NoiStudiamo

DISPENSA DI LABORATORIO

Metodi Matematici e Statistici

Probabilità e statistica in Python: simulazione, intervalli, test, regressione, metodo Monte Carlo e catene di Markov

A CURA DI

Damiano La Pusata

Ottobre 2026

INDICE

Indice dei capitoli

I capitoli della dispensa, nell'ordine in cui conviene studiarli.

1. L'ambiente di lavoro 3

2. Probabilità e simulazione 8

3. Distribuzioni: teoria e simulazione a confronto 12

4. Statistica descrittiva 18

5. Stimatori e intervalli di confidenza 22

6. Test di ipotesi 26

7. Regressione lineare 31

8. Numeri pseudocasuali e metodo Monte Carlo 35

9. Catene di Markov 40

10. Guida rapida 44

CAPITOLO 1

L'ambiente di lavoro

Le tre librerie che servono, gli array di NumPy, la generazione di numeri casuali, le distribuzioni di scipy.stats e i grafici: la cassetta degli attrezzi usata in tutti i capitoli successivi.

Come usare questa dispensa

Questa dispensa accompagna la dispensa di teoria e mostra come risolvere in Python i problemi di probabilità e statistica del corso. Ogni capitolo corrisponde a un tipo di problema e presenta il codice necessario, scritto nel modo più semplice possibile e commentato riga per riga, seguito dall'output che produce. Tutti i frammenti sono stati eseguiti davvero: rieseguendoli con lo stesso seme si ottengono gli stessi numeri.

Si assume solo la programmazione di base: variabili, condizioni, cicli, funzioni e liste. Non serve conoscere Python in profondità, perché le istruzioni che ricorrono sono poche e vengono spiegate la prima volta che compaiono.

DA RICORDARE

Quasi ogni esercizio di laboratorio segue quattro passi: **calcolare** la quantità richiesta con la formula teorica; **simulare** l'esperimento molte volte con numeri casuali; **confrontare** i due risultati, con i numeri e con un grafico; **commentare** la differenza, spiegando che dipende dalla fluttuazione casuale e che si riduce come 1/√N all'aumentare del numero N di simulazioni.

Le tre librerie

Tutto il lavoro si fa con tre librerie, che si importano all'inizio di ogni programma sempre con le stesse tre righe. **NumPy** gestisce i vettori di numeri (array) e genera numeri casuali; **scipy.stats** contiene tutte le distribuzioni notevoli e i test statistici; **matplotlib.pyplot** disegna i grafici.

```
import numpy as np                  # vettori, calcoli, numeri casuali
import scipy.stats as stats         # distribuzioni e test statistici
import matplotlib.pyplot as plt     # grafici
```

Gli array di NumPy

Un array è un vettore di numeri su cui le operazioni agiscono **elemento per elemento**, senza bisogno di cicli. È il concetto più importante di tutta la dispensa: invece di scrivere un ciclo che somma i valori uno alla volta, si scrive un'unica operazione sull'intero vettore.

```
x = np.array([2, 4, 4, 5, 7])       # un array costruito da una lista

print(x + 1)                        # somma 1 a ogni elemento
print(x ** 2)                       # eleva al quadrato ogni elemento
print(x.sum(), x.mean())            # somma e media di tutti gli elementi
print(len(x))                       # numero di elementi
print(x > 3)                        # confronto elemento per elemento
```

OUTPUT

```
[3 5 5 6 8]
[ 4 16 16 25 49]
22 4.4
5
[False  True  True  True  True]
```

L'ultimo comando restituisce un array di valori logici, True dove la condizione è vera e False dove è falsa. In un calcolo numerico True vale 1 e False vale 0: la **media** di un array logico è quindi la frazione di elementi che soddisfano la condizione, cioè una frequenza relativa. Questo trucco, che si userà di continuo, trasforma il conteggio dei casi favorevoli in una sola riga.

```
print((x > 3).mean())              # frazione di elementi maggiori di 3
print((x > 3).sum())               # quanti elementi sono maggiori di 3
```

OUTPUT

```
0.8
4
```

Altre funzioni utili sono np.arange(a, b), che produce gli interi da a fino a b escluso, np.linspace(a, b, n), che produce n numeri equidistanti tra a e b inclusi (serve per disegnare le curve), np.sqrt, np.exp, np.log per radice, esponenziale e logaritmo naturale applicati a ogni elemento, e np.cumsum per le somme cumulate.

Generare numeri casuali

I numeri casuali si generano con le funzioni di np.random. Il parametro size indica quanti numeri generare: con size=N si ottengono in un colpo solo N esperimenti indipendenti, senza scrivere cicli. Il comando np.random.seed fissa il seme del generatore, in modo che ogni esecuzione produca gli stessi numeri: è utile per poter ricontrollare i risultati.

COMANDO	CHE COSA GENERA
np.random.uniform(a, b, size=N)	N numeri uniformi nell'intervallo [a, b)
np.random.randint(a, b, size=N)	N interi da a a b − 1 (b escluso): per un dado randint(1, 7)
np.random.choice(v, size=N)	N estrazioni con reinserimento dagli elementi di v
np.random.choice(v, size=n, replace=False)	n estrazioni senza reinserimento
np.random.choice(v, size=N, p=prob)	estrazioni con le probabilità indicate in prob
np.random.binomial(n, p, size=N)	N valori da una binomiale Bin(n, p)
np.random.poisson(lam, size=N)	N valori da una Poisson di parametro lam
np.random.exponential(media, size=N)	N valori esponenziali con la media indicata, cioè 1/λ
np.random.normal(mu, sigma, size=N)	N valori da una normale con media mu e deviazione standard sigma
np.random.seed(s)	fissa il seme s per avere risultati riproducibili

ATTENZIONE

np.random.exponential vuole la **media** 1/λ, non il tasso λ: se arrivano in media 4 clienti al minuto si scrive np.random.exponential(1/4, size=N). Allo stesso modo np.random.normal vuole la **deviazione standard** σ, non la varianza σ^2.

```
np.random.seed(1)                          # seme fisso: risultati riproducibili
dadi = np.random.randint(1, 7, size=10)    # 10 lanci di un dado
print(dadi)
monete = np.random.choice(["T", "C"], size=8)   # 8 lanci di una moneta
print(monete)
```

OUTPUT

```
[6 4 5 1 2 4 6 1 1 2]
['T' 'C' 'C' 'T' 'T' 'C' 'T' 'T']
```

Le distribuzioni in scipy.stats

Ogni distribuzione notevole è un oggetto di scipy.stats che offre sempre gli stessi metodi, elencati nella tabella. Conviene impararli una volta per tutte: cambiando distribuzione cambiano solo il nome e i parametri.

METODO	CHE COSA CALCOLA	ESEMPIO
pmf(k, ...)	P(X = k), solo per le discrete	stats.binom.pmf(3, 10, 0.5)
pdf(x, ...)	la densità f(x), solo per le continue	stats.norm.pdf(0, 0, 1)
cdf(x, ...)	la ripartizione F(x) = P(X ≤ x)	stats.norm.cdf(1.96)
sf(x, ...)	la coda destra P(X > x) = 1 − F(x)	stats.norm.sf(1.96)
ppf(q, ...)	il quantile: il valore x con F(x) = q	stats.norm.ppf(0.975)
rvs(..., size=N)	N valori simulati	stats.expon.rvs(scale=2, size=5)
mean(...), var(...)	media e varianza teoriche	stats.poisson.mean(3)

DISTRIBUZIONE	OGGETTO IN SCIPY.STATS	NOTA SUI PARAMETRI
Binomiale Bin(n, p)	stats.binom(n, p)	
Poisson Pois(λ)	stats.poisson(lam)	
Geometrica Geom(p)	stats.geom(p)	conta le prove fino al primo successo (valori 1, 2, …)
Ipergeometrica	stats.hypergeom(Ntot, K, n)	Ntot elementi, K successi, n estrazioni
Uniforme U(a, b)	stats.uniform(a, b − a)	il secondo parametro è la lunghezza dell'intervallo
Esponenziale Exp(λ)	stats.expon(scale=1/lam)	scale è la media, non il tasso
Normale N(μ, σ^2)	stats.norm(mu, sigma)	il secondo parametro è σ, non σ^2
t di Student	stats.t(gl)	gl = gradi di libertà

DISTRIBUZIONE	OGGETTO IN SCIPY.STATS	NOTA SUI PARAMETRI
Chi quadro	stats.chi2(gl)	gl = gradi di libertà

ATTENZIONE

Con le distribuzioni discrete cdf(k) include k: cdf(2) è P(X ≤ 2), mentre pmf(2) è P(X = 2). Per "almeno k" si usa sf(k − 1), che vale P(X > k − 1) = P(X ≥ k). Confondere questi casi produce errori silenziosi, perché il codice funziona ma il numero è sbagliato.

```
# binomiale con n = 10 prove e p = 0.3
print(stats.binom.pmf(3, 10, 0.3))     # P(X = 3)
print(stats.binom.cdf(3, 10, 0.3))     # P(X <= 3)
print(stats.binom.sf(2, 10, 0.3))      # P(X >= 3) = P(X > 2)

# quantili della normale standard usati negli intervalli
print(stats.norm.ppf(0.975))           # quantile per il 95% bilaterale
print(stats.t.ppf(0.975, 9))           # lo stesso per la t con 9 gradi di libertà
```

OUTPUT

```
0.2668279319999998
0.6496107184000002
0.6172172136000003
1.959963984540054
2.262157162798205
```

I grafici

I grafici si costruiscono aggiungendo uno strato alla volta e si mostrano con plt.show(). I comandi che servono sono pochi.

COMANDO	CHE COSA DISEGNA
plt.hist(x, bins=30, density=True)	istogramma dei dati; density=True rende l'area totale uguale a 1
plt.bar(valori, altezze)	diagramma a barre, per dati discreti
plt.plot(x, y)	una linea, per esempio la densità teorica
plt.scatter(x, y)	una nuvola di punti, per la regressione
plt.boxplot(x)	box plot
plt.axvline(v)	una linea verticale, per esempio un valore critico
plt.xlabel, plt.ylabel, plt.title	etichette degli assi e titolo
plt.legend()	legenda, utile quando ci sono più curve

Il parametro density=True è indispensabile quando si vuole sovrapporre un istogramma di dati simulati a una densità teorica: senza di esso l'asse verticale conta le osservazioni, che dipendono da N, e le due cose non sono confrontabili.

Uno schema da cui partire

Il programma seguente riassume lo schema dei quattro passi su un esempio semplice: il numero di teste in 10 lanci di una moneta. Si calcola la media teorica, si simulano 10 000 esperimenti, si confrontano media teorica e simulata con il loro errore standard e si disegna il confronto tra le frequenze simulate e la densità binomiale.

```
np.random.seed(2)
n, p, N = 10, 0.5, 10000                  # prove, probabilità, simulazioni

teorica = n * p                           # passo 1: media teorica np
sim = np.random.binomial(n, p, size=N)    # passo 2: N esperimenti simulati
simulata = sim.mean()
es = sim.std(ddof=1) / np.sqrt(N)         # errore standard della media simulata
print("media teorica :", teorica)
print("media simulata:", round(simulata, 4), " errore standard:", round(es, 4))

k = np.arange(0, n + 1)                   # passo 3: confronto grafico
freq = [(sim == v).mean() for v in k]     # frequenze relative simulate
plt.bar(k - 0.2, freq, width=0.4, label="simulazione")
plt.bar(k + 0.2, stats.binom.pmf(k, n, p), width=0.4, label="teoria")
plt.legend()
plt.show()
```

OUTPUT

```
media teorica : 5.0
media simulata: 4.9679  errore standard: 0.0159
```

Il commento (passo 4) è che la media simulata, 4,968, dista da quella teorica di circa 0,03, cioè circa due errori standard: una differenza compatibile con la fluttuazione casuale, che aumentando N si ridurrebbe in proporzione a 1/√N. Un buon controllo da fare sempre è proprio questo: se il valore simulato dista dal teorico più di tre errori standard, c'è quasi certamente un errore nel codice o nella formula.

CAPITOLO 2

Probabilità e simulazione

Calcolare una probabilità con le formule e verificarla simulando l'esperimento: frequenze relative, calcolo combinatorio, estrazioni senza reinserimento, probabilità condizionata e teorema di Bayes, errore standard.

Stimare una probabilità con la frequenza relativa

Per la legge dei grandi numeri la frequenza relativa di un evento su N ripetizioni indipendenti tende alla sua probabilità. In Python questo si traduce in tre righe: si simulano N esperimenti in un array, si scrive la condizione che descrive l'evento, si fa la media dell'array logico ottenuto.

Esempio. Si lanciano due dadi. Qual è la probabilità che la somma sia 7? I casi possibili sono 36 e quelli favorevoli 6, cioè (1,6), (2,5), (3,4), (4,3), (5,2), (6,1): la probabilità teorica è 6/36 = 1/6.

```
np.random.seed(3)
N = 100000                              # numero di esperimenti simulati
d1 = np.random.randint(1, 7, size=N)    # primo dado
d2 = np.random.randint(1, 7, size=N)    # secondo dado
somma = d1 + d2

teorica = 6 / 36
stimata = (somma == 7).mean()           # frequenza relativa dell'evento
print("teorica:", round(teorica, 4), " stimata:", round(stimata, 4))
```

OUTPUT

```
teorica: 0.1667  stimata: 0.1673
```

Le operazioni logiche tra eventi si scrivono con tre simboli: & per l'intersezione ("e"), | per l'unione ("oppure") e ~ per il contrario ("non"). Ogni condizione va racchiusa tra parentesi, perché questi operatori hanno precedenza più alta dei confronti. Le parole and, or e not di Python non funzionano sugli array.

```
pari = (d1 % 2 == 0) & (d2 % 2 == 0)    # entrambi i dadi pari
almeno_un_6 = (d1 == 6) | (d2 == 6)     # almeno un sei
print("P(entrambi pari):", pari.mean(), "  teorica:", 9 / 36)
print("P(almeno un 6)  :", almeno_un_6.mean(), "  teorica:", round(11 / 36, 5))
print("P(nessun 6)     :", (~almeno_un_6).mean(), "  teorica:", round(25 / 36, 5))
```

OUTPUT

```
P(entrambi pari): 0.24971   teorica: 0.25
P(almeno un 6)  : 0.30276   teorica: 0.30556
P(nessun 6)     : 0.69724   teorica: 0.69444
```

ATTENZIONE

Una probabilità stimata si ottiene sempre dividendo il numero di esperimenti favorevoli per il **numero di esperimenti simulati** N, non per il numero di oggetti del problema (carte, palline, persone). Se il risultato esce dall'intervallo [0, 1] l'errore è quasi sempre questo. Ricordare anche che randint(a, b) esclude b: per un dado si scrive randint(1, 7).

Quanto è precisa la stima: l'errore standard

Una simulazione dà sempre una stima, mai il valore esatto. La frequenza relativa $\hat{p}$ su N esperimenti è una media di N variabili di Bernoulli, quindi ha deviazione standard $\sqrt{p(1-p)/N}$, che si chiama **errore standard**. Nella pratica si sostituisce p con la stima $\hat{p}$. Lo scarto tra stima e valore teorico va misurato in unità di errore standard: se è inferiore a due o tre errori standard la simulazione conferma la teoria.

```
np.random.seed(4)
p = 1 / 6                                    # probabilità teorica di somma 7
for N in [100, 1000, 10000, 100000]:
    somma = np.random.randint(1, 7, size=N) + np.random.randint(1, 7, size=N)
    stima = (somma == 7).mean()
    es = np.sqrt(stima * (1 - stima) / N)     # errore standard stimato
    print(f"N = {N:6d}  stima = {stima:.4f}  errore standard = {es:.4f}"
          f"  scarto/es = {abs(stima - p) / es:.2f}")
```

OUTPUT

```
N =    100  stima = 0.1900  errore standard = 0.0392  scarto/es = 0.59
N =   1000  stima = 0.1730  errore standard = 0.0120  scarto/es = 0.53
N =  10000  stima = 0.1684  errore standard = 0.0037  scarto/es = 0.46
N = 100000  stima = 0.1662  errore standard = 0.0012  scarto/es = 0.36
```

La tabella mostra il comportamento tipico: l'errore standard si riduce di circa $\sqrt{10} \approx 3{,}16$ volte ogni volta che N diventa dieci volte più grande, e lo scarto dalla teoria resta dell'ordine di un errore standard. È la legge $1/\sqrt{N}$: per guadagnare una cifra decimale servono cento volte più esperimenti.

Calcolo combinatorio ed estrazioni senza reinserimento

I coefficienti binomiali si calcolano con math.comb(n, k), mentre il fattoriale con math.factorial(n). Per simulare estrazioni senza reinserimento si usa np.random.choice con replace=False, che estrae n elementi distinti da un insieme; poiché ogni estrazione è un esperimento a sé, qui serve un ciclo.

Esempio. Un'urna contiene 4 palline rosse e 6 blu e se ne estraggono 3 senza reinserimento. La probabilità di estrarre esattamente 2 rosse è $C(4, 2)\cdot C(6, 1)/C(10, 3) = 36/120 = 0{,}3$, che coincide con la densità ipergeometrica.

```
teorica = comb(4, 2) * comb(6, 1) / comb(10, 3)    # casi favorevoli / possibili
print("teorica (formula)      :", teorica)
print("teorica (ipergeometrica):", stats.hypergeom.pmf(2, 10, 4, 3))

np.random.seed(5)
urna = np.array(["R"] * 4 + ["B"] * 6)   # 4 rosse e 6 blu
N = 20000
successi = 0
for i in range(N):
    estratte = np.random.choice(urna, size=3, replace=False)  # senza reinserimento
    if (estratte == "R").sum() == 2:                          # esattamente 2 rosse
        successi += 1
print("stimata                :", successi / N)
```

OUTPUT

```
teorica (formula)      : 0.3
teorica (ipergeometrica): 0.3
stimata                : 0.3011
```

ESEMPIO: COMPLEANNI COINCIDENTI

In un gruppo di n persone, qual è la probabilità che almeno due compiano gli anni lo stesso giorno (anno di 365 giorni equiprobabili)? Conviene passare al contrario: la probabilità che i compleanni siano tutti diversi è (365/365)·(364/365)·…·((365 − n + 1)/365), quindi la probabilità cercata è 1 meno questo prodotto. Per n = 23 vale circa 0,507, un risultato che sorprende quasi tutti.

```
n = 23
tutti_diversi = np.prod((365 - np.arange(n)) / 365)    # prodotto dei rapporti
print("teorica:", round(1 - tutti_diversi, 4))

np.random.seed(6)
N = 20000
coincidenze = 0
for i in range(N):
    giorni = np.random.randint(1, 366, size=n)         # compleanni del gruppo
    if len(np.unique(giorni)) < n:                     # almeno una ripetizione
        coincidenze += 1
print("stimata:", coincidenze / N)
```

OUTPUT

```
teorica: 0.5073
stimata: 0.5042
```

Probabilità condizionata e teorema di Bayes

Per stimare una probabilità condizionata P(A | B) con una simulazione si restringe l'attenzione ai soli esperimenti in cui B si è verificato e, tra questi, si calcola la frazione in cui si è verificato anche A. In NumPy basta selezionare gli elementi di un array con una condizione tra parentesi quadre: a[b] contiene solo gli elementi di a in corrispondenza dei quali b è True.

Esempio. Riprendiamo il filtro antispam della dispensa di teoria: il 20% delle email è spam, il filtro segnala il 95% dello spam e il 4% delle email legittime. Con le probabilità totali P(segnalata) = 0,222 e con Bayes P(spam | segnalata) ≈ 0,856.

```
p_spam, p_seg_spam, p_seg_legit = 0.20, 0.95, 0.04

# soluzione teorica
p_seg = p_seg_spam * p_spam + p_seg_legit * (1 - p_spam)   # probabilità totali
bayes = p_seg_spam * p_spam / p_seg                         # teorema di Bayes
print("teorica: P(segnalata) =", round(p_seg, 4),
      " P(spam | segnalata) =", round(bayes, 4))

# simulazione di N email
np.random.seed(7)
N = 100000
spam = np.random.uniform(0, 1, size=N) < p_spam             # True se è spam
u = np.random.uniform(0, 1, size=N)
segnalata = np.where(spam, u < p_seg_spam, u < p_seg_legit) # esito del filtro
print("stimata: P(segnalata) =", round(segnalata.mean(), 4),
      " P(spam | segnalata) =", round(spam[segnalata].mean(), 4))
```

OUTPUT

```
teorica: P(segnalata) = 0.222  P(spam | segnalata) = 0.8559
stimata: P(segnalata) = 0.2227  P(spam | segnalata) = 0.8599
```

La funzione np.where(c, a, b) sceglie elemento per elemento a dove la condizione c è vera e b dove è falsa: qui applica alle email di spam la probabilità di segnalazione 0,95 e alle altre 0,04. L'espressione spam[segnalata] seleziona le sole email segnalate, e la sua media è la frazione di spam tra queste, cioè la stima di P(spam | segnalata).

DA RICORDARE

Per simulare un evento di probabilità p si genera un numero uniforme u su [0, 1) e si dichiara che l'evento si è verificato se u < p. Con size=N si simulano N eventi indipendenti in una riga: np.random.uniform(0, 1, size=N) < p.

CAPITOLO 3

Distribuzioni: teoria e simulazione a confronto

Come verificare che dati simulati seguano una distribuzione teorica: barre affiancate per le discrete, istogramma e densità per le continue, confronto dei momenti, QQ plot, teorema del limite centrale e legge 1/√N.

La ricetta in tre passi

Il confronto tra teoria e simulazione si costruisce sempre allo stesso modo. Prima si dichiarano la distribuzione teorica e i suoi parametri, scrivendo anche media e varianza attese. Poi si simulano i dati e si calcolano le grandezze empiriche corrispondenti. Infine si confrontano le due cose con un grafico e con i numeri, e si commenta se le differenze sono compatibili con la fluttuazione casuale. Due numeri non saranno mai identici: la domanda giusta non è se sono uguali, ma se lo scarto è piccolo rispetto all'errore standard.

Distribuzioni discrete: barre affiancate

Esempio. Le richieste che arrivano a un server in un secondo seguono una Poisson con $\lambda = 3$. Simuliamo 5000 secondi e confrontiamo le frequenze relative osservate con la densità teorica $P(X = k) = 3^k e^{-3}/k!$.

```
np.random.seed(10)
lam, N = 3, 5000
x = np.random.poisson(lam, size=N)          # N secondi simulati

k = np.arange(0, 11)                        # valori da confrontare
freq = np.array([(x == v).mean() for v in k])   # frequenze relative
teo = stats.poisson.pmf(k, lam)                 # probabilità teoriche
for v in range(0, 7):
    print(f"k = {v}:  simulata {freq[v]:.4f}   teorica {teo[v]:.4f}")

plt.bar(k - 0.2, freq, width=0.4, label="simulazione")
plt.bar(k + 0.2, teo, width=0.4, label="teoria")
plt.xlabel("richieste al secondo"); plt.legend(); plt.show()
```

OUTPUT

```
k = 0:  simulata 0.0568   teorica 0.0498
k = 1:  simulata 0.1538   teorica 0.1494
k = 2:  simulata 0.2290   teorica 0.2240
k = 3:  simulata 0.2138   teorica 0.2240
k = 4:  simulata 0.1626   teorica 0.1680
k = 5:  simulata 0.1068   teorica 0.1008
k = 6:  simulata 0.0452   teorica 0.0504
```

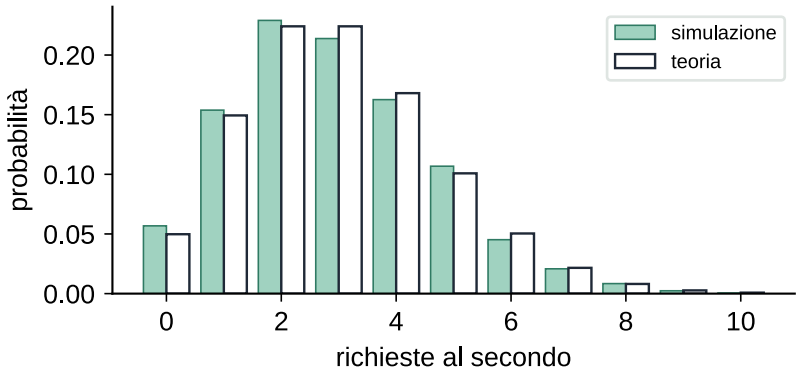

Figura 3.1. Frequenze simulate e probabilità teoriche della Poisson con λ = 3: le barre sono affiancate, sfalsate di 0,2.

Per le distribuzioni discrete si usano le barre e non l'istogramma, perché i valori sono separati. Sfalsare le barre di 0,2 a sinistra e a destra permette di vederle una accanto all'altra. Ogni frequenza simulata ha un errore standard di circa √(p(1 − p)/N), dell'ordine di 0,006 per le probabilità vicine a 0,2: le differenze visibili in tabella sono di questa entità.

Distribuzioni continue: istogramma e densità

Esempio. I clienti arrivano a uno sportello in media 4 al minuto, quindi il tempo tra due arrivi è esponenziale con λ = 4 e media 1/4 di minuto. Calcoliamo P(X > 0,5) con la formula e con la simulazione, e sovrapponiamo l'istogramma dei tempi simulati alla densità teorica.

```
np.random.seed(11)
lam, N = 4, 10000
t = np.random.exponential(1 / lam, size=N)    # attenzione: numpy vuole la media 1/lam

print("P(X > 0.5) teorica:", round(np.exp(-lam * 0.5), 4))       # e^(-2)
print("P(X > 0.5) scipy  :", round(stats.expon.sf(0.5, scale=1 / lam), 4))
print("P(X > 0.5) stimata:", round((t > 0.5).mean(), 4))
print("media teorica e simulata:", 1 / lam, round(t.mean(), 4))

xs = np.linspace(0, t.max(), 300)
plt.hist(t, bins=40, density=True, label="simulazione")
plt.plot(xs, stats.expon.pdf(xs, scale=1 / lam), label="densità teorica")
plt.xlabel("tempo tra due arrivi (minuti)"); plt.legend(); plt.show()
```

OUTPUT

```
P(X > 0.5) teorica: 0.1353
P(X > 0.5) scipy  : 0.1353
P(X > 0.5) stimata: 0.1336
media teorica e simulata: 0.25 0.2472
```

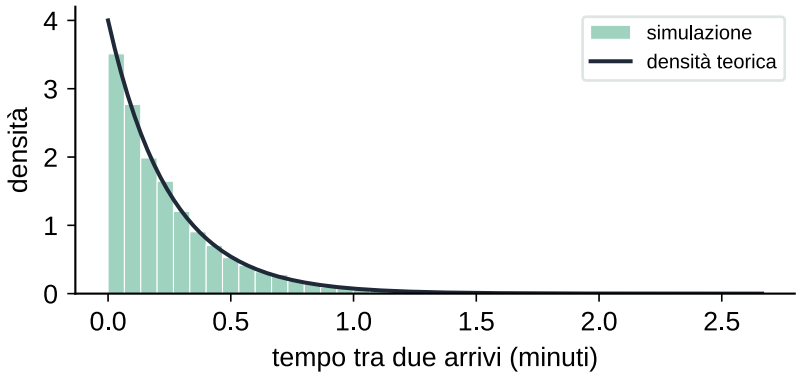

Figura 3.2. Istogramma di 10 000 tempi simulati e densità esponenziale con λ = 4.

DA RICORDARE

Perché il confronto abbia senso servono tre cose: density=True nell'istogramma, così che l'area delle barre valga 1 come quella sotto la densità; gli stessi parametri nella simulazione e nella curva teorica (per l'esponenziale scale=1/λ in entrambe); un numero di classi ragionevole, tra 30 e 50, perché con poche classi la forma non si vede e con troppe domina il rumore.

Confrontare i momenti

Oltre al grafico conviene confrontare i numeri: media, varianza, asimmetria e curtosi. Le funzioni stats.skew e stats.kurtosis calcolano asimmetria e curtosi dei dati; la seconda restituisce già la curtosi meno 3, che vale 0 per la normale. I valori teorici si ottengono con il metodo stats(moments="mvsk") della distribuzione, che restituisce media, varianza, asimmetria e curtosi.

```
m, v, sk, ku = stats.expon.stats(scale=1 / lam, moments="mvsk")   # valori teorici
print(f"{'':12}{'teoria':>10}{'simulato':>10}")
print(f"{'media':12}{float(m):10.4f}{t.mean():10.4f}")
print(f"{'varianza':12}{float(v):10.4f}{t.var():10.4f}")
print(f"{'asimmetria':12}{float(sk):10.4f}{stats.skew(t):10.4f}")
print(f"{'curtosi':12}{float(ku):10.4f}{stats.kurtosis(t):10.4f}")
```

OUTPUT

```
                teoria  simulato
media           0.2500    0.2472
varianza        0.0625    0.0608
asimmetria      2.0000    2.0436
curtosi         6.0000    6.6131
```

Media e varianza coincidono quasi esattamente; asimmetria e curtosi, che dipendono dai valori estremi, oscillano di più. È un fatto generale: i momenti di ordine alto si stimano con meno precisione e richiedono campioni più grandi.

Il QQ plot

Il **QQ plot** (grafico quantile contro quantile) mette sull'asse orizzontale i quantili della distribuzione teorica e su quello verticale i dati ordinati. Se i dati seguono la distribuzione, i punti si dispongono lungo una retta; curvature sistematiche, soprattutto alle estremità, indicano code più pesanti o più leggere del previsto. Si disegna con stats.probplot, indicando la distribuzione di riferimento e, se noti, i suoi parametri in sparams.

```
ris = stats.probplot(t, dist=stats.expon, sparams=(0, 1 / lam), plot=plt)
plt.show()
pendenza, intercetta, r = ris[1]          # retta di riferimento e correlazione
print("pendenza:", round(pendenza, 4), " intercetta:", round(intercetta, 4),
      " r:", round(r, 4))
```

OUTPUT

```
pendenza: 0.9872  intercetta: 0.0005  r: 0.9995
```

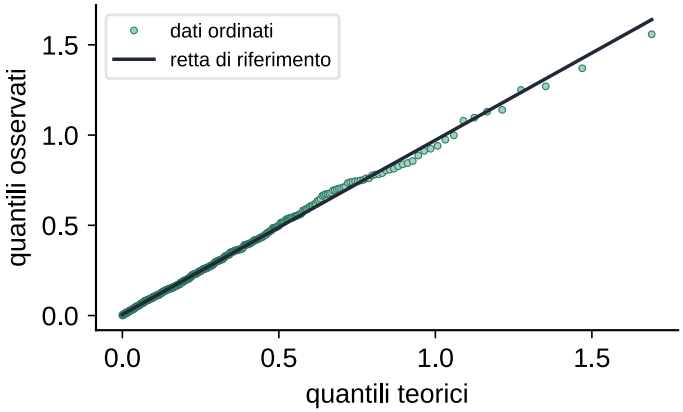

Figura 3.3. QQ plot dei tempi simulati rispetto all'esponenziale con λ = 4 (per leggibilità sono mostrati 600 punti): i punti seguono la retta.

Passando i parametri della distribuzione, la retta attesa è la bisettrice: pendenza vicina a 1, intercetta vicina a 0 e correlazione r vicina a 1 confermano il modello. Se invece non si passano i parametri (sparams omesso, cioè esponenziale con λ = 1), i punti stanno ancora su una retta ma con pendenza pari alla media 1/λ: è un modo per stimare il parametro dai dati. Usare la distribuzione sbagliata, per esempio dist="norm" con dati esponenziali, produce invece una curva evidente.

Il teorema del limite centrale in pratica

Esempio. La media di 30 lanci di un dado ha media 3,5 e varianza (35/12)/30. Per il teorema del limite centrale è approssimativamente normale, anche se il singolo lancio è uniforme discreto. Simuliamo 10 000 medie con una matrice: ogni riga è un gruppo di 30 lanci e mean(axis=1) fa la media di ogni riga.

```
np.random.seed(12)
n, N = 30, 10000
lanci = np.random.randint(1, 7, size=(N, n))    # N righe di n lanci ciascuna
medie = lanci.mean(axis=1)                       # media di ogni riga

mu, sigma = 3.5, np.sqrt(35 / 12 / n)            # parametri della normale limite
print("media e dev. standard simulate:", round(medie.mean(), 4), round(medie.std(), 4))
print("media e dev. standard teoriche:", mu, round(sigma, 4))
print("P(media > 4) simulata:", round((medie > 4).mean(), 4))
print("normale senza correzione:", round(stats.norm.sf(4, mu, sigma), 4))
print("normale con correzione  :", round(stats.norm.sf(4 + 1 / 60, mu, sigma), 4))
```

OUTPUT

```
media e dev. standard simulate: 3.4968 0.3076
media e dev. standard teoriche: 3.5 0.3118
P(media > 4) simulata: 0.0451
normale senza correzione: 0.0544
normale con correzione  : 0.0488
```

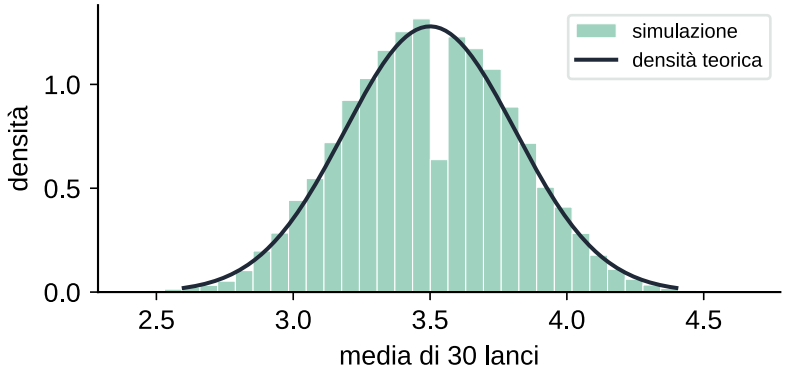

Figura 3.4. Istogramma delle 10 000 medie di 30 lanci e densità normale prevista dal teorema del limite centrale.

L'istogramma ha dei vuoti perché la media di 30 lanci può assumere solo multipli di 1/30, ma la sua forma segue bene la campana. Proprio perché la media è discreta, la probabilità di superare 4 si approssima meglio con la **correzione di continuità**: l'evento "media maggiore di 4" equivale a "somma almeno 121", cioè, per la variabile continua, a "media maggiore di 4 + 1/60" (metà del passo 1/30). Con la correzione la normale dà circa 0,049, molto vicino al valore simulato.

La legge 1/√N

Il singolo scarto tra stima e valore vero è casuale: due simulazioni con lo stesso N danno scarti diversi. Ciò che segue una legge precisa è lo scarto **medio** su molte ripetizioni, che decresce come 1/√N. Lo verifichiamo stimando P(X > 0,5) dell'esponenziale di prima con N crescente, ripetendo ogni stima 200 volte.

```
np.random.seed(13)
vero = np.exp(-2)                          # P(X > 0.5) con lambda = 4
lista_N = [100, 1000, 10000, 100000]
errori = []
for N in lista_N:
    e = [abs((np.random.exponential(0.25, size=N) > 0.5).mean() - vero)
         for i in range(200)]              # 200 ripetizioni della stima
    errori.append(np.mean(e))
    print(f"N = {N:6d}  errore medio = {np.mean(e):.5f}")
print("rapporti tra errori consecutivi:", np.round(np.array(errori[:-1]) /
      np.array(errori[1:]), 2), " atteso:", round(np.sqrt(10), 2))
```

OUTPUT

```
N =    100  errore medio = 0.02639
N =   1000  errore medio = 0.00976
N =  10000  errore medio = 0.00262
N = 100000  errore medio = 0.00088
rapporti tra errori consecutivi: [2.7  3.72 2.98]  atteso: 3.16
```

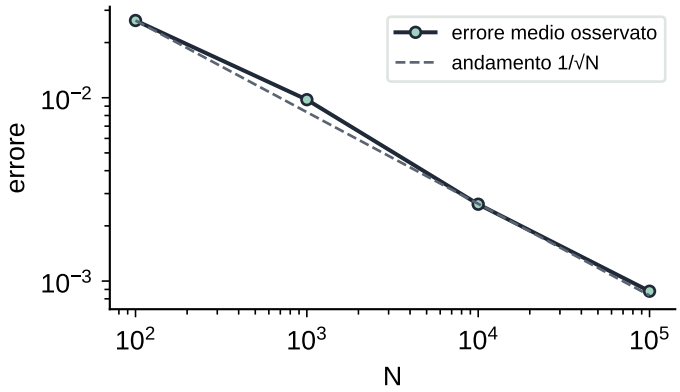

Figura 3.5. Errore medio in funzione di N, in scala logaritmica: i punti seguono la retta di pendenza 1/√N.

Ogni volta che N diventa dieci volte più grande l'errore medio si divide per circa √10 ≈ 3,16. La conseguenza pratica è la regola già vista: per dimezzare l'errore servono quattro volte più simulazioni, per guadagnare una cifra decimale cento volte di più.

CAPITOLO 4

Statistica descrittiva

Tabelle di frequenze per valori singoli e per classi, indici di posizione, dispersione e forma, il parametro ddof e i grafici descrittivi.

Tabella delle frequenze da dati grezzi

Quando i dati sono un elenco di osservazioni, la tabella delle frequenze si ottiene con np.unique(dati, return_counts=True), che restituisce i valori distinti ordinati e quante volte compare ciascuno. Da lì le frequenze relative si ottengono dividendo per N, le cumulate con np.cumsum e la moda come valore con la frequenza massima.

Esempio. Il numero di bug trovati in 20 moduli di un progetto (gli stessi dati della dispensa di teoria).

```
bug = np.array([1, 0, 2, 1, 3, 1, 0, 2, 4, 1, 2, 0, 1, 3, 2, 1, 0, 2, 3, 1])
valori, freq = np.unique(bug, return_counts=True)   # valori distinti e frequenze
N = len(bug)
rel = freq / N                                      # frequenze relative
cum = np.cumsum(freq)                               # frequenze cumulate
print("valore  ass.  rel.  cum.")
for v, f, r, c in zip(valori, freq, rel, cum):
    print(f"{v:6d} {f:5d} {r:5.2f} {c:5d}")
print("moda:", valori[freq.argmax()])
```

OUTPUT

```
valore  ass.  rel.  cum.
     0     4  0.20     4
     1     7  0.35    11
     2     5  0.25    16
     3     3  0.15    19
     4     1  0.05    20
moda: 1
```

Se i dati sono già raggruppati, cioè se si conoscono le modalità x_i e le frequenze f_i, si costruiscono direttamente i due array. Media e varianza si calcolano pesando con le frequenze, oppure si ricostruiscono i dati grezzi con np.repeat(x, f), che ripete ogni valore tante volte quanto indica la sua frequenza.

```
x = np.array([0, 1, 2, 3, 4])           # modalità
f = np.array([4, 7, 5, 3, 1])           # frequenze assolute
N = f.sum()
media = (x * f).sum() / N               # media pesata con le frequenze
var_emp = ((x - media) ** 2 * f).sum() / N
print("media:", media, " varianza empirica:", var_emp)

dati = np.repeat(x, f)                  # ricostruisce i 20 dati grezzi
print("stessi valori:", dati.mean(), dati.var())
```

OUTPUT

```
media: 1.5  varianza empirica: 1.25
stessi valori: 1.5 1.25
```

Raggruppamento in classi

Per dati continui si usano le classi. La funzione np.histogram(dati, bins=estremi) conta quanti dati cadono in ciascuna classe, dati gli estremi delle classi in ordine crescente; ogni classe include l'estremo sinistro ed esclude il destro, tranne l'ultima che li include entrambi. Le classi possono avere ampiezze diverse: in tal caso per l'istogramma serve la densità di frequenza.

Esempio. I tempi di risposta, in millisecondi, di 30 richieste a un servizio web.

```
tempi = np.array([212, 198, 230, 187, 245, 205, 219, 201, 226, 194,
                  238, 209, 215, 251, 190, 222, 207, 233, 199, 214,
                  228, 203, 196, 241, 217, 210, 225, 208, 236, 312])
estremi = np.array([180, 200, 220, 240, 260, 320])     # estremi delle classi
freq, _ = np.histogram(tempi, bins=estremi)            # frequenze per classe
ampiezze = np.diff(estremi)                            # ampiezze delle classi
centri = (estremi[:-1] + estremi[1:]) / 2              # valori centrali
for i in range(len(freq)):
    print(f"[{estremi[i]}, {estremi[i+1]})  freq {freq[i]:2d}  "
          f"densità {freq[i] / ampiezze[i]:.2f}")
print("media dai dati      :", round(tempi.mean(), 2))
print("media dalle classi  :", round((centri * freq).sum() / freq.sum(), 2))
```

OUTPUT

```
[180, 200)  freq  6  densità 0.30
[200, 220)  freq 12  densità 0.60
[220, 240)  freq  8  densità 0.40
[240, 260)  freq  3  densità 0.15
[260, 320)  freq  1  densità 0.02
media dai dati      : 219.03
media dalle classi  : 218.0
```

La media calcolata dai valori centrali delle classi è solo un'approssimazione di quella vera, perché sostituisce ogni dato con il centro della sua classe. Quando i dati grezzi sono disponibili conviene sempre usarli.

Gli indici in una schermata

Il programma seguente calcola tutti gli indici descrittivi principali. La funzione np.percentile(dati, [25, 75]) restituisce primo e terzo quartile; np.median la mediana.

```
media = np.mean(tempi)
mediana = np.median(tempi)
q1, q3 = np.percentile(tempi, [25, 75])      # primo e terzo quartile
var_emp = np.var(tempi)                      # divide per N
var_camp = np.var(tempi, ddof=1)             # divide per N - 1
s_camp = np.std(tempi, ddof=1)               # deviazione standard campionaria
print(f"media {media:.2f}   mediana {mediana:.2f}")
print(f"Q1 {q1:.2f}   Q3 {q3:.2f}   IQR {q3 - q1:.2f}")
print(f"minimo {tempi.min()}   massimo {tempi.max()}")
print(f"varianza empirica {var_emp:.2f}   varianza campionaria {var_camp:.2f}")
print(f"deviazione standard {s_camp:.2f}   CV {s_camp / media:.3f}")
print(f"asimmetria {stats.skew(tempi):.3f}   curtosi {stats.kurtosis(tempi):.3f}")
```

OUTPUT

```
media 219.03   mediana 214.50
Q1 203.50   Q3 229.50   IQR 26.00
minimo 187   massimo 312
varianza empirica 571.70   varianza campionaria 591.41
deviazione standard 24.32   CV 0.111
asimmetria 1.855   curtosi 5.135
```

ATTENZIONE

np.var e np.std dividono per N (varianza empirica), a meno di indicare ddof=1, che fa dividere per N − 1 (varianza campionaria, lo stimatore non distorto). ddof significa "delta degrees of freedom", i gradi di libertà sottratti. Si usa ddof=0 quando si vuole descrivere i dati, ddof=1 quando si vuole stimare la varianza della popolazione. Attenzione anche al fatto che alcune funzioni di scipy.stats usano già N − 1: confrontando pacchetti diversi i numeri possono non tornare per questo motivo.

Il valore 312 ms è anomalo: alza la media, rende positiva e grande l'asimmetria e fa crescere la curtosi, mentre la mediana resta quasi invariata. È l'effetto dei valori anomali descritto nella teoria.

I grafici descrittivi

L'istogramma si disegna con plt.hist; con classi di ampiezza diversa conviene aggiungere density=True, così che l'altezza sia la densità di frequenza. Il box plot si disegna con plt.boxplot, che segna automaticamente come punti isolati i valori oltre 1,5 volte la distanza interquartile dalla scatola. Per un carattere qualitativo o discreto si usa plt.bar sulle frequenze.

```
plt.hist(tempi, bins=estremi, density=True)   # istogramma per classi
plt.xlabel("tempo di risposta (ms)")
plt.show()

plt.boxplot(tempi, vert=False)                 # box plot orizzontale
plt.show()

valori_bug, freq_bug = np.unique(bug, return_counts=True)
plt.bar(valori_bug, freq_bug)                  # diagramma a barre
plt.xlabel("bug per modulo")
plt.show()
```

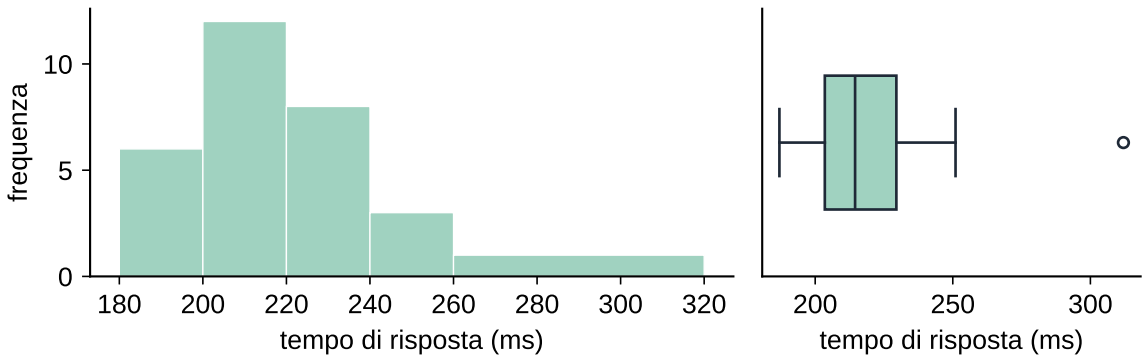

Figura 4.1. Istogramma per classi e box plot dei tempi di risposta: il valore 312 ms compare come punto isolato.

Il box plot mostra in un colpo d'occhio la mediana (linea nella scatola), la metà centrale dei dati (la scatola, da Q_1 a Q_3), l'estensione dei dati non anomali (i baffi) e i valori anomali. È particolarmente utile per confrontare più gruppi di dati affiancandoli.

CAPITOLO 5

Stimatori e intervalli di confidenza

Stime puntuali di media, varianza e proporzione, verifica per simulazione della non distorsione, intervalli di confidenza per media, proporzione e varianza, e significato del livello di confidenza.

Stime puntuali

Le stime puntuali si calcolano con le funzioni già viste: la media campionaria con np.mean, la varianza campionaria (lo stimatore non distorto) con np.var(dati, ddof=1), la deviazione standard con np.std(dati, ddof=1), una proporzione come media di un array logico.

Esempio. Dieci misure del tempo di risposta di un servizio, in millisecondi.

```
dati = np.array([212, 245, 228, 251, 219, 236, 207, 240, 231, 225])
n = len(dati)
media = dati.mean()                  # stima della media
s2 = dati.var(ddof=1)                # stima non distorta della varianza
s = dati.std(ddof=1)                 # deviazione standard campionaria
p_lenti = (dati > 240).mean()        # stima della proporzione di richieste lente
print(f"n = {n}  media = {media:.2f}  s2 = {s2:.2f}  s = {s:.2f}")
print(f"proporzione oltre 240 ms = {p_lenti:.2f}")
```

OUTPUT

```
n = 10  media = 229.40  s2 = 200.27  s = 14.15
proporzione oltre 240 ms = 0.20
```

Verificare la non distorsione con una simulazione

Che la varianza empirica, divisa per n, sia distorta e che la varianza campionaria, divisa per n − 1, non lo sia si può verificare direttamente: si estraggono moltissimi campioni da una popolazione nota, si calcolano i due stimatori su ciascun campione e se ne fa la media. Uno stimatore non distorto ha media uguale al parametro vero.

Esempio. Popolazione normale con μ = 10 e σ^2 = 4, campioni di n = 5 elementi. La teoria prevede che la varianza empirica abbia media $(n-1)\sigma^2/n$ = 3,2 e quella campionaria media 4.

```
np.random.seed(20)
mu, sigma, n, R = 10, 2, 5, 100000          # R = numero di campioni simulati
campioni = np.random.normal(mu, sigma, size=(R, n))   # ogni riga è un campione

media_camp = campioni.mean(axis=1)          # media di ogni campione
var_emp = campioni.var(axis=1)              # divide per n
var_camp = campioni.var(axis=1, ddof=1)     # divide per n - 1

print("media delle medie campionarie :", round(media_camp.mean(), 4), " vero:", mu)
print("media delle varianze empiriche:", round(var_emp.mean(), 4), " atteso:", 3.2)
print("media delle varianze campionarie:", round(var_camp.mean(), 4), " vero:", 4)
print("varianza della media campionaria:", round(media_camp.var(), 4),
      " teorica sigma^2/n:", sigma**2 / n)
```

OUTPUT

```
media delle medie campionarie : 10.0052  vero: 10
media delle varianze empiriche: 3.2001  atteso: 3.2
media delle varianze campionarie: 4.0001  vero: 4
varianza della media campionaria: 0.805  teorica sigma^2/n: 0.8
```

I numeri confermano la teoria: la media campionaria e la varianza campionaria sono centrate sui valori veri, la varianza empirica sottostima la varianza di un fattore 4/5, e la media campionaria ha varianza σ^2/n.

Intervallo per la media con varianza nota

L'intervallo di livello $1 - \alpha$ è $\bar{x} \pm z_{1-\alpha/2}\, \sigma/\sqrt{n}$, dove il quantile si calcola con stats.norm.ppf(1 − α/2). Lo stesso intervallo si ottiene con stats.norm.interval(1 − α, loc=x̄, scale=σ/√n), utile come controllo.

Esempio. La durata di 36 batterie ha media 12,4 ore, con σ = 1,8 ore nota.

```
xbar, sigma, n, alpha = 12.4, 1.8, 36, 0.05
z = stats.norm.ppf(1 - alpha / 2)            # quantile 1.96
margine = z * sigma / np.sqrt(n)             # margine di errore
print(f"z = {z:.4f}  margine = {margine:.4f}")
print(f"IC al 95%: [{xbar - margine:.3f}, {xbar + margine:.3f}]")
ic = stats.norm.interval(0.95, loc=xbar, scale=sigma / np.sqrt(n))
print("controllo:", np.round(ic, 3))
```

OUTPUT

```
z = 1.9600  margine = 0.5880
IC al 95%: [11.812, 12.988]
controllo: [11.812 12.988]
```

Intervallo per la media con varianza non nota

Se σ non è nota si usa la deviazione standard campionaria s e il quantile della t di Student con n − 1 gradi di libertà, stats.t.ppf(1 − α/2, n − 1). Il controllo si fa con stats.t.interval(1 − α, n − 1, loc=x̄, scale=s/√n).

```
alpha = 0.05
n = len(dati)
xbar, s = dati.mean(), dati.std(ddof=1)
t = stats.t.ppf(1 - alpha / 2, n - 1)        # quantile della t con n-1 gradi
margine = t * s / np.sqrt(n)
print(f"t = {t:.4f}  margine = {margine:.3f}")
print(f"IC al 95%: [{xbar - margine:.2f}, {xbar + margine:.2f}]")
ic = stats.t.interval(0.95, n - 1, loc=xbar, scale=s / np.sqrt(n))
print("controllo:", np.round(ic, 2))
```

OUTPUT

```
t = 2.2622  margine = 10.123
IC al 95%: [219.28, 239.52]
controllo: [219.28 239.52]
```

DA RICORDARE

La scelta tra normale e t dipende da una sola domanda: σ è data dal problema oppure va stimata dai dati? Se è data si usa stats.norm.ppf; se la si calcola con np.std(dati, ddof=1) si usa stats.t.ppf con n − 1 gradi di libertà.

Intervallo per una proporzione e per la varianza

Per una proporzione con n grande l'intervallo è $\hat{p} \pm z_{1-\alpha/2} \sqrt{\hat{p}(1-\hat{p})/n}$. Per la varianza di una popolazione normale si usano i due quantili della chi quadro con n − 1 gradi di libertà, ricordando che quello di ordine 1 − α/2 va al denominatore dell'estremo sinistro.

```
# proporzione: 92 utenti su 400 usano la modalità scura
k, n, alpha = 92, 400, 0.05
p = k / n
z = stats.norm.ppf(1 - alpha / 2)
m = z * np.sqrt(p * (1 - p) / n)
print(f"proporzione: p = {p:.3f}  IC al 95% [{p - m:.3f}, {p + m:.3f}]")

# varianza dei dieci tempi di risposta
n = len(dati)
s2 = dati.var(ddof=1)
chi_alto = stats.chi2.ppf(1 - alpha / 2, n - 1)   # quantile 0.975
chi_basso = stats.chi2.ppf(alpha / 2, n - 1)      # quantile 0.025
lo, hi = (n - 1) * s2 / chi_alto, (n - 1) * s2 / chi_basso
print(f"varianza: s2 = {s2:.2f}  IC al 95% [{lo:.2f}, {hi:.2f}]")
print(f"deviazione standard: IC al 95% [{np.sqrt(lo):.2f}, {np.sqrt(hi):.2f}]")
```

OUTPUT

```
proporzione: p = 0.230  IC al 95% [0.189, 0.271]
varianza: s2 = 200.27  IC al 95% [94.75, 667.46]
deviazione standard: IC al 95% [9.73, 25.84]
```

Che cosa significa il 95%

Il livello di confidenza riguarda il procedimento: costruendo l'intervallo su moltissimi campioni diversi, circa il 95% degli intervalli contiene il valore vero. Lo si vede simulando 100 campioni da una popolazione nota e contando gli intervalli che contengono la media vera.

```
np.random.seed(31)
mu, sigma, n = 50, 8, 25                    # popolazione nota
z = stats.norm.ppf(0.975)
basso, alto = [], []
for i in range(100):                         # 100 campioni indipendenti
    x = np.random.normal(mu, sigma, size=n)
    m = z * sigma / np.sqrt(n)
    basso.append(x.mean() - m)
    alto.append(x.mean() + m)
basso, alto = np.array(basso), np.array(alto)
contengono = (basso <= mu) & (mu <= alto)
print("intervalli che contengono mu:", contengono.sum(), "su 100")
```

OUTPUT

```
intervalli che contengono mu: 95 su 100
```

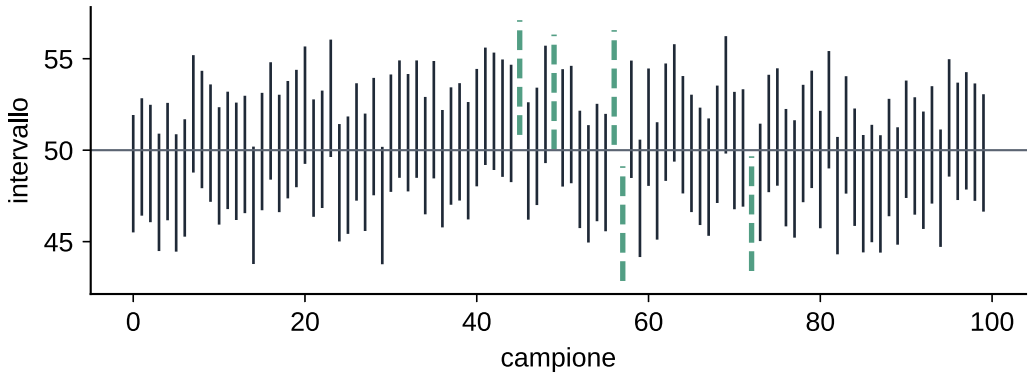

Figura 5.1. Cento intervalli di confidenza al 95% per la media μ = 50 (linea orizzontale). Quelli tratteggiati e più spessi non contengono μ.

Su 100 intervalli ci si aspetta che circa 5 manchino il valore vero; il numero osservato oscilla attorno a questo valore da una simulazione all'altra. Ripetendo con 10 000 campioni la frazione si avvicinerebbe molto a 0,95.

AMPIEZZA E NUMEROSITÀ

L'ampiezza dell'intervallo, $2z_{1-\alpha/2}\sigma/\sqrt{n}$, decresce come $1/\sqrt{n}$. Il frammento seguente lo mostra e calcola la numerosità necessaria per ottenere un margine fissato.

```
sigma = 1.8
for n in [9, 36, 144, 576]:
    amp = 2 * stats.norm.ppf(0.975) * sigma / np.sqrt(n)
    print(f"n = {n:4d}  ampiezza = {amp:.3f}")

E = 0.3                                      # margine desiderato
n_min = (stats.norm.ppf(0.975) * sigma / E) ** 2
print("numerosità minima:", int(np.ceil(n_min)))
```

OUTPUT

```
n =    9  ampiezza = 2.352
n =   36  ampiezza = 1.176
n =  144  ampiezza = 0.588
n =  576  ampiezza = 0.294
numerosità minima: 139
```

Ogni volta che n quadruplica l'ampiezza si dimezza. Per un margine di 0,3 ore servono almeno 139 batterie, come nell'esempio della teoria.

CAPITOLO 6

Test di ipotesi

Test z e t sulla media, test su una proporzione, confronto tra due medie, test sulla varianza, test chi quadro di adattamento e di indipendenza, con il calcolo manuale e con le funzioni di scipy.stats.

Lo schema in codice

Ogni test si traduce in codice con gli stessi cinque passi della teoria: si fissano le ipotesi e il livello α, si calcola la statistica test, si trova il valore critico con ppf oppure il p value con cdf o sf, si decide e si scrive la conclusione. Prima di scrivere una riga bisogna rispondere a due domande: la varianza della popolazione è nota (test z) o va stimata dai dati (test t)? L'alternativa è bilaterale, unilaterale destra o unilaterale sinistra?

ALTERNATIVA	RIFIUTO SE	P VALUE (STATISTICA Z)
bilaterale, $\theta \neq \theta_0$	|z| > norm.ppf(1 − α/2)	2 * stats.norm.sf(abs(z))
unilaterale destra, $\theta > \theta_0$	z > norm.ppf(1 − α)	stats.norm.sf(z)
unilaterale sinistra, $\theta < \theta_0$	z < norm.ppf(α)	stats.norm.cdf(z)

Per la statistica t si usano le stesse formule con stats.t e i gradi di libertà, per esempio 2 * stats.t.sf(abs(t), n − 1).

Test z sulla media

Esempio. I bulloni devono essere lunghi in media 50 mm, con σ = 0,4 mm nota. Su 25 bulloni la media è 50,18 mm. Verifichiamo H_0: μ = 50 contro H_1: μ ≠ 50 al 5%.

```
mu0, sigma, n, xbar, alpha = 50, 0.4, 25, 50.18, 0.05
z = (xbar - mu0) / (sigma / np.sqrt(n))          # statistica test
critico = stats.norm.ppf(1 - alpha / 2)          # valore critico bilaterale
p_value = 2 * stats.norm.sf(abs(z))              # p value a due code
print(f"z = {z:.3f}  critico = {critico:.3f}  p value = {p_value:.4f}")
if p_value < alpha:
    print("si rifiuta H0: la lunghezza media è diversa da 50 mm")
else:
    print("non si rifiuta H0")
```

OUTPUT

```
z = 2.250  critico = 1.960  p value = 0.0244
si rifiuta H0: la lunghezza media è diversa da 50 mm
```

Test t sulla media

Quando σ non è nota la si stima con np.std(dati, ddof=1) e si usa la t con n − 1 gradi di libertà. La funzione stats.ttest_1samp(dati, mu0, alternative=...) fa tutto il calcolo e restituisce statistica e p value; il parametro alternative vale "greater" per l'alternativa unilaterale destra e "less" per quella sinistra; se lo si omette il test è bilaterale.

Esempio. Il vecchio algoritmo impiegava in media 120 ms. Sedici esecuzioni del nuovo algoritmo danno i tempi riportati nel codice. Il nuovo algoritmo è più veloce? H_0: $\mu = 120$ contro H_1: $\mu < 120$.

```
tempi = np.array([118, 109, 125, 112, 104, 121, 115, 110,
                  127, 108, 113, 119, 102, 116, 111, 114])
mu0, alpha = 120, 0.05
n = len(tempi)
t = (tempi.mean() - mu0) / (tempi.std(ddof=1) / np.sqrt(n))   # statistica t
critico = stats.t.ppf(alpha, n - 1)                            # coda sinistra
p_value = stats.t.cdf(t, n - 1)
print(f"media = {tempi.mean():.2f}  s = {tempi.std(ddof=1):.2f}")
print(f"t = {t:.3f}  critico = {critico:.3f}  p value = {p_value:.4f}")

ris = stats.ttest_1samp(tempi, mu0, alternative="less")      # stesso test
print(f"scipy: t = {ris.statistic:.3f}  p value = {ris.pvalue:.4f}")
```

OUTPUT

```
media = 114.00  s = 6.93
t = -3.464  critico = -1.753  p value = 0.0017
scipy: t = -3.464  p value = 0.0017
```

Poiché il p value è minore di 0,05 (equivalentemente t è minore del valore critico), si rifiuta H_0: al livello del 5% c'è evidenza che il nuovo algoritmo sia più veloce. Il calcolo manuale e la funzione di scipy danno lo stesso risultato; conviene saper fare entrambi, e usare la funzione come controllo.

Test su una proporzione

Con n grande si usa la statistica z con p_0 nella varianza. In alternativa stats.binomtest(k, n, p0, alternative=...) esegue il test esatto basato sulla binomiale, che è preferibile quando n è piccolo.

Esempio. Un fornitore garantisce al massimo il 3% di pezzi difettosi; in un lotto di 500 se ne trovano 24. H_0: $p = 0{,}03$ contro H_1: $p > 0{,}03$.

```
k, n, p0, alpha = 24, 500, 0.03, 0.05
p = k / n
z = (p - p0) / np.sqrt(p0 * (1 - p0) / n)          # approssimazione normale
print(f"p stimata = {p:.3f}  z = {z:.3f}  p value = {stats.norm.sf(z):.4f}")

esatto = stats.binomtest(k, n, p0, alternative="greater")   # test binomiale esatto
print(f"p value esatto = {esatto.pvalue:.4f}")
```

OUTPUT

```
p stimata = 0.048  z = 2.359  p value = 0.0092
p value esatto = 0.0179
```

Entrambi i test portano a rifiutare H_0, ma i due p value differiscono: con p_0 piccolo il numero atteso di difettosi, $np_0 = 15$, è modesto e l'approssimazione normale è meno precisa. In questi casi il test esatto è più affidabile.

Confronto tra due medie

Con varianze note si calcola a mano la statistica z sulla differenza delle medie. Con varianze non note ma supposte uguali si usa stats.ttest_ind(x, y), che usa la varianza combinata e la t con n + m − 2 gradi di libertà; con equal_var=False esegue invece la variante di Welch, che non richiede varianze uguali.

Esempio. Tempi di elaborazione di due versioni di un programma su lotti di dati diversi. C'è differenza tra le medie?

```
v1 = np.array([31.2, 29.8, 33.5, 30.1, 32.4, 28.9, 31.7, 30.6, 32.9, 29.5])
v2 = np.array([28.4, 27.9, 30.2, 26.8, 29.1, 28.7, 27.5, 29.6, 28.0, 27.2])
ris = stats.ttest_ind(v1, v2)                  # varianze uguali (pooled)
print(f"medie: {v1.mean():.2f} e {v2.mean():.2f}")
print(f"t = {ris.statistic:.3f}  p value = {ris.pvalue:.6f}")

# lo stesso calcolo a mano con la varianza combinata
n, m = len(v1), len(v2)
sp2 = ((n - 1) * v1.var(ddof=1) + (m - 1) * v2.var(ddof=1)) / (n + m - 2)
t = (v1.mean() - v2.mean()) / np.sqrt(sp2 * (1 / n + 1 / m))
print(f"a mano: t = {t:.3f}  p value = {2 * stats.t.sf(abs(t), n + m - 2):.6f}")
```

OUTPUT

```
medie: 31.06 e 28.34
t = 4.574  p value = 0.000235
a mano: t = 4.574  p value = 0.000235
```

Test sulla varianza

Per una popolazione normale la statistica $(n-1)s^2/\sigma_0^2$ segue una chi quadro con n − 1 gradi di libertà. Non c'è una funzione dedicata in scipy.stats, ma il calcolo è di poche righe.

Esempio. Una macchina è regolare se la varianza del diametro non supera 0,25 mm^2; su 20 pezzi si osserva s^2 = 0,42. H_0: $\sigma^2 = 0{,}25$ contro H_1: $\sigma^2 > 0{,}25$.

```
n, s2, sigma0_2, alpha = 20, 0.42, 0.25, 0.05
chi = (n - 1) * s2 / sigma0_2                  # statistica test
critico = stats.chi2.ppf(1 - alpha, n - 1)     # coda destra
p_value = stats.chi2.sf(chi, n - 1)
print(f"chi2 = {chi:.2f}  critico = {critico:.2f}  p value = {p_value:.4f}")
```

OUTPUT

```
chi2 = 31.92  critico = 30.14  p value = 0.0319
```

Test chi quadro di adattamento

La funzione stats.chisquare(osservate, attese) calcola la statistica $\Sigma\,(O_i - E_i)^2/E_i$ e il p value con k − 1 gradi di libertà. Le frequenze attese devono avere la stessa somma delle osservate. Se si sono stimati r parametri dai dati si aggiunge ddof=r, che toglie altrettanti gradi di libertà.

Esempio. Un dado lanciato 120 volte dà le frequenze 14, 22, 18, 25, 16, 25. È equilibrato?

```
oss = np.array([14, 22, 18, 25, 16, 25])          # frequenze osservate
att = np.full(6, oss.sum() / 6)                   # attese: 20 per faccia
chi = ((oss - att) ** 2 / att).sum()              # statistica a mano
print(f"a mano: chi2 = {chi:.2f}  critico = {stats.chi2.ppf(0.95, 5):.2f}")
ris = stats.chisquare(oss, att)
print(f"scipy : chi2 = {ris.statistic:.2f}  p value = {ris.pvalue:.4f}")
```

OUTPUT

```
a mano: chi2 = 5.50  critico = 11.07
scipy : chi2 = 5.50  p value = 0.3579
```

Il p value di circa 0,36 è molto maggiore di 0,05: non si rifiuta l'ipotesi che il dado sia equilibrato, perché le differenze osservate sono compatibili con il caso.

ADATTAMENTO A UNA DISTRIBUZIONE CON PARAMETRO STIMATO

Esempio. Si contano gli accessi a un sito in 200 minuti e si vuole verificare se seguono una Poisson. Il parametro λ si stima con la media dei dati, quindi si perde un grado di libertà in più. Le categorie da 4 in su si accorpano in un'unica categoria "4 o più", perché le frequenze attese siano almeno 5.

```
valori = np.arange(0, 7)                       # accessi in un minuto: da 0 a 6
conteggi = np.array([28, 61, 54, 34, 15, 6, 2])  # minuti con quel numero di accessi
n = conteggi.sum()
lam = (valori * conteggi).sum() / n            # stima di lambda: la media dei dati

oss = np.append(conteggi[:4], conteggi[4:].sum())   # accorpa "4 o più"
prob = stats.poisson.pmf(np.arange(0, 4), lam)
prob = np.append(prob, stats.poisson.sf(3, lam))    # P(X >= 4)
att = n * prob
print("lambda stimato:", round(lam, 3))
print("osservate:", oss, " attese:", np.round(att, 1))
ris = stats.chisquare(oss, att, ddof=1)       # un parametro stimato
print(f"chi2 = {ris.statistic:.2f}  p value = {ris.pvalue:.4f}")
```

OUTPUT

```
lambda stimato: 1.865
osservate: [28 61 54 34 23]  attese: [31.  57.8 53.9 33.5 23.9]
chi2 = 0.51  p value = 0.9175
```

Il parametro λ si stima con tutti i dati, prima di accorpare le categorie. Con 5 categorie e un parametro stimato i gradi di libertà sono 5 − 1 − 1 = 3. Il p value elevato indica che il modello di Poisson è compatibile con i dati.

Test chi quadro di indipendenza

Per una tabella di contingenza si usa stats.chi2_contingency(tabella, correction=False), che restituisce statistica, p value, gradi di libertà e tabella delle frequenze attese. L'opzione correction=False disattiva la correzione di Yates, che scipy applica per default alle sole tabelle 2 per 2.

Esempio. Sistema operativo usato da 160 studenti di due corsi (i dati della dispensa di teoria).

```
tabella = np.array([[30, 20, 30],             # informatica: Windows, macOS, Linux
                    [50, 25, 5]])             # economia
ris = stats.chi2_contingency(tabella, correction=False)
print(f"chi2 = {ris.statistic:.2f}  gradi di libertà = {ris.dof}")
print(f"p value = {ris.pvalue:.6f}")
print("frequenze attese:")
print(ris.expected_freq)
```

OUTPUT

```
chi2 = 23.41  gradi di libertà = 2
p value = 0.000008
frequenze attese:
[[40.  22.5 17.5]
 [40.  22.5 17.5]]
```

Il livello α in una simulazione

Il livello di significatività è la probabilità di rifiutare H_0 quando è vera. Lo si può verificare generando molti campioni da una popolazione in cui H_0 è vera e contando quante volte il test la rifiuta: la frazione deve essere vicina ad α.

```
np.random.seed(30)
R, n, alpha = 10000, 15, 0.05
rifiuti = 0
for i in range(R):
    x = np.random.normal(100, 10, size=n)       # H0 vera: media 100
    if stats.ttest_1samp(x, 100).pvalue < alpha:
        rifiuti += 1
print("frazione di rifiuti con H0 vera:", rifiuti / R)
```

OUTPUT

```
frazione di rifiuti con H0 vera: 0.0509
```

La frazione è vicina a 0,05, come deve essere: in un caso su venti il test rifiuta un'ipotesi vera. È il prezzo dell'errore di primo tipo, fissato in anticipo scegliendo α.

CAPITOLO 7

Regressione lineare

Covarianza, correlazione e retta dei minimi quadrati calcolate con le formule e con linregress, grafico, previsioni, residui, modelli linearizzabili e regressione multipla con le matrici.

Le formule in codice

Le quantità della regressione semplice si calcolano direttamente con le formule della teoria, che usano le varianze empiriche (divisore n). Conviene scriverle in una funzione, così da poterla riusare su dati diversi senza rischiare errori di trascrizione.

```
def regressione(x, y):
    """Restituisce beta0, beta1, rho e R con le formule dei minimi quadrati."""
    x, y = np.array(x, float), np.array(y, float)
    var_x = np.mean(x ** 2) - x.mean() ** 2            # varianza empirica di x
    var_y = np.mean(y ** 2) - y.mean() ** 2            # varianza empirica di y
    cov_xy = np.mean(x * y) - x.mean() * y.mean()      # covarianza empirica
    beta1 = cov_xy / var_x                             # pendenza
    beta0 = y.mean() - beta1 * x.mean()                # intercetta
    rho = cov_xy / np.sqrt(var_x * var_y)              # correlazione
    return beta0, beta1, rho, rho ** 2                 # R = rho al quadrato

ore = np.array([2, 3, 5, 6, 8, 9, 11, 12])             # ore di studio
punti = np.array([52, 55, 63, 61, 72, 75, 82, 84])     # punteggio
b0, b1, rho, R = regressione(ore, punti)
print(f"retta: y = {b0:.3f} + {b1:.4f} x")
print(f"rho = {rho:.4f}   R = {R:.4f}")
```

OUTPUT

```
retta: y = 44.946 + 3.2935 x
rho = 0.9911   R = 0.9822
```

I risultati coincidono con quelli calcolati a mano nella dispensa di teoria. La funzione stats.linregress(x, y) esegue lo stesso calcolo e restituisce un oggetto con gli attributi slope (β_1), intercept (β_0), rvalue (ρ), pvalue (il p value del test che la pendenza sia nulla) e stderr (l'errore standard della pendenza). Il coefficiente di determinazione non è restituito direttamente: si ottiene come rvalue al quadrato.

```
lr = stats.linregress(ore, punti)
print(f"linregress: beta1 = {lr.slope:.4f}  beta0 = {lr.intercept:.3f}")
print(f"rho = {lr.rvalue:.4f}  R = {lr.rvalue ** 2:.4f}  p value = {lr.pvalue:.2e}")
```

OUTPUT

```
linregress: beta1 = 3.2935  beta0 = 44.946
rho = 0.9911  R = 0.9822  p value = 1.77e-06
```

Grafico e previsioni

Il grafico mostra i dati come nuvola di punti e la retta disegnata solo nell'intervallo dei dati osservati, dove il modello è stato stimato. Una previsione si ottiene sostituendo x nell'equazione della retta.

```
xs = np.linspace(ore.min(), ore.max(), 100)        # solo dove ci sono dati
plt.scatter(ore, punti, label="dati")
plt.plot(xs, b0 + b1 * xs, label="retta di regressione")
plt.xlabel("ore di studio"); plt.ylabel("punteggio"); plt.legend()
plt.show()

for x_nuovo in [4, 10]:
    print(f"previsione per {x_nuovo} ore: {b0 + b1 * x_nuovo:.1f} punti")
```

OUTPUT

```
previsione per 4 ore: 58.1 punti
previsione per 10 ore: 77.9 punti
```

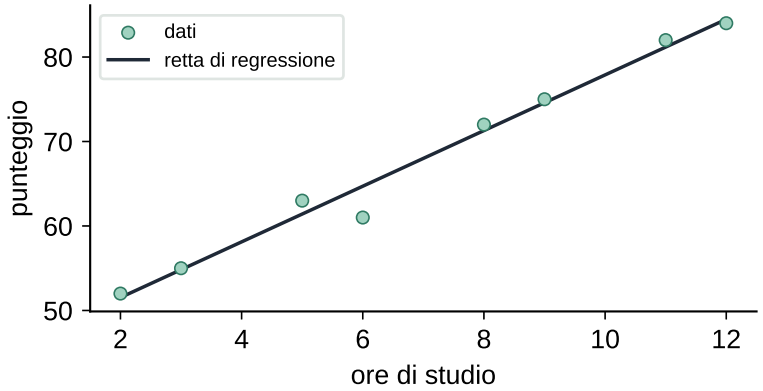

Figura 7.1. Dati e retta dei minimi quadrati per l'esempio delle ore di studio.

Residui e scomposizione della devianza

I residui sono le differenze tra i valori osservati e quelli previsti dalla retta. La loro somma è sempre zero, e la devianza totale si scompone in devianza spiegata più devianza residua: il codice lo verifica e ricalcola R come frazione di devianza spiegata.

```
previsti = b0 + b1 * ore
residui = punti - previsti
dev_tot = np.sum((punti - punti.mean()) ** 2)
dev_spieg = np.sum((previsti - punti.mean()) ** 2)
dev_res = np.sum(residui ** 2)
print("somma dei residui:", round(abs(residui.sum()), 10))
print(f"totale {dev_tot:.2f} = spiegata {dev_spieg:.2f} + residua {dev_res:.2f}")
print(f"R = 1 - residua/totale = {1 - dev_res / dev_tot:.4f}")
```

OUTPUT

```
somma dei residui: 0.0
totale 1016.00 = spiegata 997.92 + residua 18.08
R = 1 - residua/totale = 0.9822
```

Un grafico dei residui in funzione di x è un buon controllo del modello: se i residui sono sparsi a caso attorno a zero il modello lineare è adeguato; se mostrano un andamento sistematico, per esempio a forma di U, la relazione non è lineare.

Modelli linearizzabili

Se i dati crescono in modo esponenziale, $y = a \cdot e^{bx}$, passando ai logaritmi si ottiene $\ln y = \ln a + bx$, che è una retta tra x e ln y. Si applica la regressione a (x, ln y) e poi si torna indietro con $a = e^{\beta_0}$.

Esempio. Il numero di utenti registrati a un servizio nei primi otto mesi.

```
mesi = np.array([1, 2, 3, 4, 5, 6, 7, 8])
utenti = np.array([120, 165, 230, 310, 440, 590, 830, 1120])
c0, c1, rho_log, R_log = regressione(mesi, np.log(utenti))   # retta su ln(y)
a, b = np.exp(c0), c1
print(f"modello: y = {a:.1f} * exp({b:.4f} x)   R su ln(y) = {R_log:.4f}")
print(f"crescita mensile stimata: {100 * (np.exp(b) - 1):.1f}%")
print(f"previsione per il mese 9: {a * np.exp(b * 9):.0f} utenti")
```

OUTPUT

```
modello: y = 87.3 * exp(0.3201 x)   R su ln(y) = 0.9998
crescita mensile stimata: 37.7%
previsione per il mese 9: 1556 utenti
```

Il coefficiente b è il tasso di crescita continuo: ogni mese il numero di utenti viene moltiplicato per e^{b}. Anche qui la previsione al mese 9 è un'estrapolazione appena fuori dall'intervallo dei dati, da prendere con cautela.

Regressione multipla con le matrici

Con più variabili esplicative si costruisce la matrice del disegno X, con una prima colonna di 1 per l'intercetta e una colonna per ogni variabile, e si risolvono le equazioni normali $X^{T}X\beta = X^{T}y$. In NumPy il prodotto tra matrici si scrive con l'operatore @, la trasposta con .T, e il sistema lineare si risolve con np.linalg.solve. La funzione np.linalg.lstsq(X, y) risolve direttamente il problema dei minimi quadrati ed è numericamente più stabile.

Esempio. Il tempo di esecuzione in funzione della dimensione dell'input e del numero di processori (i dati della dispensa di teoria).

```
x1 = np.array([1, 2, 3, 4, 5, 6])                  # migliaia di righe
x2 = np.array([1, 2, 1, 2, 1, 2])                  # processori
y = np.array([2.1, 2.9, 6.2, 6.8, 10.1, 10.9])     # secondi

X = np.column_stack([np.ones(len(y)), x1, x2])     # matrice del disegno
beta = np.linalg.solve(X.T @ X, X.T @ y)           # equazioni normali
print("X^T X =")
print(X.T @ X)
print("beta =", np.round(beta, 4))
print("con lstsq:", np.round(np.linalg.lstsq(X, y, rcond=None)[0], 4))

previsti = X @ beta
R = 1 - np.sum((y - previsti) ** 2) / np.sum((y - y.mean()) ** 2)
n, p = len(y), 2                                   # osservazioni e variabili
R_corr = 1 - (1 - R) * (n - 1) / (n - p - 1)       # coefficiente corretto
print(f"R = {R:.5f}   R corretto = {R_corr:.5f}")
```

OUTPUT

```
X^T X =
[[ 6. 21.  9.]
 [21. 91. 33.]
 [ 9. 33. 15.]]
beta = [ 1.4     2.     -1.2667]
con lstsq: [ 1.4     2.     -1.2667]
R = 0.99979   R corretto = 0.99966
```

Il modello stimato è $y \approx 1{,}4 + 2x_1 - 1{,}267x_2$: a parità di processori ogni mille righe costano 2 secondi, a parità di input ogni processore in più fa risparmiare circa 1,27 secondi. La funzione np.column_stack affianca i vettori come colonne di una matrice; np.ones(n) produce la colonna di 1.

CAPITOLO 8

Numeri pseudocasuali e metodo Monte Carlo

Un generatore congruenziale scritto a mano, la generazione di variabili con densità assegnata per inversione e per rigetto, il calcolo di integrali con i metodi hit or miss e della media campionaria, e l'errore delle stime.

Un generatore congruenziale

Il generatore congruenziale lineare $X_{n+1} = (aX_n + c) \bmod m$ si scrive in poche righe; in Python il resto della divisione si calcola con l'operatore %. La funzione seguente restituisce i primi k numeri $U_n = X_n/m$.

```
def lcg(a, c, m, seme, k):
    """Primi k numeri del generatore congruenziale, divisi per m."""
    x = seme
    numeri = []
    for i in range(k):
        x = (a * x + c) % m          # passo del generatore
        numeri.append(x / m)         # numero in [0, 1)
    return np.array(numeri)

u = lcg(9, 7, 16, 1, 17)
print("valori X:", (u * 16).astype(int))     # periodo pieno: compaiono tutti
u2 = lcg(9, 4, 16, 1, 8)
print("con c = 4:", (u2 * 16).astype(int))   # c non primo con 16: periodo 4
```

OUTPUT

```
valori X: [ 0  7  6 13 12  3  2  9  8 15 14  5  4 11 10  1  0]
con c = 4: [13  9  5  1 13  9  5  1]
```

Con m = 16 i valori possibili sono solo 16 e la sequenza si ripete subito; per un uso reale servono moduli enormi. Un generatore storico è quello "minimo standard", moltiplicativo con $m = 2^{31} - 1$ e a = 16807. I numeri che produce si distribuiscono in modo uniforme, come si verifica con un test chi quadro sulle frequenze di dieci classi uguali.

```
u = lcg(16807, 0, 2**31 - 1, 12345, 100000)
freq, _ = np.histogram(u, bins=10, range=(0, 1))   # dieci classi di ampiezza 0.1
print("frequenze:", freq)
print("media:", round(u.mean(), 4), " (attesa 0.5)   varianza:", round(u.var(), 5),
      " (attesa 1/12 =", round(1 / 12, 5), ")")
print("p value del test chi quadro:", round(stats.chisquare(freq).pvalue, 4))
```

OUTPUT

```
frequenze: [ 9974 10042  9987 10079 10001  9939 10003 10024  9988  9963]
media: 0.5  (attesa 0.5)   varianza: 0.08337  (attesa 1/12 = 0.08333 )
p value del test chi quadro: 0.9974
```

Senza secondo argomento stats.chisquare usa come frequenze attese quelle uniformi, qui 10 000 per classe. Il p value alto indica che le frequenze sono compatibili con l'uniformità.

Il metodo dell'inversione

Se U è uniforme su (0, 1) allora $F^{-1}(U)$ ha funzione di ripartizione F. Per l'esponenziale l'inversa è esplicita: $X = -\ln(U)/\lambda$. Generiamo così 10 000 tempi esponenziali con $\lambda = 2$ e confrontiamoli con la teoria.

```
np.random.seed(40)
lam = 2
u = np.random.uniform(0, 1, size=10000)
x = -np.log(u) / lam                          # inversione della ripartizione
print("media simulata:", round(x.mean(), 4), "  teorica:", 1 / lam)
print("P(X > 1) simulata:", round((x > 1).mean(), 4),
      "  teorica:", round(np.exp(-lam), 4))
```

OUTPUT

```
media simulata: 0.4929   teorica: 0.5
P(X > 1) simulata: 0.1361   teorica: 0.1353
```

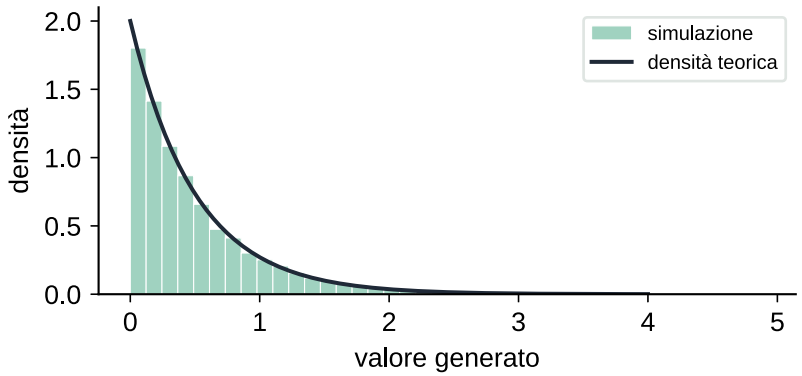

Figura 8.1. Valori generati per inversione e densità esponenziale con $\lambda = 2$.

VARIABILI DISCRETE

Per una variabile discreta si confronta U con le probabilità cumulate. La funzione np.searchsorted(cumulate, u) restituisce, per ogni u, la posizione del primo valore cumulato che lo supera, cioè l'indice dell'esito estratto.

Esempio. Lo stato di un server: attivo con probabilità 0,7, carico con 0,2, in manutenzione con 0,1.

```
np.random.seed(41)
stati = np.array(["attivo", "carico", "manutenzione"])
prob = np.array([0.7, 0.2, 0.1])
cumulate = np.cumsum(prob)                       # 0.7, 0.9, 1.0
u = np.random.uniform(0, 1, size=10000)
indici = np.searchsorted(cumulate, u, side="right")   # primo valore che supera u
estratti = stati[indici]
for st, p in zip(stati, prob):
    print(f"{st:13s} simulata {(estratti == st).mean():.4f}   teorica {p}")
```

OUTPUT

```
attivo        simulata 0.7033   teorica 0.7
carico        simulata 0.2013   teorica 0.2
manutenzione  simulata 0.0954   teorica 0.1
```

Lo stesso risultato si ottiene più semplicemente con np.random.choice(stati, size=10000, p=prob), che usa internamente questo metodo; scriverlo a mano serve a capire come funziona.

Il metodo del rigetto

Esempio. Vogliamo generare valori con densità f(x) = 6x(1 − x) su [0, 1], la cui ripartizione non si inverte facilmente. Il massimo di f è 1,5, in x = 1/2; scegliendo come densità di comodo l'uniforme su [0, 1] e M = 1,5 si ha f(x) ≤ M per ogni x. Si genera una proposta Y uniforme e la si accetta se un secondo numero uniforme U soddisfa U ≤ f(Y)/M.

```
np.random.seed(42)
f = lambda x: 6 * x * (1 - x)               # densità da cui generare
M = 1.5                                     # massimo di f su [0, 1]
N = 20000
y = np.random.uniform(0, 1, size=N)         # proposte
u = np.random.uniform(0, 1, size=N)
accettati = y[u <= f(y) / M]                # tengo solo le proposte accettate
print("frazione accettata:", round(len(accettati) / N, 4),
      "  teorica 1/M:", round(1 / M, 4))
print("media:", round(accettati.mean(), 4), "  teorica: 0.5")
print("varianza:", round(accettati.var(), 4), "  teorica: 0.05")
```

OUTPUT

```
frazione accettata: 0.6625   teorica 1/M: 0.6667
media: 0.4994   teorica: 0.5
varianza: 0.049   teorica: 0.05
```

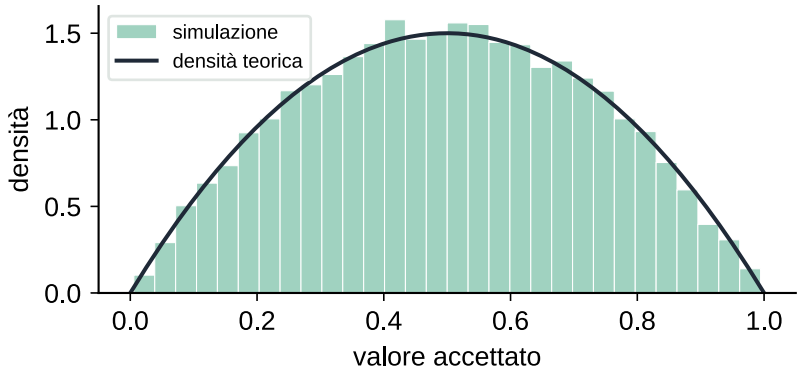

Figura 8.2. Valori generati con il metodo del rigetto e densità f(x) = 6x(1 − x).

ATTENZIONE

M deve essere almeno il massimo del rapporto f/g, altrimenti il metodo è sbagliato e non solo inefficiente. Conviene verificarlo numericamente, per esempio con xs = np.linspace(0, 1, 1000) e controllando che f(xs).max() / M non superi 1. Scegliere M più grande del necessario non introduce errori ma riduce la frazione di proposte accettate, che è 1/M.

Integrali con il metodo hit or miss

Per stimare I = ∫ f(x) dx su [a, b], con 0 ≤ f(x) ≤ c, si generano N punti uniformi nel rettangolo [a, b] × [0, c] e si conta la frazione di punti sotto la curva. L'errore standard della stima è c(b − a)√(p̂(1 − p̂)/N).

Esempio. $I = \int x^2\,dx$ su [0, 1], valore esatto 1/3.

```
np.random.seed(43)
f = lambda x: x ** 2
a, b, c = 0, 1, 1                          # rettangolo [a, b] x [0, c]
N = 10000
x = np.random.uniform(a, b, size=N)
y = np.random.uniform(0, c, size=N)
colpiti = y <= f(x)                        # punti sotto la curva
p = colpiti.mean()
area = c * (b - a)                         # area del rettangolo
stima = area * p
es = area * np.sqrt(p * (1 - p) / N)       # errore standard
print(f"stima = {stima:.4f}   errore standard = {es:.4f}   esatto = {1/3:.4f}")

stime = area * np.cumsum(colpiti) / np.arange(1, N + 1)   # stima dopo ogni punto
```

OUTPUT

```
stima = 0.3398   errore standard = 0.0047   esatto = 0.3333
```

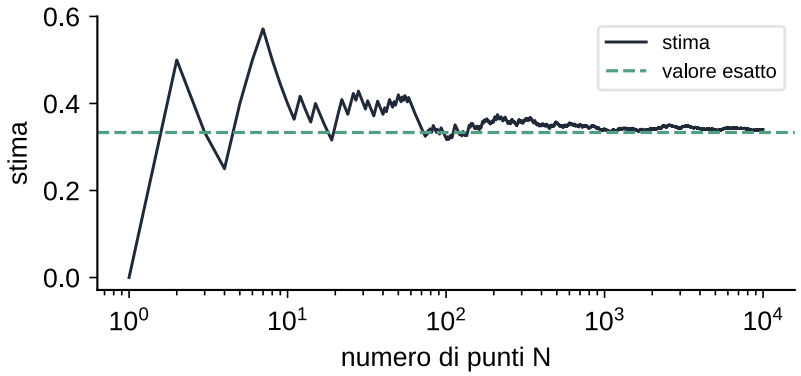

Figura 8.3. La stima hit or miss in funzione del numero di punti usati: oscilla molto all'inizio e si stabilizza attorno a 1/3.

L'ultima riga calcola la stima dopo 1, 2, …, N punti con np.cumsum, che conta i colpi cumulati: il grafico mostra come la stima si stabilizza al crescere di N, in accordo con la legge dei grandi numeri.

ESEMPIO: STIMARE Π

Il quarto di cerchio di raggio 1 ha area π/4 ed è contenuto nel quadrato unitario: un punto uniforme (x, y) cade nel quarto di cerchio se $x^2 + y^2 \le 1$. Quattro volte la frazione di punti dentro è una stima di π.

```
np.random.seed(44)
N = 1000000
x = np.random.uniform(0, 1, size=N)
y = np.random.uniform(0, 1, size=N)
p = (x ** 2 + y ** 2 <= 1).mean()          # frazione nel quarto di cerchio
es = 4 * np.sqrt(p * (1 - p) / N)          # errore standard della stima
print(f"pi stimato = {4 * p:.4f}   errore standard = {es:.4f}")
```

OUTPUT

```
pi stimato = 3.1406   errore standard = 0.0016
```

Il metodo della media campionaria

Se X è uniforme su [a, b], la quantità $(b - a) \cdot f(X)$ ha valore atteso uguale all'integrale. La stima è quindi $(b - a)$ per la media di f su N punti uniformi, con errore standard $(b - a)s_f/\sqrt{N}$, dove s_f è la deviazione standard dei valori $f(X_i)$. Il metodo non richiede di conoscere un limite superiore di f e di solito è più preciso dell'hit or miss.

```
np.random.seed(45)
N = 10000
# stesso integrale di prima: x^2 su [0, 1]
x = np.random.uniform(0, 1, size=N)
fx = x ** 2
es = fx.std(ddof=1) / np.sqrt(N)
print(f"x^2:   stima = {fx.mean():.4f}   errore standard = {es:.4f}")

# integrale di sin(x) su [0, pi], valore esatto 2
a, b = 0, np.pi
x = np.random.uniform(a, b, size=N)
fx = np.sin(x)
stima = (b - a) * fx.mean()
es = (b - a) * fx.std(ddof=1) / np.sqrt(N)
print(f"sin x: stima = {stima:.4f}   errore standard = {es:.4f}   esatto = 2")
```

OUTPUT

```
x^2:   stima = 0.3315   errore standard = 0.0030
sin x: stima = 2.0018   errore standard = 0.0096   esatto = 2
```

A parità di punti l'errore standard della media campionaria per x^2 (circa 0,003) è più piccolo di quello dell'hit or miss (circa 0,0047), come previsto dalla teoria. In entrambi i metodi l'errore decresce come $1/\sqrt{N}$.

DA RICORDARE

Ogni stima Monte Carlo va accompagnata dal suo errore standard. Una stima che dista dal valore esatto meno di due o tre errori standard è compatibile con la teoria; per ridurre l'errore di un fattore 10 servono 100 volte più punti.

CAPITOLO 9

Catene di Markov

Matrice di transizione, potenze di matrice e distribuzione al tempo n, distribuzione stazionaria con tre metodi diversi, simulazione di una traiettoria e classificazione automatica degli stati.

La matrice di transizione

Una catena con k stati si descrive con un array k per k: la riga i contiene le probabilità di passare dallo stato i a ciascuno degli stati. Prima di tutto si verifica che la matrice sia stocastica, cioè che gli elementi siano non negativi e che **ogni riga** sommi a 1; P.sum(axis=1) somma lungo le righe e np.allclose confronta tenendo conto degli arrotondamenti.

Esempio. La catena del server della dispensa di teoria, con stati attivo, carico e manutenzione.

```
stati = ["attivo", "carico", "manutenzione"]
P = np.array([[0.7, 0.2, 0.1],
              [0.5, 0.3, 0.2],
              [0.6, 0.0, 0.4]])
print("righe:", P.sum(axis=1))
print("stocastica:", np.allclose(P.sum(axis=1), 1) and (P >= 0).all())
```

OUTPUT

```
righe: [1. 1. 1.]
stocastica: True
```

ATTENZIONE

La riga è lo stato di partenza e la colonna quello di arrivo. Leggendo un grafo, ogni anello su uno stato diventa l'elemento sulla diagonale: dimenticarlo è l'errore che fa sì che una riga non sommi a 1.

Transizioni in n passi e distribuzione al tempo n

La matrice di transizione in n passi è la potenza P^n, che si calcola con np.linalg.matrix_power(P, n). Attenzione: P ** n eleverebbe a potenza ogni elemento separatamente, che non è il prodotto tra matrici. La distribuzione al tempo n si ottiene dal vettore riga della distribuzione iniziale con $\pi^{(n)} = \pi^{(0)}P^n$, oppure aggiornando passo per passo con pi = pi @ P.

```
np.set_printoptions(precision=4, suppress=True)   # stampa con 4 decimali
print("P^2 =")
print(np.linalg.matrix_power(P, 2))

pi = np.array([0.0, 0.0, 1.0])            # parte in manutenzione
storia = [pi]
for n in range(1, 11):
    pi = pi @ P                           # un passo della catena
    storia.append(pi)
storia = np.array(storia)
print("pi(1) =", storia[1])
print("pi(2) =", storia[2])
print("pi(10) =", storia[10])
```

OUTPUT

```
P^2 =
[[0.65 0.2  0.15]
 [0.62 0.19 0.19]
 [0.66 0.12 0.22]]
pi(1) = [0.6 0.  0.4]
pi(2) = [0.66 0.12 0.22]
pi(10) = [0.6462 0.1846 0.1692]
```

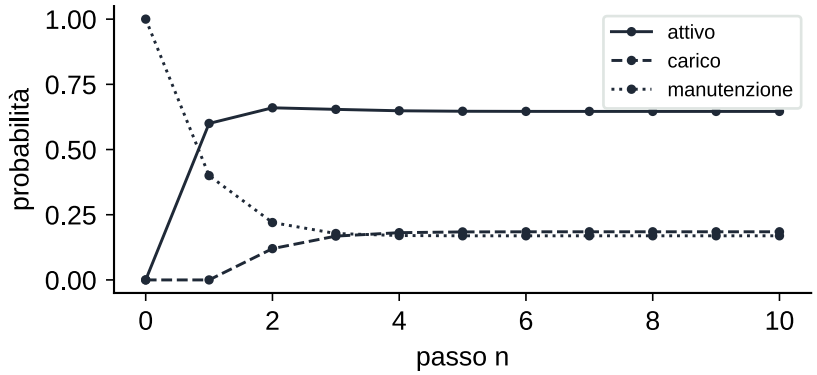

Figura 9.1. Evoluzione della distribuzione del server partendo dallo stato di manutenzione: dopo pochi passi le probabilità si stabilizzano.

La distribuzione cambia rapidamente nei primi passi e poi si stabilizza: dopo una decina di passi non dipende più, in pratica, dallo stato di partenza. Ripetendo il calcolo da $\pi^{(0)}$ = (1, 0, 0) si arriverebbe agli stessi valori.

La distribuzione stazionaria

La distribuzione stazionaria π soddisfa π = πP con la somma delle componenti uguale a 1. Al calcolatore si trova in tre modi.

Primo metodo: potenza. Per una catena ergodica tutte le righe di P^n tendono a π, quindi basta elevare P a una potenza alta e leggere una riga.

```
P100 = np.linalg.matrix_power(P, 100)
print(P100)                               # tutte le righe uguali
```

OUTPUT

```
[[0.6462 0.1846 0.1692]
 [0.6462 0.1846 0.1692]
 [0.6462 0.1846 0.1692]]
```

Secondo metodo: sistema lineare. Le equazioni $\pi(P - I) = 0$, trasposte, diventano $(P^T - I)\pi^T = 0$. Sono dipendenti, quindi si sostituisce l'ultima con la condizione di normalizzazione, una riga di 1, e si risolve il sistema con np.linalg.solve.

```
k = len(P)
A = P.T - np.eye(k)                       # equazioni (P^T - I) pi = 0
A[-1, :] = 1                              # ultima equazione: somma = 1
b = np.zeros(k)
b[-1] = 1
pi_staz = np.linalg.solve(A, b)
print("pi stazionaria:", pi_staz)
print("in frazioni   :", np.round(pi_staz * 65, 4), "/ 65")
print("verifica pi P :", pi_staz @ P)
```

OUTPUT

```
pi stazionaria: [0.6462 0.1846 0.1692]
in frazioni   : [42. 12. 11.] / 65
verifica pi P : [0.6462 0.1846 0.1692]
```

Terzo metodo: autovettori. π è un autovettore sinistro di P relativo all'autovalore 1, cioè un autovettore di P^T. La funzione np.linalg.eig restituisce autovalori e autovettori; si sceglie l'autovettore dell'autovalore più vicino a 1 e lo si normalizza perché sommi a 1.

```
autovalori, autovettori = np.linalg.eig(P.T)
j = np.argmin(np.abs(autovalori - 1))     # posizione dell'autovalore 1
v = np.real(autovettori[:, j])
print("pi dagli autovettori:", v / v.sum())
print("tempi medi di ritorno:", 1 / pi_staz)
```

OUTPUT

```
pi dagli autovettori: [0.6462 0.1846 0.1692]
tempi medi di ritorno: [1.5476 5.4167 5.9091]
```

I tre metodi danno lo stesso vettore (42/65, 12/65, 11/65) trovato a mano nella teoria. Gli inversi delle componenti sono i tempi medi di ritorno: partendo dallo stato di manutenzione, il primo ritorno in quello stato avviene in media dopo circa 5,9 passi (contando anche il caso in cui il server vi resta al passo successivo).

Simulare una traiettoria

Per simulare la catena si parte da uno stato e a ogni passo si estrae lo stato successivo dalla riga corrente di P, con np.random.choice(k, p=P[stato]). Contando la frazione di passi trascorsi in ciascuno stato si ottiene una stima della distribuzione stazionaria.

```
np.random.seed(50)
passi = 100000
stato = 2                                  # parte da manutenzione
visite = np.zeros(k)
for t in range(passi):
    stato = np.random.choice(k, p=P[stato])   # estrae lo stato successivo
    visite[stato] += 1
print("frazione di tempo simulata:", visite / passi)
print("distribuzione stazionaria :", pi_staz)
```

OUTPUT

```
frazione di tempo simulata: [0.6458 0.1884 0.1659]
distribuzione stazionaria : [0.6462 0.1846 0.1692]
```

Le frazioni simulate sono vicine alla distribuzione stazionaria, come previsto dalla legge dei grandi numeri per le catene ergodiche. La simulazione è più lenta del calcolo esatto, ma è l'unico strumento disponibile quando la catena è troppo complicata per essere risolta.

Classificare gli stati

Lo stato j è accessibile da i se esiste un cammino da i a j. Con k stati basta controllare cammini lunghi al più $k - 1$ passi, quindi j è accessibile da i se e solo se l'elemento (i, j) di $(I + P)^{k-1}$ è positivo. Da qui si ricavano le classi di comunicazione (stati accessibili l'uno dall'altro) e si verifica quali sono chiuse.

Esempio. La catena a quattro stati della dispensa di teoria, con lo stato 4 assorbente.

```
Q = np.array([[0.0, 1.0, 0.0, 0.0],
              [0.5, 0.0, 0.5, 0.0],
              [0.0, 0.6, 0.0, 0.4],
              [0.0, 0.0, 0.0, 1.0]])
k = len(Q)
R = np.linalg.matrix_power(np.eye(k) + Q, k - 1) > 0   # R[i, j]: j accessibile da i
C = R & R.T                                              # i e j comunicano
classi = []
for i in range(k):
    classe = set(int(j) + 1 for j in np.where(C[i])[0])  # stati numerati da 1
    if classe not in classi:
        classi.append(classe)
for cl in classi:
    idx = [s - 1 for s in cl]
    chiusa = np.allclose(Q[idx][:, idx].sum(axis=1), 1)  # non si esce dalla classe
    print("classe", cl, "chiusa" if chiusa else "non chiusa")
print("irriducibile:", len(classi) == 1)
print("Q^50, riga dello stato 1:", np.linalg.matrix_power(Q, 50)[0])
```

OUTPUT

```
classe {1, 2, 3} non chiusa
classe {4} chiusa
irriducibile: False
Q^50, riga dello stato 1: [0.0024 0.     0.0024 0.9953]
```

Il programma trova le due classi {1, 2, 3}, non chiusa, e {4}, chiusa: la catena non è irriducibile. L'ultima riga mostra che, partendo dallo stato 1, dopo 50 passi la catena è quasi certamente finita nello stato assorbente 4, confermando che gli stati 1, 2 e 3 sono transitori.

CAPITOLO 10

Guida rapida

Le funzioni usate nella dispensa raccolte per tipo di problema, lo schema da seguire in ogni esercizio e gli errori più comuni con il loro rimedio.

Quale funzione usare

PROBLEMA	FUNZIONI
Stimare una probabilità	np.random.*(size=N), condizione tra parentesi, .mean() dell'array logico
Probabilità esatte	stats.dist.pmf, pdf, cdf, sf, ppf; math.comb per i coefficienti binomiali
Confronto discreto	frequenze (x == v).mean(), plt.bar sfalsato di 0,2
Confronto continuo	plt.hist(..., density=True) e plt.plot della pdf
Momenti	np.mean, np.var, stats.skew, stats.kurtosis, dist.stats(moments="mvsk")
QQ plot	stats.probplot(x, dist=..., sparams=..., plot=plt)
Tabelle di frequenze	np.unique(..., return_counts=True), np.histogram, np.cumsum
Indici descrittivi	np.median, np.percentile, np.var(ddof=1), np.std(ddof=1)
Intervalli di confidenza	stats.norm.ppf, stats.t.ppf, stats.chi2.ppf; stats.norm.interval, stats.t.interval
Test sulla media	stats.ttest_1samp, stats.ttest_ind; a mano con norm.sf e t.sf
Test su una proporzione	statistica z a mano; stats.binomtest per il test esatto
Test chi quadro	stats.chisquare(oss, att, ddof=r), stats.chi2_contingency(tab, correction=False)
Regressione semplice	formule con np.mean; stats.linregress (R = rvalue al quadrato)
Regressione multipla	np.column_stack, X.T @ X, np.linalg.solve, np.linalg.lstsq
Monte Carlo	np.random.uniform, frazione di colpi, np.cumsum per la convergenza
Catene di Markov	np.linalg.matrix_power, pi @ P, np.linalg.solve, np.linalg.eig, np.random.choice(k, p=P[i])

Lo schema di ogni esercizio

Un esercizio di laboratorio ben svolto ha sempre la stessa struttura, e conviene seguirla anche quando la traccia non la chiede esplicitamente.

1. **Modello.** Scrivere quale distribuzione descrive il fenomeno e con quali parametri, con media e varianza attese.
2. **Calcolo teorico.** Calcolare la quantità richiesta con la formula o con scipy.stats.
3. **Simulazione.** Ripetere l'esperimento N volte con np.random, fissando il seme se si vuole riproducibilità.
4. **Confronto.** Mettere a confronto teoria e simulazione con i numeri, l'errore standard e un grafico.

5. **Commento.** Spiegare la differenza: è compatibile con la fluttuazione casuale se è dell'ordine dell'errore standard, e si riduce come 1/√N.
6. **Codice commentato.** Ogni blocco deve avere un commento che ne spiega lo scopo, e i risultati vanno stampati con print e riportati nel testo.

Errori comuni e rimedi

ERRORE	RIMEDIO
Probabilità simulata maggiore di 1	dividere per il numero di simulazioni N, non per il numero di oggetti del problema
randint(1, 6) per un dado	il secondo estremo è escluso: randint(1, 7)
and, or, not su array	usare &, |, ~ con ogni condizione tra parentesi
Esponenziale con parametro sbagliato	np.random.exponential e stats.expon vogliono la media $1/\lambda$ (scale=$1/\lambda$)
Normale con la varianza	stats.norm e np.random.normal vogliono σ, non σ^2
cdf invece di pmf o sf	cdf(k) = $P(X \le k)$, pmf(k) = $P(X = k)$, sf(k − 1) = $P(X \ge k)$
Varianza con il divisore sbagliato	np.var divide per N; per lo stimatore non distorto serve ddof=1
Normale invece della t	se σ è stimata dai dati si usa la t con n − 1 gradi di libertà
Test bilaterale per una domanda direzionale	"maggiore" o "minore" richiedono il test unilaterale
Istogramma non confrontabile con la densità	aggiungere density=True
P ** n per la potenza di matrice	usare np.linalg.matrix_power(P, n); ** agisce elemento per elemento
P @ pi per la distribuzione	il vettore va a sinistra: pi @ P
"Non si rifiuta H_0, quindi H_0 è vera"	scrivere che i dati non forniscono evidenza sufficiente per rifiutarla