laddove la sequenza di istruzioni precedenti significa che abbiamo passato `msleep` alla funzione `select()` contenuta nel pacchetto `dplyr` e l'output

di `select()` è stato salvato (usando l'operatore di assegnazione, `<-`) nell'oggetto `dt`. Alla funzione `select()` abbiamo passato gli argomenti `name`, `vore` e `sleep_total`.

P.7.2.4 Filtrare le osservazioni (righe) con `filter()`

Il verbo `filter()` consente di selezionare da un `tibble` un sottoinsieme di righe (osservazioni). Per esempio, possiamo selezionare tutte le osservazioni nella variabile `vore` contrassegnate come `carni` (ovvero, tutti i carnivori):

```
dt %>%
  dplyr::filter(vore == "carni")
#> # A tibble: 19 x 3
#>   name                 vore  sleep_total
#>   <chr>                <chr>       <dbl>
#> 1 Cheetah              carni        12.1
#> 2 Northern fur seal    carni         8.7
#> 3 Dog                  carni        10.1
#> 4 Long-nosed armadillo carni        17.4
#> 5 Domestic cat         carni        12.5
#> 6 Pilot whale          carni         2.7
#> 7 Gray seal            carni         6.2
#> 8 Thick-tailed opposum carni        19.4
#> # ... with 11 more rows
```

Per utilizzare il verbo `filter()` in modo efficace è neccessario usare gli operatori relazionali (Tabella P.1) e gli operatori logici (Tabella P.2) di `R`. Per un approfondimento, si veda il Capitolo Comparisons[10] di *R for Data Science*.

P.7.2.5 Creare una nuova variabile con `mutate()`

Talvolta vogliamo creare una nuova variabile, per esempio, sommando o dividendo due variabili, oppure calcolandone la media. A questo scopo si usa il verbo `mutate()`. Per esempio, se vogliamo esprimere i valori di `sleep_total` in minuti, moltiplichiamo per 60:

[10] https://r4ds.had.co.nz/transform.html

Tabella P.1: *Operatori relazionali.*

uguale	==
diverso	!=
minore	<
maggiore	>
minore o uguale	<=
maggiore o uguale	>=

Tabella P.2: *Operatori logici.*

AND	&
OR	|
NOT	!

```
dt %>%
  mutate(
    sleep_minutes = sleep_total * 60
  ) %>%
  dplyr::select(sleep_total, sleep_minutes)
#> # A tibble: 83 x 2
#>   sleep_total sleep_minutes
#>         <dbl>         <dbl>
#> 1        12.1           726
#> 2        17            1020
#> 3        14.4           864
#> 4        14.9           894
#> 5         4             240
#> 6        14.4           864
#> 7         8.7           522
#> 8         7             420
#> # ... with 75 more rows
```

P.7.2.6 Ordinare i dati con `arrange()`

Il verbo `arrange()` ordina i dati in base ai valori di una o più variabili. Per esempio, possiamo ordinare la variabile `sleep_total` dal valore più alto al più basso in questo modo:

```
dt %>%
  arrange(
    desc(sleep_total)
  )
#> # A tibble: 83 x 3
#>   name                  vore    sleep_total
#>   <chr>                 <chr>         <dbl>
#> 1 Little brown bat      insecti        19.9
#> 2 Big brown bat         insecti        19.7
#> 3 Thick-tailed opposum  carni          19.4
#> 4 Giant armadillo       insecti        18.1
#> 5 North American Opossum omni          18
#> 6 Long-nosed armadillo  carni          17.4
#> 7 Owl monkey            omni           17
#> 8 Arctic ground squirrel herbi         16.6
#> # ... with 75 more rows
```

P.7.2.7 Raggruppare i dati con `group_by()`

Il verbo `group_by()` raggruppa insieme i valori in base a una o più variabili. Lo vedremo in uso in seguito insieme a `summarise()`.

Nota: con `dplyr()`, le operazioni raggruppate vengono iniziate con la funzione `group_by()`. È una buona norma utilizzare `ungroup()` alla fine di una serie di operazioni raggruppate, altrimenti i raggruppamenti verranno mantenuti nelle analisi successiva, il che non è sempre auspicabile.

P.7.2.8 Sommario dei dati con `summarise()`

Il verbo `summarise()` collassa il dataset in una singola riga dove viene riportato il risultato della statistica richiesta. Per esempio, la media del tempo totale del sonno è

```
dt %>%
  summarise(
    m_sleep = mean(sleep_total, na.rm = TRUE)
  )
#> # A tibble: 1 x 1
#>   m_sleep
```

```
#>     <dbl>
#> 1    10.4
```

P.7.2.9 Operazioni raggruppate

Sopra abbiamo visto come i mammiferi considerati dormano, in media, 10.4 ore al giorno. Troviamo ora il sonno medio in funzione di `vore`:

```
dt %>%
  group_by(vore) %>%
  summarise(
    m_sleep = mean(sleep_total, na.rm = TRUE),
    n = n()
  )
#> # A tibble: 5 x 3
#>   vore    m_sleep     n
#>   <chr>     <dbl> <int>
#> 1 carni     10.4     19
#> 2 herbi      9.51    32
#> 3 insecti   14.9      5
#> 4 omni      10.9     20
#> 5 <NA>      10.2      7
```

Si noti che, nel caso di 7 osservazioni, il valore di `vore` non era specificato. Per tali osservazioni, dunque, la classe di appartenenza è `NA`.

P.7.2.10 Applicare una funzione su più colonne: `across()`

È spesso utile eseguire la stessa operazione su più colonne, ma copiare e incollare è sia noioso che soggetto a errori:

```
df %>%
  group_by(g1, g2) %>%
  summarise(
    a = mean(a),
    b = mean(b),
    c = mean(c),
    d = mean(d)
  )
```

In tali circostanze è possibile usare la funzione `across()` che consente di riscrivere il codice precedente in modo più succinto:

```
df %>%
  group_by(g1, g2) %>%
  summarise(across(a:d, mean))
```

Per i dati presenti, ad esempio, possiamo avere:

```
msleep %>%
  group_by(vore) %>%
  summarise(across(starts_with("sleep"), ~ mean(.x, na.rm = TRUE)))
#> # A tibble: 5 x 4
#>   vore    sleep_total sleep_rem sleep_cycle
#>   <chr>         <dbl>     <dbl>       <dbl>
#> 1 carni         10.4       2.29       0.373
#> 2 herbi          9.51      1.37       0.418
#> 3 insecti       14.9       3.52       0.161
#> 4 omni          10.9       1.96       0.592
#> 5 <NA>          10.2       1.88       0.183
```

P.7.3 Dati categoriali in `R`

Consideriamo una variabile che descrive il genere e include le categorie `male`, `female` e `non-conforming`. In `R`, ci sono due modi per memorizzare queste informazioni. Uno è usare la classe *character strings* e l'altro è usare la classe *factor*. Non ci addentrimo qui nelle sottigliezze di questa distinzione, motivata in gran parte per le necessità della programmazione con le funzioni di `tidyverse`. Per gli scopi di questo insegnamento sarà sufficiente codificare le variabili qualitative usando la classe *factor*. Una volta codificati i dati qualitativi utilizzando la classe *factor*, si pongono spesso due problemi:

1. modificare le etichette dei livelli (ovvero, le modalità) di un fattore,
2. riordinare i livelli di un fattore.

P.7.3.1 Modificare le etichette dei livelli di un fattore

Esaminiamo l'esempio seguente.

```
f_1 <- c("old_3", "old_4", "old_1", "old_1", "old_2")
f_1 <- factor(f_1)
y <- 1:5
df <- tibble(f_1, y)
df
#> # A tibble: 5 x 2
#>   f_1       y
#>   <fct> <int>
#> 1 old_3     1
#> 2 old_4     2
#> 3 old_1     3
#> 4 old_1     4
#> 5 old_2     5
```

Supponiamo ora di volere che i livelli del fattore `f_1` abbiano le etichette `new_1`, `new_2`, ecc. Per ottenere questo risultato usiamo la funzione `forcats::fct_recode()`:

```
df <- df %>%
  mutate(f_1 =
    forcats::fct_recode(
      f_1,
      "new_poco" = "old_1",
      "new_medio" = "old_2",
      "new_tanto" = "old_3",
      "new_massimo" = "old_4"
      )
   )
df
#> # A tibble: 5 x 2
#>   f_1             y
#>   <fct>       <int>
#> 1 new_tanto       1
#> 2 new_massimo     2
#> 3 new_poco        3
#> 4 new_poco        4
#> 5 new_medio       5
```

P.7.3.2 Riordinare i livelli di un fattore

Spesso i livelli dei fattori hanno un ordinamento naturale. Quindi, gli utenti devono avere un modo per imporre l'ordine desiderato sulla codifica delle loro variabili qualitative. Se per qualche motivo vogliamo ordinare i livelli `f_1` in ordine inverso, ad esempio, possiamo procedere nel modo seguente.

```
df$f_1 <- factor(df$f_1,
  levels = c(
    "new_massimo", "new_tanto", "new_medio", "new_poco"
  )
)
summary(df$f_1)
#> new_massimo   new_tanto   new_medio    new_poco
#>           1           1           1           2
```

Per approfondire le problematiche della manipolazione di variabili qualitative in `R`, si veda McNamara and Horton (2018).

P.7.4 Creare grafici con `ggplot2()`

Il pacchetto `ggplot2()` è un potente strumento per rappresentare graficamente i dati. Le iniziali del nome, `gg`, si riferiscono alla "Grammar of Graphics", che è un modo di pensare le figure come una serie di layer stratificati. Originariamente descritta da Wilkinson (2012), la grammatica dei grafici è stata aggiornata e applicata in `R` da Hadley Wickham, il creatore del pacchetto.

La funzione da cui si parte per inizializzare un grafico è `ggplot()`. La funzione `ggplot()` richiede due argomenti. Il primo è l'oggetto di tipo data.frame che contiene i dati da visualizzare – in alternativa al primo argomento, un dataframe può essere passato a `ggplot()` mediante l'operatore pipe. Il secondo è una particolare lista che viene generata dalla funzione `aes()`, la quale determina l'aspetto (*aesthetic*) del grafico. La funzione `aes()` richiede necessariamente di specificare "x" e "y", ovvero i nomi delle colonne del data.frame che è stato utilizzato quale primo argomento di `ggplot()` (o che è stato passato da pipe), le quali rappresentano le variabili da porre rispettivamente sugli assi orizzontale e verticale.

La definizione della tipologia di grafico e i vari parametri sono poi defini-

ti successivamente, aggiungendo all'oggetto creato da `ggplot()` tutte le componenti necessarie. Saranno quindi altre funzioni, come `geom_bar()`, `geom_line()` o `geom_point()` a occuparsi di aggiungere al livello di base barre, linee, punti, e così via. Infine, tramite altre funzioni, ad esempio `labs()`, sarà possibile definire i dettagli più fini.

Gli elementi grafici (bare, punti, segmenti, …) usati da `ggplot2` sono chiamati `geoms`. Mediante queste funzioni è possibile costruire diverse tipologie di grafici:

- `geom_bar()`: crea un layer con delle barre;
- `geom_point()`: crea un layer con dei punti (diagramma a dispersione);
- `geom_line()`: crea un layer con una linea retta;
- `geom_histogram()`: crea un layer con un istogramma;
- `geom_boxplot()`: crea un layer con un box-plot;
- `geom_errorbar()`: crea un layer con barre che rappresentano intervalli di confidenza;
- `geom_hline()` e `geom_vline()` : crea un layer con una linea orizzontale o verticale definita dall'utente.

Un comando generico ha la seguente forma:

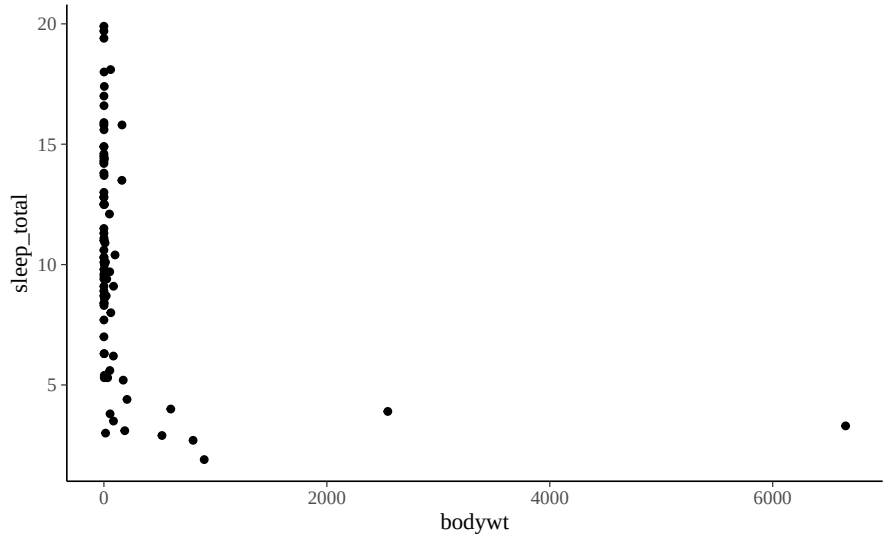
```
my_graph <- my_data %>%
  ggplot(aes(x_var, y_var)) +
  geom_...()
```

La prima volta che si usa il pacchetto `ggplot2` è necessario installarlo. Per fare questo possiamo installare `tidyverse` che, oltre a caricare `ggplot2`, carica anche altre utili funzioni per l'analisi dei dati. Ogni volta che si inizia una sessione R è necessario attivare i pacchetti che si vogliono usare, ma non è necessario istallarli una nuova volta. Se è necessario specificare il pacchetto nel quale è contenuta la funzione che vogliamo utilizzare, usiamo la sintassi `package::function()`. Per esempio, l'istruzione `ggplot2::ggplot()` rende esplicito che stiamo usando la funzione `ggplot()` contenuta nel pacchetto `ggplot2`.

P.7.5 Diagramma a dispersione

Consideriamo nuovamenti i dati contenuti nel `tibble msleep` e poniamoci il problema di rappresentare graficamente la relazione tra il numero medio di ore di sonno giornaliero (`sleep_total`) e il peso dell'animale (`bodywt`). Usando le impostazioni di default di `ggplot2`, con le istruzioni seguenti, otteniamo il grafico fornito dalla figura seguente.

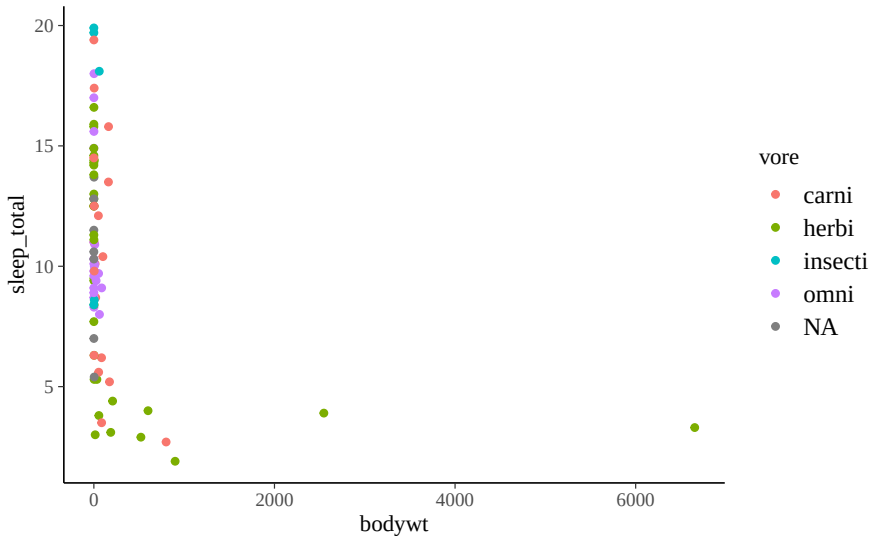
```
data("msleep")
p <- msleep %>%
  ggplot(
    aes(x = bodywt, y = sleep_total)
  ) +
  geom_point()
print(p)
```



Coloriamo ora in maniera diversa i punti che rappresentano animali carnivori, erbivori, ecc.

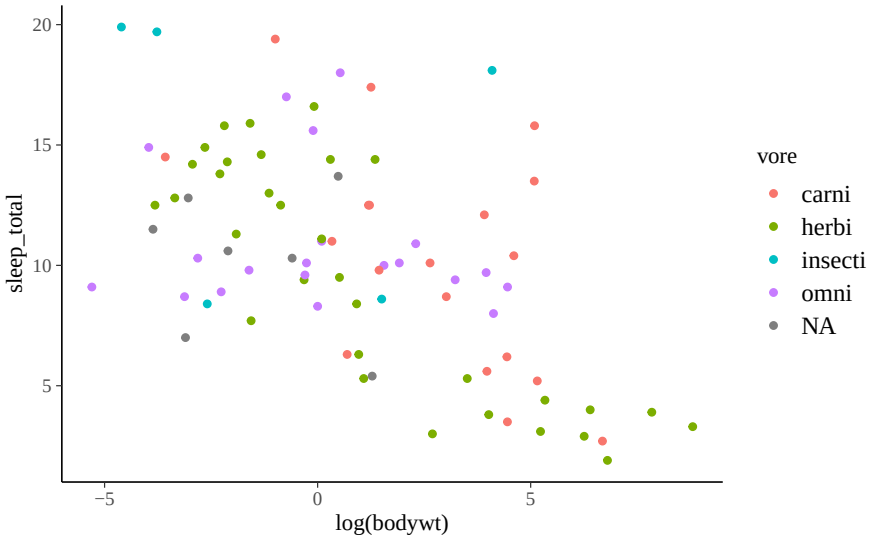
```
p <- msleep %>%
  ggplot(
    aes(x = bodywt, y = sleep_total, col = vore)
  ) +
```

```
  geom_point()
print(p)
```



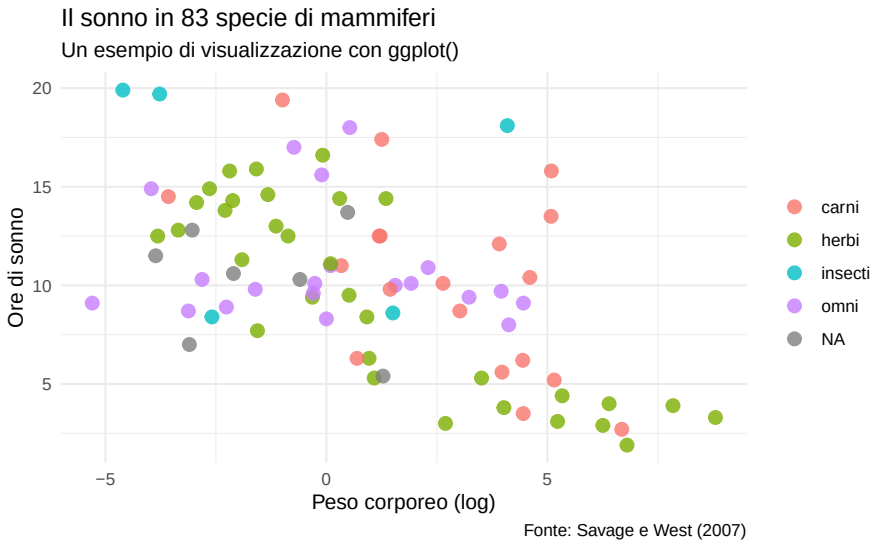
È chiaro, senza fare alcuna analisi statistica, che la relazione tra le due variabili non è lineare. Trasformando in maniera logaritmica i valori dell'asse x la relazione si linearizza.

```
p <- msleep %>%
  ggplot(
    aes(x = log(bodywt), y = sleep_total, col = vore)
  ) +
  geom_point()
print(p)
```



Infine, aggiustiamo il "tema" del grafico (si noti l'utilizzo di una tavolozza di colori adatta ai daltonici mediante il pacchetto `viridis`), aggiungiamo le etichette sugli assi e il titolo.

```
library("viridis")
msleep %>%
  ggplot(
    aes(x = log(bodywt), y = sleep_total, col = vore)
  ) +
  geom_point(size = 3, alpha = .8) +
  labs(
    x = "Peso corporeo (log)",
    y = "Ore di sonno",
    title = "Il sonno in 83 specie di mammiferi",
    subtitle = "Un esempio di visualizzazione con ggplot()",
    caption = "Fonte: Savage e West (2007)"
  ) +
  scale_fill_viridis(discrete = TRUE, option = "viridis") +
  theme_minimal() +
  theme(legend.title = element_blank())
```

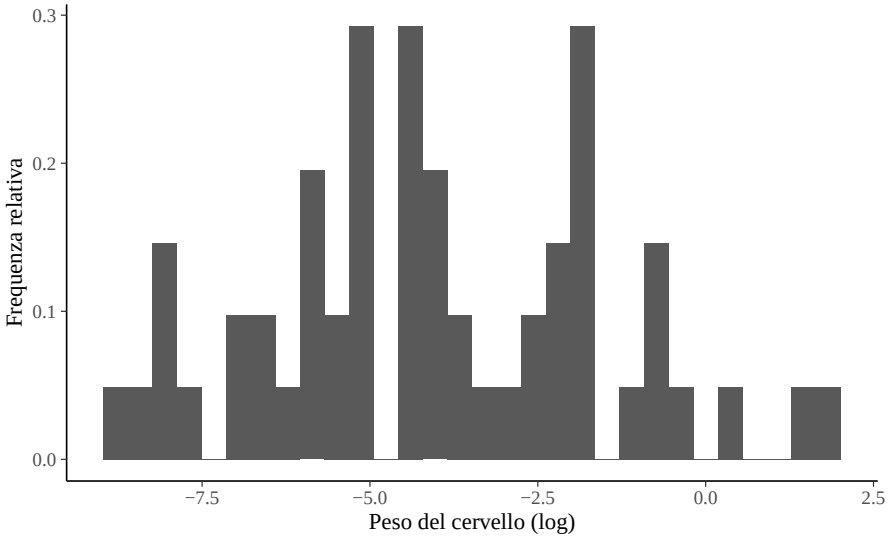


La visualizzazione può essere migliorata cambiando le etichette della legenda del grafico. Per fare questo è necessario intervenire sui dati prima di usare `ggplot()` – per esempio, come abbiamo fatto in precedenza con la funzione `forcats::fct_recode()`.

P.7.5.1 Istogramma

Creiamo ora un istogramma che rappresenta la distribuzione del (logaritmo del) peso medio del cervello delle 83 specie di mammiferi considerate da Savage and West (2007). L'argomento `aes(y = ..density..)` in `geom_histogram()` produce le frequenze relative. L'opzione di default (senza questo argomento) porta `ggplot()` a rappresentare le frequenze assolute.

```
msleep %>%
  ggplot(
    aes(log(brainwt))
  ) +
  geom_histogram(aes(y = ..density..)) +
  labs(
    x = "Peso del cervello (log)",
    y = "Frequenza relativa"
  ) +
  theme(legend.title = element_blank())
```



P.7.6 Scrivere il codice `R` con stile

Uno stile di programmazione è un insieme di regole per la gestione dell'indentazione dei blocchi di codice, per la creazione dei nomi dei file e delle variabili e per le convenzioni tipografiche che vengono usate. Scrivere il codice in `R` con stile consente di creare listati più leggibili e semplici da modificare, minimizza la possibilità di errore, e consente correzioni e modifiche più rapide. Vi sono molteplici stili di programmazione che possono essere utilizzati dall'utente, anche se è bene attenersi a quelle che sono le convenzioni maggiormente diffuse, allo scopo di favorire la comunicazione. In ogni caso, l'importante è di essere coerenti, ovvero di adottare le stesse convenzioni in tutte le parti del codice che si scrive. Ad esempio, se si sceglie di usare lo stile `snake_case` per il nome composto di una variabile (es., `personality_trait`), non è appropriato usare lo stile *lower Camel case* per un'altra variabile (es., `socialStatus`). Dato che questo argomento è stato trattato ampiamente in varie sedi, mi limito qui a rimandare ad uno stile di programmazione[11] molto popolare, quello proposto da Hadley Wickham, il creatore di `tidyverse`. La soluzione più semplice è quella installare `stiler`, che è uno RStudio Addin, e formattare il codice in maniera automatica utilizzando lo stile proposto da Hadley Wickham. Si possono ottenere informazioni su `stiler` seguendo

[11] http://style.tidyverse.org/

questo link[12].

P.8 Ottenere informazioni sulle funzioni R

Oltre a `?help <funzione>`, è possibile ricorrere al pacchetto `introverse`:

```
remotes::install_github("spielmanlab/introverse")
```

Il pacchetto `introverse` fornisce documentazione alternativa per funzioni e concetti comunemente usati in Base R e nel `tidyverse`. Istruzioni relative all'uso delle funzioni disponibili vengono fornite quando si carica il pacchetto:

```
library("introverse")
```

```
Welcome to the {introverse}!

Not sure where to start? You can...

- Run show_topics() to see all the different functions
  and topics you can ask for help with.

- Run show_topics("library or category of interest") to see
  all the different functions within a certain library/category
  of interest. For example, to see all help topics for {dplyr}
  functions, run: show_topics("dplyr").

- Run get_help("carnivores") and get_help("msleep") to learn
  about the datasets used in examples.

- Run the function get_help() to see the {introverse} docs
  for a function or topic. For example, to get help using
  the `length()` function, run: get_help("length").
  Don't forget quotation marks around the argument to get_help()!
```

[12] https://github.com/r-lib/styler

P.9 Flusso di lavoro riproducibile

P.9.1 La crisi della riproducibilità

"Per il metodo scientifico è essenziale che gli esperimenti siano riproducibili. Vale a dire che una persona diversa dallo sperimentatore originale deve essere in grado di ottenere gli stessi risultati seguendo lo stesso protocollo sperimentale (Gilbert Chin)." Ma in psicologia (e non solo) la riproducibilità è inferiore a quanto previsto o desiderato. In un famoso studio pubblicato su *Science*, un ampio gruppo di ricercatori (Open Science Collaboration and others, 2015) è riuscito a replicare solo il 40 per cento circa dei risultati di 100 studi di psicologia cognitiva e sociale pubblicati in precedenza. I risultati di questo studio, e di molti altri pubblicati in seguito, sono stati interpretati in modi diversi. La preoccupazione sulla riproducibilità della ricerca è stata espressa mediante l'affermare secondo la quale "la maggior parte dei risultati della ricerca sono falsi" (Ioannidis, 2005) oppure mediante l'affermazione secondo cui "dobbiamo apportare modifiche sostanziali al modo in cui conduciamo la ricerca" (Cumming, 2014). Alcuni ricercatori sono arrivati a definire la presente situazione come una "crisi della riproducibilità dei risultati della ricerca".

Il termine "riproducibilità" (o "replicabilità") è stato definito in vari modi. Consideriamo la definizione fornita da Goodman et al. (2016):

- la riproducibilità dei metodi "si riferisce al fatto che il ricercatore fornisce dettagli sufficienti sulle procedure e sui dati dello studio in modo che le stesse procedure possano … essere replicate esattamente" (pag. 2) con gli stessi dati;
- la riproducibilità dei risultati "si riferisce all'ottenimento degli stessi risultati dalla conduzione di uno studio indipendente le cui procedure replicano il più esattamente possibile quelle dell'esperimento originale" (pag. 2-3) con dati indipendenti;
- la riproducibilità inferenziale "si riferisce alla possibilità di trarre conclusioni qualitativamente simili da una replica indipendente di uno studio o da una nuova analisi dello studio originale" (pag. 4).

Per gli scopi presenti, ci focalizzeremo qui sulla riproducibilità dei metodi. Cioè, discuteremo di come `R` può aiutarci a migliorare questo aspetto

della riproducibilità. In questo capitolo mostreremo come R possa essere utilizzato all'interno di un flusso di lavoro (*workflow*) riproducibile che integra (1) il codice di analisi dei dati, (2) i dati medesimi e (3) il testo della relazione che comunica i risultati dello studio. A tal fine utilizzeremo due pacchetti R: `rmarkdown` e `knitr`. Questi pacchetti consentono di unire il codice R ad un linguaggio di marcatura (o di markup) chiamato Markdown. Il linguaggio di markup Markdown sta diventando sempre più popolare e viene usato, oltre che per creare reports[13] di analisi di dati, anche per creare siti web[14], blog[15], libri[16], articoli accademici[17], curriculum vitae[18], slide[19], tesi di laurea[20]. Per esempio, il presente sito web è stato scritto usando R-markdown.

P.9.2 R-markdown

Un linguaggio di markup permette di aggiungere mediante marcatori (tag) informazioni sulla struttura e sulla formattazione da applicare ad un documento. Un'introduzione al linguaggio Markdown può essere trovata, per esempio, qui[21] oppure qui[22].

In questo capitolo ci focalizzeremo però sugli aspetti più importanti di R-markdown che permette di costruire documenti in cui combinare testo formattato (quindi non solo commenti ma anche formule, titoli etc) e istruzioni codice (`R` e non solo) con i corrispettivi output. Informazioni dettagliate su R-markdown sono disponibili qui[23] e qui[24].

Un file R-markdown è composto da tre tipi di oggetti:

1. header in formato `YAML` delimitato da `---`,
2. testo in formato `markdown`,
3. blocchi ("chunks") di codice `R`, delimitati da tre apici.

[13] https://avehtari.github.io/ROS-Examples/Simplest/simplest.html
[14] https://alison.rbind.io
[15] https://djnavarro.net
[16] https://r4ds.had.co.nz
[17] https://osf.io/9te8p/
[18] https://github.com/mitchelloharawild/vitae
[19] https://rmarkdown.rstudio.com/lesson-11.html
[20] https://github.com/ismayc/thesisdown
[21] https://rmarkdown.rstudio.com/authoring_pandoc_markdown.html
[22] https://experienceleague.adobe.com/docs/contributor/contributor-guide/writing-essentials/markdown.html?lang=it#estensioni-personalizzate-markdown
[23] https://bookdown.org/yihui/rmarkdown/
[24] https://bookdown.org/yihui/rmarkdown-cookbook/

P.9.2.1 Header

L'intestazione di un documento `.Rmd` (R-markdown) corrisponde al cosiddetto *YAML header* (un acronimo che significa *Yet Another Markup Language*). Lo YAML header controlla le caratteristiche generali del documento, incluso il tipo di documento che viene prodotto (un documento HTML che può essere visualizzato su tutti i principali browser, un documento Microsoft Word o un PDF se abbiamo installato LaTeX sul nostro computer), la dimensione del carattere, lo stile, il titolo, l'autore, ecc. Nello YAML header (a differenza del codice `R`) è necessario rispettare la spaziatura prestabilita delle istruzioni che vengono elencate. Gli elementi principali sono `title:`, `author:`, `output:`.

L'argomento di `output:` è dove diciamo a R-markdown quale tipo di file vogliamo che venga prodotto. Il tipo più flessibile, che non richiede alcuna configurazione, è `html_document`.

P.9.2.2 Testo

Alla conclusione dello YAML header inizia il documento R-markdown. Da questo punto in poi possiamo utilizzare testo normale, codice `R` e sintassi Markdown per controllare cosa viene mostrato e come.

P.9.2.3 Formattazione

È possibile contrassegnare intestazioni, grassetto e corsivo come indicato di seguito.

```
# Intestazione 1
## Intestazione 2
### Intestazione 3
#### Intestazione 4
##### Intestazione 5
###### Intestazione 6

Questo è un testo normale.
Possiamo scrivere in **grassetto** il testo usando due asterischi.
Possiamo scrivere in *corsivo* usando un asterisco.

>Questa è un'**area rientrata**.

Questa riga invece non è più rientrata.
```

P.9.2.4 Elenchi

Per creare un elenco puntato si utilizza il segno più, il trattino o l'asterisco. Tutte le tre soluzioni portano allo stesso risultato.

```
- Punto 1 della lista
- Punto 2 della lista
- Punto 3 della lista
```

Un elenco numerato, invece, si crea con un numero seguito da un punto.

```
1. Punto 1 della lista
2. Punto 2 della lista
3. Punto 3 della lista
```

P.9.2.5 Hyperlink

Per inserire un hyperlink ci sono due metodi:

- specificare solo il percorso `<http://rmarkdown.rstudio.com>`, http://rmarkdown.rstudio.com
- creare un link[25] con `[link](http://rmarkdown.rstudio.com)`

P.9.2.6 Immagini

Per inserire un'immagine la sintassi è molto simile: `![Esempio di immagine inserita in un documento R-markdown.](images/hex-rmarkdown.png){width=20%}`:

Figura P.2: *Esempio di immagine inserita in un documento R-markdown.*

P.9.2.7 Codice inline

Per contrassegnare un'area di testo come codice, markdown utilizza il cosiddetto backtick, noto anche come gravis o accento grave, da non

[25] http://rmarkdown.rstudio.com

confondere con la virgoletta singola. La marcatura prevede un accento all'inizio e uno alla fine dell'area di testo corrispondente.

```
Questo è `codice`.
```

P.9.2.8 Equazioni

Equazioni possono essere inserite in un documento R-markdown usando la sintassi LaTeX. Qualsiasi cosa all'interno del segno di dollaro $ viene trattata come un'equazione "inline". Qualunque cosa all'interno di due segni di dollaro $$ viene trattata come un'equazione a sé stante.

Per esempio, questa è la formula della distribuzione Normale espressa in notazione LaTeX e riprodotta all'interno di un documento R-markdown:

```
f(x) = \frac{1}{\sigma\sqrt{2\pi}}
  \exp\left( -\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^{\!2}\,\right)
```

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

P.9.2.9 Codice R

In un documento R-markdown istruzioni di codice vengono inserite in blocchi delimitati da tre apici. Ciò consente di valutare il codice all'interno del documento e di produrre un output che verrà stampato nel documento stesso. Possiamo dunque stampare tabelle e figure prodotti direttamente dal codice `R`. Ciò significa inoltre, che se qualcosa cambia nei dati o nelle analisi dei dati, le tabelle e le figure si aggiorneranno automaticamente.

Un chunk `R` viene valutato proprio come il normale codice `R`, quindi si applica tutto ciò che abbiamo imparato nei capitoli precedenti. Se il chunk `R` produce un output, questo output verrà visualizzato nel documento.

P.9.3 Compilare la presentazione R-markdown

Ma dove si trova questo magico documento che include il testo e l'output prodotto dal codice `R`? Ottima domanda. Siamo stati abituati ai programmi di videoscrittura (come Microsoft Word) che si conformano al cosiddetto stile "WYSIWYG" (What You See Is What You Get) – cioè, si vede come apparirà il documento stampato mentre lo si digita.

Questo può avere alcuni vantaggi ma può anche essere molto limitante. R-Markdown, d'altra parte, funziona in modo diverso. Ovvero, deve essere "compilato" (knitted) per passare dal file sorgente al documento formattato. In RStudio, tale operazione è semplice: c'è un pulsante in alto a sinistra nel pannello di scripting di un documento `.Rmd`. È sufficiente selezionare tale pulsante e il nostro documento verrà creato.

È importante notare che il codice del documento deve essere autonomo. Ciò significa che tutto ciò che vogliamo che venga eseguito deve essere incluso nel documento, indipendentemente da ciò che era già stato eseguito al di fuori di esso. Ad esempio, è perfettamente legittimo (e anche molto utile) testare il codice `R` al di fuori del documento `Rmd`. Tuttavia, quando compiliamo il documento `Rmd`, tutto ciò che è stato fatto al di fuori del documento `Rmd` viene dimenticato. Ciò consente di creare un documento autosufficiente che favorisce la riproducibilità dei metodi di analisi dei dati: utilizzando uno specifico documento `Rmd` con un campione di dati si giunge sempre allo stesso risultato e alla stessa interpretazione. Ciò non è invece vero se si utilizza un software con un interfaccia point-and-click.

P.10 Dati mancanti

P.10.1 Motivazione

La pulizia dei dati (*data cleaning*) in `R` è fondamentale per effettuare qualsiasi analisi. Uno degli aspetti più importanti della pulizia dei dati è la gestione dei dati mancanti. I valori mancanti (*missing values*) vengono indicati dal codice `NA`, che significa *not available* — non disponibile.

P.10.2 Trattamento dei dati mancanti

Se una variabile contiene valori mancanti, `R` non è in grado di applicare ad essa alcune Funzioni, come ad esempio la media. Per questa ragione, la gran parte delle funzioni di `R` prevedono modi specifici per trattare i valori mancanti.

Ci sono diversi tipi di dati "mancanti" in `R`;

- `NA` - generico dato mancante;
- `NaN` - il codice `NaN` (*Not a Number*) indica i valori numerici impossibili, quali ad esempio un valore 0/0;

- `Inf` e `-Inf` - Infinity, si verifca, ad esempio, quando si divide un numero per 0.

La funzione `is.na()` ritorna un output che indica con TRUE le celle che contengono NA o NaN.

Si noti che

- se `is.na(x)` è TRUE, allora `!is.na(x)` è FALSE;
- `all(!is.na(x))` ritorna TRUE se tutti i valori `x` sono NOT NA;
- `any(is.na(x))` risponde alla domanda: c'è qualche valore NA (almeno uno) in `x`?;
- `complete.cases(x)` ritorna TRUE se ciascun elemento di `x` è is NOT NA; ritorna FALSE se almeno un elemento di `x` è NA;

Le funzioni `R is.nan()` e `is.infinite()` si applicano ai tipi di dati `NaN` e `Inf`.

Per esempio, consideriamo il seguente data.frame:

```
d <- tibble(
  w = c(1, 2, NA, 3, NA),
  x = 1:5,
  y = 1,
  z = x ^ 2 + y,
  q = c(3, NA, 5, 1, 4)
)
d
#> # A tibble: 5 x 5
#>       w     x     y     z     q
#>   <dbl> <int> <dbl> <dbl> <dbl>
#> 1     1     1     1     2     3
#> 2     2     2     1     5    NA
#> 3    NA     3     1    10     5
#> 4     3     4     1    17     1
#> 5    NA     5     1    26     4
```

```
is.na(d$w)
#> [1] FALSE FALSE  TRUE FALSE  TRUE
is.na(d$x)
#> [1] FALSE FALSE FALSE FALSE FALSE
```

Per creare un nuovo Dataframe senza valori mancanti:

```
d_clean <- d[complete.cases(d), ]
d_clean
#> # A tibble: 2 x 5
#>       w     x     y     z     q
#>   <dbl> <int> <dbl> <dbl> <dbl>
#> 1     1     1     1     2     3
#> 2     3     4     1    17     1
```

Oppure, se vogliamo eliminare le righe con NA solo in una variabile:

```
d1 <- d[!is.na(d$q), ]
d1
#> # A tibble: 4 x 5
#>       w     x     y     z     q
#>   <dbl> <int> <dbl> <dbl> <dbl>
#> 1     1     1     1     2     3
#> 2    NA     3     1    10     5
#> 3     3     4     1    17     1
#> 4    NA     5     1    26     4
```

Se vogliamo esaminare le righe con i dati mancanti in qualunque colonna:

```
d_na <- d[!complete.cases(d), ]
d_na
#> # A tibble: 3 x 5
#>       w     x     y     z     q
#>   <dbl> <int> <dbl> <dbl> <dbl>
#> 1     2     2     1     5    NA
#> 2    NA     3     1    10     5
#> 3    NA     5     1    26     4
```

Spesso i valori mancanti vengono sostiuti con valori "ragionevoli", come ad esempio la media dei valori in quella colonna del Dataframe. Oppure, vengono considerati come "ragionevoli" i valori che vengono predetti conoscendo le altre variabili del Dataframe. Questa procedura si chiama *imputazione multipla*. Questo è però un argomento avanzato che non verrà trattato in questo insegnamento. La cosa più semplice da fare, in

presenza di dati mancanti, è semplicemente quella di escludere tutte le righe nelle quali ci sono degli NAs.

Bibliografia

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of statistical software*, 76(1):1–32.

Cumming, G. (2014). The new statistics: Why and how. *Psychological science*, 25(1):7–29.

Eckhardt, R. (1987). Stan Ulam, John Von Neumann and the Monte Carlo Method. *Los Alamos Science Special Issue*.

Goodman, S. N., Fanelli, D., and Ioannidis, J. P. (2016). What does research reproducibility mean? *Science translational medicine*, 8(341):341ps12–341ps12.

Grolemund, G. (2014). *Hands-on programming with R: Write your own functions and simulations*. O'Reilly Media, Inc.

Horn, S. and Loewenstein, G. (2021). Underestimating learning by doing. *Available at SSRN 3941441*.

Ioannidis, J. P. (2005). Why most published research findings are false. *PLoS medicine*, 2(8):e124.

McNamara, A. and Horton, N. J. (2018). Wrangling categorical data in r. *The American Statistician*, 72(1):97–104.

Open Science Collaboration and others (2015). Estimating the reproducibility of psychological science. *Science*, 349(6251):aac4716.

Savage, V. M., Allen, A. P., Brown, J. H., Gillooly, J. F., Herman, A. B., Woodruff, W. H., and West, G. B. (2007). Scaling of number, size, and metabolic rate of cells with body size in mammals. *Proceedings of the National Academy of Sciences*, 104(11):4718–4723.

Savage, V. M. and West, G. B. (2007). A quantitative, theoretical framework for understanding mammalian sleep. *Proceedings of the National Academy of Sciences*, 104(3):1051–1056.

Wilkinson, L. (2012). The grammar of graphics. In *Handbook of computational statistics*, pages 375–414. Springer.

Zetsche, U., Bürkner, P.-C., and Renneberg, B. (2019). Future expectations in clinical depression: Biased or realistic? *Journal of Abnormal Psychology*, 128(7):678–688.