

Gradient Descent

-  [Machine Learning 2025/2026](#)
 - [Laboratorio: Funzione di costo, rischio e discesa del gradiente](#)
 - [Obiettivi del laboratorio](#)
 - [Rischio e funzione di costo](#)
 - [Rischio e minimizzazione](#)
 - [Condizioni](#)
 - [Esempio: prevedere la pioggia](#)
 - [1. Funzioni di costo](#)
 - [2. Distribuzione congiunta \$p\(x,y\)\$](#)
 - [3. Modelli predittivi](#)
 - [4. Rischio atteso](#)
 - [♦ Calcolo di esempio: \$f_1\$ con \$L_1\$](#)
 - [♦ Risultati riassuntivi](#)
 - [5. Conclusione](#)
 - [Rischio reale VS Rischio empirico](#)
 - [Minimizzazione della funzione di rischio e modelli lineari](#)
 - [Ricerca analitica dell'ottimo](#)
 - [Forma compatta: il gradiente](#)
 - [Difficoltà pratiche](#)
 - [Gradient Descent](#)
 - [Intuizione](#)
 - [Gradient Descent e dataset](#)
 - [Tre varianti principali](#)
 - [Esempio: gradiente di una parabola in un punto](#)
 - [1 Calcoliamo le derivate parziali](#)
 - [2 Formiamo il gradiente](#)
 - [3 Calcolo del gradiente in un punto specifico](#)
 - [4 Interpretazione geometrica](#)
 - [Let's work](#)
 - [La scelta del modello](#)
 - [La funzione di costo: cross-entropy](#)
 - [Caricamento delle librerie](#)
 - [Scaricare i dati](#)
 - [Carichiamo il dataset \(già scaricato con wget\)](#)
 - [Funzioni di Visualizzazione](#)
 - [Verso la discesa del gradiente](#)

- [1 I parametri del modello](#)
- [2 La funzione di costo](#)
- [3 Il gradiente del rischio](#)
 - [♦ Deriviamo rispetto a \$\theta_j\$](#)
 - [♦ Dalla derivata alla discesa del gradiente](#)
 - [♦ Forma vettoriale compatta](#)
 - [♦ Dalla forma compatta all'aggiornamento iterativo](#)
 - [♦ Cosa sta succedendo?](#)
- [Un piccolo approfondimento riguardo: Binary Cross-Entropy Loss](#)
 - [! Problema di \$\log\(0\)\$](#)
 - [💡 Soluzione pratica](#)
 - [📊 Valori della loss](#)
- [Preparazione dei dati](#)
 - [1 Caricamento dei dati](#)
 - [2 Costruzione delle variabili](#)
 - [3 Output](#)
- [Batch Gradient Descent \(BGD\)](#)
 - [♦ Significato operativo](#)
 - [♦ Procedura tipica](#)
 - [⚙️ Pro e contro](#)
 - [Applicazione del Batch Gradient Descent](#)
 - [1 Parametri di addestramento](#)
 - [2 Esecuzione](#)
 - [3 Analisi dei risultati](#)
 - [💭 Considerazioni sul Batch Gradient Descent](#)
 - [♦ Comportamento tipico](#)
 - [⚙️ Vantaggi](#)
 - [! Limiti](#)
 - [💡 Interpretazione pratica](#)
- [Stochastic Gradient Descent \(SGD\)](#)
 - [♦ Idea di base](#)
 - [⚙️ Procedura di aggiornamento](#)
 - [🧩 Effetti pratici](#)
 - [♦ Aggiornamento dei parametri nella classificazione binaria](#)
 - [Considerazioni sullo Stochastic Gradient Descent](#)
 - [♦ Caratteristiche osservate](#)
 - [♦ Vantaggi](#)
 - [! Limiti](#)
- [Mini-Batch Gradient Descent](#)

- [!\[\]\(6302aad5aed157b291fddf37b4870784_img.jpg\) Idea di base](#)
- [!\[\]\(a9ca2c237943a6d0a9f22252f295b6f3_img.jpg\) Procedura operativa](#)
- [!\[\]\(9a01a64e0b4ff865df7d32ee7991fe8b_img.jpg\) Aggiornamento dei parametri nella classificazione binaria](#)
- [!\[\]\(6aefe9a3d997eb8b55c40ecd5fa7053f_img.jpg\) Effetti pratici](#)
- [!\[\]\(baa8f8ba8c970db55300f5bb45bb3460_img.jpg\) Nella pratica](#)
- [Considerazioni sul Mini-Batch Gradient Descent](#)
 - [!\[\]\(a6e28495607b2299466d3d5d3193848c_img.jpg\) Comportamento osservato](#)
 - [!\[\]\(ed205fcb6e75c95529564351570724d7_img.jpg\) Vantaggi](#)
 - [!\[\]\(27a992a1de9d3e89591e2e26256c5a71_img.jpg\) Limiti](#)
 - [!\[\]\(4e3fbe2ef35291baab7a42cb80921f3b_img.jpg\) In sintesi](#)
- [!\[\]\(7e07afcbfd46dd92c708e363ec417c00_img.jpg\) Confronto dei risultati](#)
- [!\[\]\(7e5084a8da4d5ff6d50d22c09ead9317_img.jpg\) Dulcis in fundo: come vanno questi metodi?](#)
- [Algoritmi avanzati di ottimizzazione](#)

Machine Learning 2025/2026

Laboratorio: Funzione di costo, rischio e discesa del gradiente

Docente: Danilo Croce, Giorgio Gambosi


Obiettivi del laboratorio

- Capire cos'è una **funzione di costo (loss function)** e perché è importante.
- Introdurre il concetto di **rischio** ed **empirical risk**.
- Vedere in pratica come funziona la **discesa del gradiente (gradient descent)**.
- Sperimentare su un **dataset semplice** e visualizzare i risultati.

 Questo laboratorio è pensato per **capire le idee di base**, iniziando ad entrare nei dettagli matematici.

Rischio e funzione di costo

- Il nostro obiettivo è addestrare un algoritmo che faccia **previsioni corrette**.
- Per misurare **quanto sbaglia**, usiamo una **funzione di costo (loss function)**.
- La loss ci dice “quanto costa” una certa predizione (potenzialmente sbagliata) rispetto al valore corretto.

 **Importante:** diverse funzioni di costo possono portare a valutazioni molto diverse. La scelta della loss **dipende dal problema e dalle priorità** (ad esempio: meglio evitare falsi positivi o falsi negativi?).

Rischio e minimizzazione

Un qualunque algoritmo di apprendimento, dato un input x , produce una previsione $f(x)$. La qualità di questa previsione può essere valutata tramite una **funzione di costo** (*loss function*) $L(x_1, x_2)$, dove:

- x_1 è il valore predetto dal modello,
- x_2 è il valore corretto associato a x .

Il valore $L(f(x), y)$ misura quindi quanto “costa” prevedere $f(x)$ invece del valore corretto y .

Dato che il costo dipende dalla coppia (x, y) , per valutare in generale la bontà delle predizioni si considera il **valore atteso** della funzione di costo al variare di x e y , assumendo una distribuzione di probabilità congiunta $p(x, y)$.

- $p(x, y)$ rappresenta la probabilità che il prossimo input sia x e che il valore corretto sia y .
 - Non si assume che ad uno stesso x corrisponda sempre lo stesso y : si considera solo la probabilità condizionata $p(y | x)$, in modo da includere anche il **rumore nelle osservazioni**.
-

Formalmente, indicando con D_x e D_y i domini di x e y , il **rischio** \mathcal{R} associato ad un algoritmo f è definito come:

$$\mathcal{R}(f) = \mathbb{E}_p[L(f(x), y)] = \int_{D_x} \int_{D_y} L(f(x), y) p(x, y) dx dy$$

In altre parole, il rischio misura il **costo medio atteso** di utilizzare $f(x)$ per le predizioni.

Condizioni

Il rischio è calcolato assumendo che:

1. x sia estratto casualmente dalla distribuzione marginale

$$p(x) = \int_{D_y} p(x, y) dy$$

2. il valore corretto y sia estratto casualmente dalla distribuzione condizionata

$$p(y | x) = \frac{p(x, y)}{p(x)}$$

3. il costo sia misurato dalla funzione $L(x_1, x_2)$.

👉 In altre parole, il **rischio** ci dice quanto ci aspettiamo di “pagare” in media se usiamo il modello $f(x)$ per fare previsioni:

- prendiamo un input x a caso,
- osserviamo il suo valore corretto y ,
- calcoliamo quanto il modello sbaglia con la funzione di costo $L(f(x), y)$,
- ripetiamo mentalmente questo processo su tutti i possibili (x, y) , pesandoli con la loro probabilità $p(x, y)$.

Il risultato è il **costo medio atteso delle previsioni**.

Esempio: prevedere la pioggia

Immaginiamo di voler prevedere la **possibilità di pioggia** durante la giornata, date le condizioni del cielo al mattino.

- **Osservazioni possibili:** "sereno" (S), "nuvoloso" (N), "coperto" (C).
- **Etichette reali:** "pioggia" (T) e "non pioggia" (F).
- **Predizioni possibili:** "pioggia" (T) e "non pioggia" (F).

La bontà delle previsioni dipende dalla **funzione di costo** $L : \{T, F\}^2 \mapsto \mathbb{R}$, che assegna un “peso” agli errori.

1. Funzioni di costo

Caso 1 – Costi simmetrici

Sbagliare in un senso o nell'altro è ugualmente spiacevole:

y/pred	T	F
T	0	1
F	1	0

Caso 2 – Costi asimmetrici

Bagnarsi è molto peggio che portare l'ombrello inutilmente:

y/pred	T	F
T	0	1

y/pred	T	F
F	25	0

2. Distribuzione congiunta $p(x, y)$

x/y	T	F
S	.05	.20
N	.25	.25
C	.20	.05

3. Modelli predittivi

x	$f_1(x)$	$f_2(x)$
S	F	F
N	F	T
C	T	T

4. Rischio atteso

Il **rischio** di un classificatore f è definito come

$$\mathcal{R}(f) = \sum_x \sum_y L(y, f(x)) p(x, y)$$

cioè: la media pesata dei costi, usando le probabilità congiunte.

♦ Calcolo di esempio: f_1 con L_1

- Per $x = S$: $f_1(S) = F$
 - se $y = T$: $L(T, F) = 1$, peso $p(S, T) = 0.05 \rightarrow 0.05$
 - se $y = F$: $L(F, F) = 0$, peso $p(S, F) = 0.20 \rightarrow 0$

- Per $x = N$: $f_1(N) = F$
 - se $y = T$: $L(T, F) = 1$, peso $p(N, T) = 0.25 \rightarrow 0.25$
 - se $y = F$: $L(F, F) = 0$, peso $p(N, F) = 0.25 \rightarrow 0$
- Per $x = C$: $f_1(C) = T$
 - se $y = T$: $L(T, T) = 0$, peso $p(C, T) = 0.20 \rightarrow 0$
 - se $y = F$: $L(F, T) = 1$, peso $p(C, F) = 0.05 \rightarrow 0.05$

Totale:

$$\mathcal{R}(f_1, L_1) = 0.05 + 0.25 + 0.05 = 0.35$$

◆ Risultati riassuntivi

- Con L_1 (**costi simmetrici**):
 $\mathcal{R}(f_1) = 0.35$, $\mathcal{R}(f_2) = 0.30 \rightarrow$ **meglio f_2**
- Con L_2 (**costi asimmetrici**):
 $\mathcal{R}(f_1) = 5.80$, $\mathcal{R}(f_2) = 6.05 \rightarrow$ **meglio f_1**

5. Conclusione

La scelta del modello migliore dipende da:

1. **come pesiamo gli errori** (funzione di costo L),
2. **come sono distribuiti i dati** ($p(x, y)$).

Se cambiano i pesi o le probabilità, la decisione può ribaltarsi.

Rischio reale VS Rischio empirico

La distribuzione reale $p(x, y)$ è sconosciuta (se la conoscessimo potremmo prevedere direttamente $p(y | x)$).

Per questo motivo, il **rischio reale** non è calcolabile, e dobbiamo stimarlo a partire dai dati disponibili.

L'approccio standard è usare la **media aritmetica sul training set** come stimatore del valore atteso.

Si definisce quindi il **rischio empirico**:

$$\overline{\mathcal{R}}(f; X) = \frac{1}{n} \sum_{i=1}^n L(f(x_i), y_i)$$

dove $X = \{(x_1, y_1), \dots, (x_n, y_n)\}$ è il training set.

Il modello scelto sarà quello che **minimizza il rischio empirico**:

$$f^* = \operatorname{argmin}_{f \in F} \overline{\mathcal{R}}(f; X)$$

In altre parole:

- si calcola la media degli errori commessi sul training set;
 - si sceglie la funzione f (tra quelle considerate) che produce la media più bassa.
-

Fattori che influenzano la bontà di questa approssimazione:

- **Numero di dati (n)**: più il training set è grande, più $\overline{\mathcal{R}}(f; X)$ si avvicina al rischio reale $\mathcal{R}(f)$.
 - **Distribuzione reale $p(x, y)$** : se è molto complessa, servono più dati per stimarla bene.
 - **Funzione di costo L** : se assegna costi molto elevati a casi rari, può creare problemi.
 - **Classe di funzioni F** :
 - Se troppo ampia e complessa → servono molti dati per non sovrastimare.
 - Se troppo ristretta → si rischia di escludere modelli buoni.
-

Minimizzazione della funzione di rischio e modelli lineari

In generale, l'insieme F delle funzioni può essere descritto in forma **parametrica**:

$$F = \{f(\mathbf{x}; \theta)\}$$

dove $\theta \in D_\theta$ è un insieme di parametri (spesso un vettore) che specifica una particolare funzione all'interno della famiglia F .

Esempio: regressione lineare

Vogliamo prevedere una variabile $y \in \mathbb{R}$ a partire da m attributi $\mathbf{x} = (x_1, \dots, x_m)$.

Restringendo F alle sole funzioni lineari, otteniamo:

$$f_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \dots + w_m x_m$$

dove i parametri sono i coefficienti $\mathbf{w} = (w_0, \dots, w_m)$.

Rischio empirico come funzione dei parametri

$$\overline{\mathcal{R}}(\theta; X) = \frac{1}{n} \sum_{i=1}^n L(f(\mathbf{x}_i; \theta), t_i) \quad f \in F$$

La minimizzazione del rischio empirico consiste nel trovare i parametri θ che lo riducono al minimo:

$$\theta^* = \operatorname{argmin}_{\theta \in D_\theta} \overline{\mathcal{R}}(\theta; X)$$

Da qui deriva la funzione ottima (all'interno della famiglia F):

$$f^* = f(\mathbf{x}; \theta^*)$$

In altre parole:

- scegliamo una famiglia di funzioni F (es. tutte le rette);
- valutiamo quanto bene ogni funzione predice i dati (rischio empirico);
- cerchiamo i **parametri θ ottimali** che rendono minima la media degli errori.

Il metodo per eseguire questa minimizzazione può variare a seconda del problema e della complessità del modello (es. metodi analitici, algoritmi numerici, gradient descent, ecc.).

Ricerca analitica dell'ottimo

Se il problema si pone come una minimizzazione **senza vincoli** (cioè all'interno di \mathbb{R}^m), un primo approccio classico è quello dell'**analisi matematica**: cerchiamo i valori $\bar{\theta}$ di θ per cui si annullano tutte le derivate parziali del rischio empirico.

In formule:

$$\left. \frac{\partial \overline{\mathcal{R}}(\theta; X)}{\partial \theta_i} \right|_{\theta=\bar{\theta}} = 0 \quad i = 1, \dots, m$$

dove m è il numero di componenti del vettore θ . Questo porta a un sistema di m equazioni con m incognite.

Forma compatta: il gradiente

La stessa condizione può essere scritta in forma vettoriale come:

$$\nabla_{\theta} \overline{\mathcal{R}}(\theta; X) = 0$$

dove ∇_{θ} indica il **gradiente** (cioè il vettore delle derivate parziali rispetto ai parametri).

Difficoltà pratiche

- In molti casi la **soluzione analitica** di questo sistema è **troppo complessa** o addirittura impossibile da calcolare.
- Inoltre, il gradiente nullo può corrispondere sia a un **minimo locale**, sia a un "**punto di sella**", non necessariamente a un minimo globale.

Per questi motivi, in pratica si usano spesso **metodi numerici di ottimizzazione** (es. discesa del gradiente, vedi dopo).

Gradient Descent

La **discesa del gradiente** (*gradient descent*) è un metodo **numerico iterativo** per approssimare la soluzione di quel sistema.

Invece di cercare direttamente il punto in cui il gradiente si annulla, **aggiorniamo progressivamente i parametri nella direzione in cui il rischio empirico diminuisce più rapidamente**:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \cdot \nabla_{\theta} \overline{\mathcal{R}}(\theta^{(k)})$$

- $\nabla_{\theta} \overline{\mathcal{R}}(\theta)$ = gradiente del rischio empirico (direzione di massima crescita)
- η = *learning rate*, cioè l'ampiezza del passo

Intuizione

Immagina di essere su una collina al buio e voler scendere a valle:

- il gradiente ti indica dove la salita è più ripida,
- quindi, muovendoti nella **direzione opposta**, scendi più velocemente,
- la lunghezza del passo η controlla la velocità della discesa:
 - se è troppo piccolo, scendi lentamente;
 - se è troppo grande, rischi di "saltare oltre" il minimo.

👉 Si parla di **metodo di primo ordine**, perché utilizza solo le derivate prime (le pendenze) della funzione da minimizzare.

Gradient Descent e dataset

Nel Machine Learning, il rischio empirico si calcola come media delle perdite sui dati:

$$\overline{\mathcal{R}}(\theta) = \frac{1}{n} \sum_{i=1}^n L(f(\mathbf{x}_i; \theta), t_i)$$

Di conseguenza, anche il gradiente è una **media dei gradienti sui singoli esempi**:



$$\nabla_{\theta} \overline{\mathcal{R}}(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} L(f(\mathbf{x}_i; \theta), t_i)$$

Questa osservazione porta a diverse **varianti pratiche** della discesa del gradiente.



Tre varianti principali

Le varianti dipendono da **quanti esempi del dataset vengono usati** per calcolare il gradiente a ogni passo di aggiornamento:



1. Batch Gradient Descent

- Usa *tutti* i dati ad ogni aggiornamento.
-  Aggiornamenti accurati e stabili
-  Molto lento per dataset grandi

2. Stochastic Gradient Descent (SGD)

- Usa **un solo esempio alla volta**.
-  Aggiornamenti velocissimi
-  Molto rumoroso: la funzione di costo oscilla molto

3. Mini-Batch Gradient Descent

- Compromesso: usa piccoli gruppi (batch) di esempi.
-  Equilibrio tra velocità e stabilità
-  È lo **standard attuale** nel Deep Learning

In sintesi:

- Più esempi → aggiornamento accurato ma lento
 - Meno esempi → aggiornamento veloce ma instabile
 - Mini-batch → compromesso ideale (veloce, scalabile e stabile)
-

Esempio: gradiente di una parabola in un punto

Consideriamo la funzione:

$$f(x, y) = x^2 + y^2$$

Questa è una **paraboloide**: una superficie “a ciotola” che ha il suo minimo in corrispondenza dell'origine (0, 0).

1 Calcoliamo le derivate parziali

- Derivata parziale rispetto a x :

$$\frac{\partial f}{\partial x} = 2x$$

- Derivata parziale rispetto a y :

$$\frac{\partial f}{\partial y} = 2y$$

2 Formiamo il gradiente

Il **gradiente** è il vettore che raccoglie tutte le derivate parziali:

$$\nabla f(x, y) = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

3 Calcolo del gradiente in un punto specifico

Scegliamo, ad esempio, il punto $(x, y) = (1, 2)$:

$$\nabla f(1, 2) = \begin{bmatrix} 2(1) \\ 2(2) \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

4 Interpretazione geometrica

- Il gradiente nel punto $(1, 2)$ è il vettore **(2, 4)**.
- Questo vettore **punta nella direzione di massima crescita** della funzione f .
- La sua direzione è quella verso cui la superficie “sale” più rapidamente.
- Se vogliamo **scendere** (cioè trovare il minimo), dobbiamo muoverci nella direzione **opposta** al gradiente, cioè verso **(-2, -4)**.

 **In sintesi:**

Punto	Gradiente	Significato
(0,0)	(0,0)	Minimo (nessuna pendenza)

Punto	Gradiente	Significato
(1,2)	(2,4)	La funzione cresce di più verso (2,4)
(-1,-1)	(-2,-2)	La funzione cresce di più verso (-2,-2), quindi scende verso (1,1)

📍 Il **gradiente** è come una freccia che indica “da che parte sale la montagna” nel punto in cui ti trovi.

Let's work

Per capire meglio come funziona la discesa del gradiente, applichiamo ad un semplice problema di **classificazione binaria**.

Immaginiamo di avere un dataset bidimensionale: ogni punto è descritto da due caratteristiche (x_1, x_2) e da un'etichetta $t \in \{0, 1\}$ che indica la classe di appartenenza.

L'obiettivo è costruire un modello predittivo $f(\mathbf{x}; \theta)$ che, dato un nuovo punto, restituisca un valore vicino a 0 o 1 a seconda della classe.

La scelta del modello

Partiamo con un modello molto semplice: una **combinazione lineare** delle feature (x_1, x_2) con i parametri $\theta = (\theta_0, \theta_1, \theta_2)$:

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

Per trasformare z in un valore compreso tra 0 e 1 (che possiamo interpretare come probabilità di appartenenza alla classe positiva), applichiamo la **funzione sigmoide**:

$$f(\mathbf{x}; \theta) = \sigma(z) = \frac{1}{1 + e^{-z}}.$$

👉 Quindi il modello predice la probabilità che l'osservazione appartenga alla classe 1. Solo più avanti vedremo che questa formulazione prende il nome di **logistic regression**.

La funzione di costo: cross-entropy

Per valutare la bontà delle predizioni usiamo la **cross-entropy loss**, che misura quanto la probabilità stimata $f(\mathbf{x}; \theta)$ si discosta dal valore vero t :

$$L(t, f(\mathbf{x}; \theta)) = -[t \log f(\mathbf{x}; \theta) + (1 - t) \log(1 - f(\mathbf{x}; \theta))].$$

- Se $t = 1$, il costo è grande se $f(\mathbf{x}; \theta)$ è vicino a 0 (cioè se sbagliamo con alta sicurezza).
- Se $t = 0$, il costo è grande se $f(\mathbf{x}; \theta)$ è vicino a 1.

👉 La cross-entropy penalizza fortemente le previsioni sicure ma sbagliate.

Caricamento delle librerie

Qui importiamo alcune librerie fondamentali:

- **NumPy, Pandas** → per la gestione e manipolazione dei dati
- **Matplotlib** → per produrre grafici
- **SciPy** → per funzioni matematiche avanzate
- **time** → per misurare i tempi di esecuzione

Molte di queste librerie sono già presenti in Colab, mentre in locale potrebbero dover essere installate (ad esempio con `conda install` o `pip install`).

```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import scipy as scipy
import scipy.special as sp
import pandas as pd

import time
```

Scaricare i dati

Per lavorare su un problema pratico ci serve un **dataset**. In questo laboratorio i dati non sono già presenti dentro Colab, quindi dobbiamo **scaricarli da una sorgente esterna** (in questo caso un link fornito dal docente).

In Colab (e in generale nei notebook Jupyter) il simbolo `!` davanti a un comando permette di eseguire un **comando di shell** direttamente dal notebook, come se fossimo in un terminale Linux.

- `wget` è un programma da riga di comando che serve per **scaricare file da Internet** (ad esempio dataset, script, immagini).
- Lo usiamo qui per recuperare il file `testSet.txt` dal link indicato.

In questo modo il file viene salvato nell'ambiente di esecuzione di Colab e possiamo poi leggerlo con Python (ad esempio con `pandas.read_csv`) ed utilizzarlo per i nostri esperimenti.

```
!wget https://tvm1.github.io/ml2425/dataset/testSet.txt
```

Prima di procedere con la costruzione del modello, è utile dare uno sguardo al dataset su cui lavoreremo.

Il file `testSet.txt` contiene tre colonne:

- **x1** e **x2**: le due caratteristiche (coordinate) che descrivono ogni punto;
- **t**: l'etichetta della classe, che può assumere valore 0 o 1.

Nelle prossime celle mostriamo alcune righe del dataset e la distribuzione delle classi, così da avere un'idea più concreta dei dati su cui applicheremo la discesa del gradiente.

Carichiamo il dataset (già scaricato con wget)

```
data = pd.read_csv("testSet.txt",
                   delim_whitespace=True,
                   header=None,
                   names=['x1', 'x2', 't'])
```

Funzioni di Visualizzazione

Per rendere il laboratorio più chiaro, definiamo alcune **funzioni di supporto** che servono a:

- Visualizzare i dati in un grafico 2D
- Tracciare l'andamento della funzione di costo durante l'ottimizzazione
- Mostrare come evolvono i parametri del modello durante il training

👉 Queste funzioni non fanno parte del modello in sé, ma aiutano a **capire cosa succede** in modo visivo.

```
colors = ["xkcd:dusty blue", "xkcd:dark peach", "xkcd:dark seafoam green",
          "xkcd:dusty purple", "xkcd:watermelon", "xkcd: dusky blue",
          "xkcd:amber",
          "xkcd:purplish", "xkcd:dark teal", "xkcd:orange", "xkcd:slate"]
```

```
plt.style.use('ggplot')
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.serif'] = 'Ubuntu'
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 10
plt.rcParams['axes.labelsize'] = 10
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titlesize'] = 10
plt.rcParams['xtick.labelsize'] = 8
```

```

plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['legend.fontsize'] = 10
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['image.cmap'] = 'jet'
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['figure.figsize'] = (16, 8)
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.markersize'] = 8
def plot_ds(data, m=None, q=None):
    fig = plt.figure(figsize=(16,8))
    x = np.linspace(data.x1.min()*1.1, data.x1.max()*1.1, 1000)
    ax = plt.gca()
    ax.scatter(data[data.t==0].x1, data[data.t==0].x2, s=40,
edgecolor='k', alpha=.7)
    ax.scatter(data[data.t==1].x1, data[data.t==1].x2, s=40,
edgecolor='k', alpha=.7)
    if (m is not None) and (q is not None):
        ax.plot(x, m*x + q, lw=2) # colore di default ok
    ax.set_xlabel('$x_1$'); ax.set_ylabel('$x_2$');
ax.set_title('Dataset')
plt.show()
def plot_all(cost_history, m, q, low=0, high=None, step=1):
    if high is None:
        high = m.shape[0]
    idx = range(low, high, step)
    ch = cost_history[idx].ravel()
    th1 = m[idx]
    th0 = q[idx]

    fig = plt.figure(figsize=(18,6))
    ax = fig.add_subplot(1,2,1)
    ax.plot(range(len(ch)), ch, linewidth=2)
    ax.set_xlabel('iterations'); ax.set_ylabel('cost')
    ax.xaxis.set_major_formatter(mpl.ticker.FuncFormatter(
        lambda x, pos: f'{x*step+low:0.0f}'))

    ax = fig.add_subplot(1,2,2)
    x_min, x_max = th0.min(), th0.max()
    y_min, y_max = th1.min(), th1.max()
    dx, dy = .1*(x_max-x_min+1e-12), .1*(y_max-y_min+1e-12)
    ax.plot(th0, th1, linewidth=2)
    ax.scatter(th0[-1], th1[-1], marker='o', s=40)
    ax.set_xlabel(r'$q$'); ax.set_ylabel(r'$m$')

```



```
ax.set_xlim(x_min-dx, x_max+dx); ax.set_ylim(y_min-dy, y_max+dy)
plt.tight_layout(); plt.show()
```

Per visualizzare meglio la natura del problema, rappresentiamo i punti del dataset nel piano delle caratteristiche (x_1, x_2) .

Ogni punto è colorato in base alla sua etichetta t :

- un colore per la **classe 0**
- un colore diverso per la **classe 1**

In questo modo possiamo osservare come i dati siano distribuiti e intuire la possibilità di separarli mediante una retta.

plot_ds(data)

Verso la discesa del gradiente

Una volta definita la funzione di rischio empirico $\mathcal{R}_n(\theta)$, il passo successivo è capire **come modificarla** per trovare i parametri θ che la minimizzano.

1 I parametri del modello

Indichiamo con $\theta = (\theta_0, \theta_1, \theta_2)$ il vettore dei **parametri** del modello.

Nel nostro caso, essi determinano la retta che separa le due classi nel piano delle feature (x_1, x_2) :

$$z = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

La funzione del modello, che restituisce la probabilità stimata di appartenere alla classe positiva, è:

$$f(\mathbf{x}; \theta) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

2 La funzione di costo

Per misurare la bontà delle predizioni del modello utilizziamo la **cross-entropy loss**:

$$L(t, f(\mathbf{x}; \theta)) = -[t \log f(\mathbf{x}; \theta) + (1 - t) \log(1 - f(\mathbf{x}; \theta))]$$

dove:

- $t \in \{0, 1\}$ è il **valore reale** (target) associato al campione \mathbf{x} ;
- $f(\mathbf{x}; \theta)$ è la **predizione del modello**, ovvero la probabilità stimata che \mathbf{x} appartenga alla classe positiva.

Di conseguenza, il **rischio empirico medio** su n osservazioni diventa:

$$\mathcal{R}_n(\theta) = \frac{1}{n} \sum_{i=1}^n L(t_i, f(\mathbf{x}_i; \theta)).$$

3 Il gradiente del rischio

Per minimizzare $\mathcal{R}_n(\theta)$ dobbiamo calcolare **come cambia** la funzione di costo media rispetto ai parametri $\theta = (\theta_0, \theta_1, \theta_2)$.

Il **gradiente** è il vettore delle derivate parziali:

$$\nabla_{\theta} \mathcal{R}_n(\theta) = \begin{bmatrix} \frac{\partial \mathcal{R}_n}{\partial \theta_0} \\ \frac{\partial \mathcal{R}_n}{\partial \theta_1} \\ \frac{\partial \mathcal{R}_n}{\partial \theta_2} \end{bmatrix}.$$

◆ Deriviamo rispetto a θ_j

Partiamo dalla funzione di rischio empirico media:

$$\mathcal{R}_n(\theta) = -\frac{1}{n} \sum_{i=1}^n [t_i \log f_i + (1 - t_i) \log(1 - f_i)], \quad f_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}}, \quad z_i = \theta^\top x_i.$$

Applichiamo la **chain rule** (ricordiamo che \mathcal{R} dipende da f che dipende da θ) per derivare rispetto a un singolo parametro θ_j :

$$\frac{\partial \mathcal{R}_n}{\partial \theta_j} = -\frac{1}{n} \sum_{i=1}^n \left[\frac{t_i}{f_i} \frac{\partial f_i}{\partial \theta_j} - \frac{1 - t_i}{1 - f_i} \frac{\partial f_i}{\partial \theta_j} \right].$$

Poiché la funzione f_i dipende da θ_j tramite $z_i = \theta^\top x_i$, possiamo scomporre la derivata come:

$$\frac{\partial f_i}{\partial \theta_j} = \frac{\partial f_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial \theta_j}.$$

- Derivata della **sigmoide**. Se $f_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}}$, allora

$$\frac{\partial f_i}{\partial z_i} = \frac{e^{-z_i}}{(1 + e^{-z_i})^2} = \left(\frac{1}{1 + e^{-z_i}} \right) \left(\frac{e^{-z_i}}{1 + e^{-z_i}} \right) = \sigma(z_i) (1 - \sigma(z_i)) = f_i (1 - f_i).$$

- Derivata della parte **lineare**:

$$\frac{\partial z_i}{\partial \theta_j} = x_{ij}$$

Quindi:

$$\frac{\partial f_i}{\partial \theta_j} = f_i(1 - f_i)x_{ij}.$$

Sostituendo nella formula precedente

$$\begin{aligned} \frac{\partial \mathcal{R}_n}{\partial \theta_j} &= -\frac{1}{n} \sum_{i=1}^n \left[\frac{t_i}{f_i} \frac{\partial f_i}{\partial \theta_j} - \frac{1-t_i}{1-f_i} \frac{\partial f_i}{\partial \theta_j} \right] = \\ &= -\frac{1}{n} \sum_{i=1}^n \left[\frac{t_i}{f_i} f_i(1-f_i)x_{ij} - \frac{1-t_i}{1-f_i} f_i(1-f_i)x_{ij} \right]. \end{aligned}$$

e semplificando i termini algebricamente:

$$\left(-\frac{t_i}{f_i} + \frac{1-t_i}{1-f_i} \right) f_i(1-f_i) = -t_i(1-f_i) + (1-t_i)f_i = f_i - t_i.$$

Otteniamo infine:

$$\frac{\partial \mathcal{R}_n}{\partial \theta_j} = \frac{1}{n} \sum_{i=1}^n (f_i - t_i)x_{ij} = -\frac{1}{n} \sum_{i=1}^n (t_i - f_i)x_{ij}.$$

dove:

- $f_i = f(\mathbf{x}_i; \theta) = \sigma(\theta^\top \mathbf{x}_i)$ è la **predizione del modello**,
cioè la probabilità stimata che l'osservazione i appartenga alla classe positiva;
- $\sigma(z) = \frac{1}{1 + e^{-z}}$ è la **funzione sigmoide**;
- $t_i \in \{0, 1\}$ è il **valore reale (target)** associato al campione i ;
- x_{ij} è il **valore della feature** j per il campione i .

◆ Dalla derivata alla discesa del gradiente

Abbiamo trovato l'espressione della derivata parziale di $\mathcal{R}_n(\theta)$ rispetto a ogni parametro θ_j . Ora possiamo combinare tutte le componenti e scrivere il **gradiente completo**:

$$\nabla_{\theta} \mathcal{R}_n(\theta) = -\frac{1}{n} \sum_{i=1}^n (t_i - f_i) \mathbf{x}_i,$$

dove \mathbf{x}_i è il vettore delle feature del campione i

e $f_i = f(\mathbf{x}_i; \theta) = \sigma(\theta^\top \mathbf{x}_i)$ è la previsione del modello.

👉 Questa è la **media dei contributi** di tutti i dati di training: ogni campione “spinge” i parametri nella direzione dell'errore $(t_i - f_i)$.

♦ Forma vettoriale compatta

Scrivendo la somma precedente in forma matriciale, otteniamo:

$$\nabla_{\theta} \mathcal{R}_n(\theta) = -\frac{1}{n} X^{\top} (t - f(\theta, X)).$$

Qui:

- X è la **matrice dei dati** $(n \times d)$, con una riga per ogni campione e una colonna per ogni feature;
 - X^{\top} è la **trasposta** $(d \times n)$, che “raccolge” i contributi di tutti i campioni e restituisce un vettore $(d \times 1)$;
 - $(t - f(\theta, X))$ è il vettore $(n \times 1)$ degli **errori di predizione**;
 - il prodotto $X^{\top} (t - f(\theta, X))$ realizza la stessa **somma ponderata** della forma con la sommatoria, ma in modo più compatto ed efficiente.
-

♦ Dalla forma compatta all'aggiornamento iterativo

Ora possiamo scrivere la **regola generale della discesa del gradiente**:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \cdot \nabla_{\theta} \overline{\mathcal{R}}(\theta^{(k)}),$$

e, sostituendo il gradiente appena calcolato:

$$\theta^{(k+1)} = \theta^{(k)} + \frac{\eta}{n} X^{\top} (t - f(\theta^{(k)}, X)).$$

♦ Cosa sta succedendo?

- $(t - f(\theta, X))$ misura **quanto sbaglia** il modello su ciascun campione;
 - X^{\top} combina questi errori in base alle feature, fornendo la **direzione complessiva** in cui aggiornare i pesi;
 - $\frac{\eta}{n}$ controlla la **velocità** dell'aggiornamento medio.
-

👉 In sintesi:

1. Calcoliamo il gradiente come media degli errori ponderati:

$$\nabla_{\theta} \mathcal{R}_n(\theta) = -\frac{1}{n} \sum_{i=1}^n (t_i - f_i) \mathbf{x}_i.$$

2. Aggiorniamo θ muovendoci nella direzione opposta al gradiente.

3. Ripetiamo il processo per più **epoche**, fino alla convergenza.

Questa è la **discesa del gradiente batch**, la base da cui deriveranno le versioni **stocastica** e **mini-batch**.

Un piccolo approfondimento riguardo: Binary Cross-Entropy Loss

La **funzione di costo binaria** (*binary cross-entropy loss*) è:

$$L(t, \hat{y}) = -[t \log(\hat{y}) + (1 - t) \log(1 - \hat{y})]$$

dove:

- $t \in \{0, 1\}$ è il **valore target (vero)**,
- $\hat{y} \in [0, 1]$ è la **probabilità predetta** dal modello.

⚠ Problema di $\log(0)$

Quando la predizione è esattamente 0 o 1, compare il termine $\log(0)$, che è **infinito**:

- se $t = 1$ e $\hat{y} = 0 \Rightarrow L = \infty$
- se $t = 0$ e $\hat{y} = 1 \Rightarrow L = \infty$

💡 Soluzione pratica

In Python (e in generale nei calcoli numerici) non si usano mai 0 e 1 "puri", ma si sostituiscono con valori leggermente interni all'intervallo $([0, 1])$, ad esempio $\epsilon = 10^{-15}$. In questo modo la funzione resta **stabile e finita**.

📊 Valori della loss

Predizione \hat{y}	$L(t=0, \hat{y}) = -\log(1 - \hat{y})$	$L(t=1, \hat{y}) = -\log(\hat{y})$
0.00	0.000	∞
0.25	0.288	1.386
0.50	0.693	0.693
0.75	1.386	0.288
1.00	∞	0.000

In sintesi:

La *cross-entropy* misura quanto la predizione è coerente col target.

È piccola quando il modello è sicuro **e ha ragione**, ed esplode quando è sicuro **ma sbaglia**.

In Python si evita l'infinito introducendo un piccolo **epsilon (ϵ)** per stabilizzare il calcolo, e invece di ∞ si ottiene... 34!

```
## 1 Funzione del modello (sigmoide logistica)
```

```
def f(theta, X):
```

```
    """
```

```
    Calcola le probabilità predette dal modello.
```

```
     $f(x; \theta) = \sigma(\theta^T x) = 1 / (1 + e^{(-\theta^T x)})$ 
```

```
    Dove:
```

- θ è il vettore dei parametri (bias + pesi)
- X è la matrice dei dati (con una colonna di 1 per il bias)
- σ è la funzione sigmoide logistica

```
    La sigmoide converte la combinazione lineare  $z = \theta^T x$ 
    in un valore compreso tra 0 e 1, interpretabile come probabilità
    che il campione appartenga alla classe positiva ( $t = 1$ ).
```

```
    Usiamo scipy.special.expit(z) invece di 1/(1+exp(-z))
    perché è una versione numericamente stabile che evita overflow.
```

```
    """
```

```
    z = np.dot(X, theta) # combinazione lineare delle feature
    return sp.expit(z).reshape(-1, 1) # applica la sigmoide logistica
```

2 Funzione di costo (cross-entropy)

```
def cost(theta, X, t):
```

```
    """
```

```
    Calcola la *cross-entropy loss* media sul dataset.
```

```
     $L(t, f(x;\theta)) = - [ t \log(f(x;\theta)) + (1 - t) \log(1 - f(x;\theta)) ]$ 
```

```
    Questa misura penalizza fortemente le previsioni errate
    fatte con alta sicurezza (valori di  $f(x;\theta)$  molto vicini a 0 o 1).
```

```
    Parametri:
```

- θ : vettore dei parametri del modello
- X: matrice dei dati (n x d)
- t: vettore delle etichette (n x 1), con valori 0 o 1

```
    Nota numerica:
```

```
    Per evitare errori di  $\log(0)$ , i valori di  $f(x;\theta)$  sono "clippati"
    in un intervallo  $[\text{eps}, 1 - \text{eps}]$ .
```

```
    """
```

```
    eps = 1e-15 # piccolo valore per evitare  $\log(0)$ 
```

```
    v = np.clip(f(theta, X), eps, 1 - eps)
```

```
    term1 = np.dot(np.log(v).T, t)
```

```
    term2 = np.dot(np.log(1 - v).T, (1 - t))
```

```
    return ((-term1 - term2) / len(X))[0]
```

3 Gradiente della funzione di costo

```
def gradient(theta, X, t):
```

```
    """
```

```
    Calcola il gradiente della funzione di costo rispetto ai parametri  $\theta$ .
```

```
     $\nabla_{\theta} J(\theta) = - (1/n) * X^T (t - f(\theta, X))$ 
```

```
    Dove:
```

- $(t - f(\theta, X))$ rappresenta l'errore di predizione su ogni campione
- $X^T (t - f)$ propaga questi errori pesandoli in base alle feature
- La divisione per n restituisce la media del gradiente sul dataset

```
    Il gradiente indica la direzione di massima crescita della funzione
    di costo: per minimizzarla, durante la discesa del gradiente ci
```

```
    muoviamo
```

```
    nella direzione opposta a questo vettore.
```

```
    """
```

```
    return -np.dot(X.T, (t - f(theta, X))) / len(X)
```

Preparazione dei dati

Prima di applicare i metodi di ottimizzazione, dobbiamo preparare il dataset in una forma adatta al calcolo numerico.

Il file `testSet.txt` contiene tre colonne:

- **x1, x2** → le due caratteristiche (feature) che descrivono ciascun campione;
 - **t** → l'etichetta di classe (0 o 1).
-

1 Caricamento dei dati

Utilizziamo la libreria `pandas` per leggere il file di testo e creare un `DataFrame`. Questo ci permette di ispezionare facilmente i dati e di convertirli in array NumPy per le operazioni successive.

2 Costruzione delle variabili

- **X** → matrice delle feature, con due colonne (x1, x2);
- **t** → vettore colonna con le etichette di classe;
- **X con bias** → aggiungiamo una colonna di 1 per gestire il termine noto (bias) nel modello lineare.

In pratica:

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} \end{bmatrix}, \quad t = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_n \end{bmatrix}$$

3 Output

Al termine, otteniamo:

- `n` → numero di esempi nel dataset,
- `nfeatures` → numero di feature per esempio,
- `X` e `t` pronti per essere utilizzati negli algoritmi di ottimizzazione.

Batch Gradient Descent (BGD)

Abbiamo appena derivato la regola generale della discesa del gradiente:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla_{\theta} \mathcal{R}_n(\theta^{(k)}),$$

e, nel caso della **cross-entropy loss**, il gradiente risulta:

$$\nabla_{\theta} \mathcal{R}_n(\theta) = -\frac{1}{n} X^{\top} (t - f(\theta, X)).$$

Sostituendo nella formula di aggiornamento otteniamo la forma **batch** della discesa del gradiente:

$$\theta^{(k+1)} = \theta^{(k)} + \frac{\eta}{n} X^{\top} (t - f(\theta^{(k)}, X))$$

◆ Significato operativo

- L'aggiornamento è calcolato su **tutto il dataset** → gradiente “preciso” ma più lento da calcolare.
- $(t - f(\theta, X))$ rappresenta gli **errori di predizione**; moltiplicando per X^{\top} si aggregano gli effetti di tutti i campioni.
- Il termine η/n controlla la **dimensione media del passo**.

◆ Procedura tipica

1. Inizializza θ (a zero o valori casuali piccoli);
2. Per ogni **epoca**:
 - Calcola $f(\theta, X) = \sigma(X\theta)$;
 - Calcola il gradiente medio $-\frac{1}{n} X^{\top} (t - f(\theta, X))$;
 - Aggiorna i parametri θ secondo la regola sopra;
 - Registra il valore della funzione di costo per monitorare la convergenza.

⚙️ Pro e contro

✓ Vantaggi	✗ Limiti
Aggiornamenti stabili e regolari	Computazionalmente costoso su dataset grandi
Convergenza fluida	Poco flessibile per dati in streaming

👉 In pratica, il **Batch Gradient Descent (BGD)** è la versione “completa” della discesa del gradiente:

usa tutti i dati a ogni passo, fornendo la base teorica da cui nasceranno le versioni **stocastica** e **mini-batch**, più efficienti nei contesti reali.

```
def batch_gd(X, t, eta=0.1, epochs=10000):
    """
    Implementa il Batch Gradient Descent (BGD) per la regressione
    logistica.

    Ad ogni iterazione:
    - calcola il gradiente della funzione di costo su *tutto il dataset*;
    - aggiorna i parametri  $\theta$  nella direzione opposta al gradiente;
    - salva i valori di  $\theta$  e della funzione di costo per analizzare la
    convergenza.

    Parametri:
    -----
    X : ndarray (n x d)
        Matrice dei dati (inclusa la colonna di 1 per il bias).
    t : ndarray (n x 1)
        Vettore delle etichette (0 o 1).
    eta : float
        Learning rate – controlla la dimensione dei passi.
    epochs : int
        Numero totale di iterazioni (passate su tutto il dataset).

    Ritorna:
    -----
    cost_history : ndarray
        Valori della funzione di costo a ogni iterazione.
    theta_history : ndarray
        Valori dei parametri  $\theta$  a ogni passo.
    m, q : ndarray
        Pendenza e intercetta della retta di separazione nel piano (x1,
    x2).
    """

    # 1 Inizializzazione dei parametri del modello (bias + 2 pesi)
    theta = np.zeros(nfeatures + 1).reshape(-1, 1)

    # Liste per memorizzare l'evoluzione dei parametri e del costo
    theta_history = []
```

```

cost_history = []

# 2 Ciclo principale di ottimizzazione
for k in range(epochs):
    # Calcolo del gradiente sul dataset completo
    grad = gradient(theta, X, t)

    # Aggiornamento dei parametri (discesa nella direzione opposta al
    # gradiente)
    delta = -eta * grad
    theta = theta + delta

    # Salvataggio dello stato corrente (per analisi successive)
    theta_history.append(theta)
    cost_history.append(cost(theta, X, t))

# 3 Conversione in array NumPy per elaborazioni successive
theta_history = np.array(theta_history).reshape(-1, 3)
cost_history = np.array(cost_history).reshape(-1, 1)

# 4 Estrazione di m e q (parametri della retta di separazione nel
# piano dei dati)
m = -theta_history[:, 1] / theta_history[:, 2]
q = -theta_history[:, 0] / theta_history[:, 2]

# Ritorno dei risultati
return cost_history, theta_history, m, q

```

Applicazione del Batch Gradient Descent

Ora possiamo mettere in pratica il metodo **Batch Gradient Descent (BGD)** sul dataset preparato.

1 Parametri di addestramento

Per eseguire il metodo, dobbiamo scegliere:

- **Learning rate η :** controlla la velocità di aggiornamento dei parametri.
 - Un valore troppo piccolo → convergenza lenta
 - Un valore troppo grande → oscillazioni o divergenza
- **Numero di epoche:** quante volte scansioniamo tutto il dataset.

2 Esecuzione

Durante l'addestramento:

- ad ogni iterazione, si calcola il gradiente sull'intero dataset;
 - si aggiornano i parametri θ ;
 - si salvano i valori della funzione di costo e dei parametri per analizzare la convergenza.
-

3 Analisi dei risultati

Dopo il training:

- osserviamo il tempo di esecuzione,
- il numero di iterazioni effettuate,
- la discesa della funzione di costo nel tempo,
- e la traiettoria dei coefficienti della retta di separazione nel piano dei parametri (m, q) .

L'andamento regolare del costo mostra la **convergenza stabile** tipica del *Batch Gradient Descent*.

Considerazioni sul Batch Gradient Descent

Il comportamento del **Batch Gradient Descent (BGD)** è quello di una discesa regolare e stabile della funzione di costo, indice di una **convergenza controllata** verso un minimo.

♦ Comportamento tipico

Osservando il grafico della funzione di costo $\mathcal{R}(\theta)$:

- si nota una **decrescita monotona**, che tende a stabilizzarsi in prossimità del minimo;
- la traiettoria dei parametri (m, q) nello spazio dei coefficienti mostra un percorso **graduale e diretto** verso la soluzione ottimale (m^*, q^*) .

Questo riflette il fatto che, a ogni iterazione, l'algoritmo utilizza **tutte le informazioni del dataset** per stimare la direzione di discesa più accurata possibile.

Vantaggi

- **Stabilità:** la funzione di costo decresce in modo regolare, senza oscillazioni improvvise.
 - **Precisione:** ogni aggiornamento tiene conto di tutto il dataset, fornendo una direzione di discesa affidabile.
 - **Convergenza garantita** (nelle funzioni convesse):
 - se $\mathcal{R}(\theta)$ è convessa \rightarrow raggiunge il minimo globale;
 - se non è convessa (es. reti neurali) \rightarrow converge a un minimo locale stabile.
-

⚠ Limiti

- **Costo computazionale elevato:** ogni aggiornamento richiede di calcolare il gradiente su tutti i n campioni.
 - 👉 Diventa rapidamente inefficiente per dataset molto grandi.
 - **Memoria:** l'intero dataset deve risiedere in RAM per ogni iterazione.
 - **Aggiornamenti lenti:** i parametri vengono aggiornati solo una volta per epoca, rallentando la convergenza.
-

💡 Interpretazione pratica

Il *Batch Gradient Descent* rappresenta:

- il modo più **intuitivo e pulito** di capire la logica dell'ottimizzazione basata sul gradiente;
 - una **baseline teorica** utile per confrontare le versioni successive (*Stochastic* e *Mini-Batch*);
 - una scelta valida per problemi **di piccole dimensioni** o in fase di analisi preliminare.
-

📌 In sintesi:

- Il BGD è **preciso ma lento**.
- È ideale per dataset piccoli o per visualizzare il processo di convergenza.
- Le varianti successive (*SGD* e *Mini-Batch*) mantengono la stessa idea di fondo, ma offrono un compromesso diverso tra **velocità, stabilità e accuratezza**.

Stochastic Gradient Descent (SGD)

Dopo aver analizzato il *Batch Gradient Descent*, passiamo a una sua variante più "agile": la **Stochastic Gradient Descent (SGD)**.

◆ Idea di base

Nel *Batch Gradient Descent*, il gradiente viene calcolato ad ogni iterazione usando **tutti gli esempi** del dataset.

Questo garantisce stabilità, ma è computazionalmente costoso.

La **Stochastic Gradient Descent** risolve il problema aggiornando i parametri **dopo ogni singolo esempio**.

In pratica:

- il gradiente viene calcolato su **un solo punto alla volta**;
- i parametri θ vengono aggiornati **immediatamente**.

Matematicamente:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla_{\theta} \mathcal{R}(\theta^{(k)}; \mathbf{x}_i)$$

dove \mathbf{x}_i è il singolo campione scelto all'iterazione corrente.

⚙️ Procedura di aggiornamento

Per ogni epoca:

1. si mescola il dataset (per evitare effetti dovuti all'ordine dei dati);
 2. per ogni esempio \mathbf{x}_i :
 - si calcola il gradiente della *loss* su quell'esempio;
 - si aggiornano immediatamente i parametri θ .
-

🧩 Effetti pratici

- Gli aggiornamenti frequenti rendono il metodo **molto più veloce**.
- Tuttavia, il gradiente stimato su un solo esempio è **rumoroso**: la traiettoria di discesa non è regolare ma “zigzagante”.

La funzione di costo mostra tipicamente **oscillazioni** attorno a un trend decrescente.

Questo rumore, però, può essere utile: permette al metodo di **uscire da minimi locali** e continuare la ricerca di soluzioni migliori.

◆ Aggiornamento dei parametri nella classificazione binaria

Nel nostro caso (funzione sigmoide e loss di tipo cross-entropy):

$$\begin{aligned}\theta_j^{(k+1)} &= \theta_j^{(k)} + \eta(t_i - f(\mathbf{x}_i; \theta^{(k)}))x_{ij} \quad \text{per } j = 1, \dots, d \\ \theta_0^{(k+1)} &= \theta_0^{(k)} + \eta(t_i - f(\mathbf{x}_i; \theta^{(k)}))\end{aligned}$$

Ogni campione contribuisce subito a correggere i pesi in base all'errore di predizione. Nel tempo, gli aggiornamenti si “bilanciano”, e θ converge verso una soluzione stabile.

 In sintesi:

Caratteristica	Batch GD	Stochastic GD
Aggiornamento	dopo ogni epoca	dopo ogni campione
Stabilità	alta	bassa (rumore elevato)
Velocità	lenta	veloce
Scalabilità	bassa	ottima
Adatto a	dataset piccoli	dataset grandi

Il comportamento oscillante è il prezzo da pagare per l'efficienza e la scalabilità.

```
def stochastic_gd(X, t, eta=0.01, epochs=1000):
    """
    Implementa lo Stochastic Gradient Descent (SGD) per la regressione
    logistica.

    A differenza del Batch Gradient Descent, qui il gradiente viene
    calcolato
    e applicato su **un solo campione alla volta**, aggiornando i
    parametri
    immediatamente dopo ogni esempio.

    Monitoraggio:
    - Per visualizzare l'andamento della funzione di costo, la 'loss'
      viene calcolata sull'INTERO dataset (X, t) e salvata in
      cost_history.
    - ⚠ Attenzione: questo NON influisce sull'aggiornamento, serve solo a
      tracciare la curva di convergenza.
```

```

Parametri:
-----
X : ndarray (n x d)
    Matrice dei dati (con colonna di 1 per il bias).
t : ndarray (n x 1)
    Vettore delle etichette (0 o 1).
eta : float
    Learning rate – controlla l'ampiezza del passo di aggiornamento.
epochs : int
    Numero di epoche (ogni epoca scansiona tutto il dataset una
volta).

Ritorna:
-----
cost_history : ndarray
    Valori della funzione di costo durante l'addestramento.
theta_history : ndarray
    Valori dei parametri  $\theta$  ad ogni aggiornamento.
m, q : ndarray
    Pendenza e intercetta della retta di separazione nel piano (x1,
x2).
"""

# 1 Inizializzazione dei parametri del modello
theta = np.zeros(nfeatures + 1).reshape(-1, 1)

# Liste per tenere traccia dell'evoluzione dei parametri e del costo
theta_history = []
cost_history = []

# 2 Ciclo principale su tutte le epoche
for j in range(epochs):

    # (opzionale ma consigliato) – mescoliamo i dati a ogni epoca
    # per evitare che l'ordine degli esempi influenzi l'apprendimento
    idx = np.random.permutation(len(X))
    X, t = X[idx], t[idx]

    # 3 Aggiornamento dei parametri per ogni campione
    for i in range(n):
        # Calcola il gradiente rispetto al singolo campione (riga
X[i])

        grad_i = gradient(theta, X[i, :].reshape(1, -1), t[i])

```



```

        # Aggiorna immediatamente i parametri nella direzione opposta
al gradiente
        theta = theta - eta * grad_i

        # Salva lo stato corrente
        theta_history.append(theta)
        cost_history.append(cost(theta, X, t)) # costo calcolato
sull'intero dataset

# 4 Conversione dei risultati in array per analisi successive
theta_history = np.array(theta_history).reshape(-1, 3)
cost_history = np.array(cost_history).reshape(-1, 1)

# 5 Calcolo della pendenza e intercetta della retta di separazione
m = -theta_history[:, 1] / theta_history[:, 2]
q = -theta_history[:, 0] / theta_history[:, 2]

return cost_history, theta_history, m, q

```

Considerazioni sullo Stochastic Gradient Descent

Il comportamento osservato è molto diverso rispetto al *Batch Gradient Descent*.

◆ Caratteristiche osservate

- La funzione di costo **oscilla** durante l'addestramento, ma mostra un **trend complessivamente decrescente**.
- I parametri (m, q) seguono una traiettoria irregolare, ma tendono a stabilizzarsi in prossimità del minimo.
- Il metodo converge rapidamente verso una buona soluzione, anche se meno precisa del caso batch.

◆ Vantaggi

- **Efficienza**: ogni aggiornamento usa solo un campione → perfetto per dataset grandi.
- **Scalabilità**: adatto a flussi di dati continui (*online learning*).
- **Flessibilità**: l'aggiornamento immediato può aiutare a uscire dai minimi locali.

! Limiti

- **Oscillazioni:** la discesa non è regolare, e la funzione di costo può anche aumentare localmente.
- **Rumore:** il gradiente stimato su un singolo campione è poco accurato.
- **Scelta sensibile di η :** un learning rate troppo grande può causare divergenza.

■ In sintesi:

Lo *Stochastic Gradient Descent* è un metodo **più reattivo e scalabile**, ideale per problemi di grandi dimensioni.

Pur introducendo rumore e oscillazioni, rappresenta un passo fondamentale verso le varianti moderne (come *Mini-Batch*, *Momentum*, e *Adam*), che mirano a combinare **efficienza e stabilità**.

Mini-Batch Gradient Descent

Il **Mini-Batch Gradient Descent (MBGD)** rappresenta un compromesso efficace tra i due approcci estremi:

- il *Batch Gradient Descent* (che utilizza tutti i dati per ogni aggiornamento),
- e lo *Stochastic Gradient Descent* (che aggiorna i parametri ad ogni singolo campione).

◆ Idea di base

L'idea è di dividere il dataset in **sottoinsiemi (mini-batch)** di dimensione s .

A ogni iterazione, il gradiente viene calcolato **su un intero mini-batch**, e i parametri vengono aggiornati in base alla media dei gradienti calcolati sui campioni di quel gruppo.

Matematicamente:

$$\theta^{(k+1)} = \theta^{(k)} - \frac{\eta}{s} \sum_{\mathbf{x} \in X_i} \nabla_{\theta} \mathcal{R}(\theta^{(k)}; \mathbf{x})$$

dove:

- $X_i \subset X$ è il mini-batch corrente,
- $s = |X_i|$ è la sua dimensione,
- η è il *learning rate*.

Procedura operativa

Per ogni epoca:

1. si suddivide il dataset in $\lceil n/s \rceil$ mini-batch;
2. per ciascun mini-batch:
 - si calcola il gradiente medio sui suoi campioni;
 - si aggiornano i parametri θ .

♦ Aggiornamento dei parametri nella classificazione binaria

Nel nostro caso (con funzione sigmoide e *cross-entropy*), gli aggiornamenti diventano:

$$\begin{aligned}\theta_j^{(k+1)} &= \theta_j^{(k)} + \frac{\eta}{s} \sum_{\mathbf{x} \in X_i} (t - f(\mathbf{x}; \theta^{(k)})) x_{ij} & j = 1, \dots, d \\ \theta_0^{(k+1)} &= \theta_0^{(k)} + \frac{\eta}{s} \sum_{\mathbf{x} \in X_i} (t - f(\mathbf{x}; \theta^{(k)}))\end{aligned}$$

Ogni mini-batch fornisce quindi un aggiornamento “bilanciato” dei parametri.

Effetti pratici

- Gli aggiornamenti sono **più stabili** rispetto a quelli di SGD, perché il gradiente medio sul mini-batch riduce la varianza.
- Il metodo è **più veloce** di BGD, poiché non richiede l'intero dataset per ogni iterazione.

La scelta della dimensione s influisce sul comportamento:

- $s = 1 \rightarrow$ SGD (massimo rumore, ma veloce)
- $s = n \rightarrow$ BGD (stabile, ma lento)
- $1 < s < n \rightarrow$ Mini-Batch GD (compromesso ideale)



Nella pratica

Il *Mini-Batch Gradient Descent* è l'algoritmo più usato nel **Deep Learning**, poiché consente di sfruttare:

- l'efficienza del calcolo vettoriale (GPU),
- la stabilità del gradiente medio,
- la scalabilità su dataset di grandi dimensioni.

Tipicamente, la dimensione del mini-batch è compresa tra **32 e 256**... ma dipende da metodo usato, modello e dataset!

```
def mb_gd(X, t, eta=0.01, epochs=1000, minibatch_size=10):
    """
        Implementa il Mini-Batch Gradient Descent (MBGD) per la regressione
        logistica.

        A differenza del Batch GD (che usa tutto il dataset) e dello
        Stochastic GD
        (che usa un solo esempio per volta), qui il gradiente viene calcolato
        su
        piccoli gruppi di esempi ("mini-batch") di dimensione prefissata.

        Monitoraggio:
        - Anche qui, per tracciare l'andamento della funzione di costo, la
        'loss'
        viene calcolata sull'INTERO dataset (X, t) e salvata in
        cost_history.
        - ⚠ Attenzione: il costo globale NON viene usato per l'aggiornamento,
        serve solo a monitorare la convergenza.

        Parametri:
        -----
        X : ndarray (n x d)
            Matrice dei dati (con colonna di 1 per il bias).
        t : ndarray (n x 1)
            Vettore delle etichette (0 o 1).
        eta : float
            Learning rate – ampiezza del passo di aggiornamento.
        epochs : int
            Numero di epoche (quante volte il dataset viene interamente
            attraversato).
        minibatch_size : int
            Numero di campioni usati per ogni aggiornamento dei parametri.

        Ritorna:
```

```

-----
cost_history : ndarray
    Valori della funzione di costo durante l'addestramento.
theta_history : ndarray
    Valori dei parametri  $\theta$  (bias + pesi) a ogni aggiornamento.
m, q : ndarray
    Pendenza e intercetta della retta di separazione nel piano (x1,
x2).
"""

# 1 Inizializzazione
n = len(X)                                # numero di esempi
theta = np.zeros(X.shape[1]).reshape(-1,1) # parametri (bias + pesi)
theta_history, cost_history = [], []      # per tracciare evoluzione

# 2 Ciclo sulle epoche
for _ in range(epochs):

    # Mescoliamo i dati a ogni epoca (per evitare effetti dovuti
all'ordine)
    perm = np.random.permutation(n)
    X_shuf, t_shuf = X[perm], t[perm]

    # 3 Ciclo sui mini-batch
    for start in range(0, n, minibatch_size):
        # Estrazione del mini-batch corrente
        Xb = X_shuf[start:start+minibatch_size]
        tb = t_shuf[start:start+minibatch_size]

        # Calcolo del gradiente sul mini-batch e aggiornamento
parametri
        theta -= eta * gradient(theta, Xb, tb)

        # Salviamo i valori correnti
        theta_history.append(theta.copy())
        cost_history.append(cost(theta, X, t)) # costo sempre su
tutto il dataset

# 4 Conversione in array NumPy
theta_history = np.array(theta_history).reshape(-1, X.shape[1])
cost_history = np.array(cost_history).reshape(-1, 1)

# 5 Calcolo della pendenza m e intercetta q della retta di

```

```
separazione
```

```
    denom = np.where(np.abs(theta_history[:,2]) < 1e-12, np.nan,  
theta_history[:,2])  
    m = -theta_history[:,1] / denom  
    q = -theta_history[:,0] / denom  
  
    return cost_history, theta_history, m, q
```

Considerazioni sul Mini-Batch Gradient Descent

◆ Comportamento osservato

Il Mini-Batch Gradient Descent mostra un andamento **oscillante ma regolare**:

- gli aggiornamenti sono meno rumorosi rispetto allo SGD,
- ma più veloci e leggeri rispetto al Batch GD.

La funzione di costo decresce progressivamente,
pur con leggere fluttuazioni dovute alla natura stocastica dei mini-batch.

◆ Vantaggi

- **Equilibrio** tra velocità e stabilità.
 - **Riduzione della varianza** del gradiente.
 - **Scalabile**: permette addestramento efficiente anche su dataset molto grandi.
 - **Ottimizzato per GPU**: i mini-batch consentono calcoli paralleli molto efficienti.
-

⚠ Limiti

- La convergenza dipende dalla scelta della dimensione del batch:
 - batch troppo piccolo → rumore elevato;
 - batch troppo grande → aggiornamenti lenti.
 - Come per gli altri metodi, la scelta del *learning rate* rimane cruciale.
-

💡 In sintesi

Aspetto	Batch GD	Stochastic GD	Mini-Batch GD
Aggiornamento	Tutto il dataset	1 campione	s campioni
Stabilità	Alta	Bassa	Media
Velocità	Lenta	Alta	Alta
Varianza del gradiente	Bassa	Alta	Media
Scalabilità	Bassa	Alta	Alta
Tipico uso	Dataset piccoli	Dataset enormi / online	Deep Learning

Conclusione

Il Mini-Batch Gradient Descent è oggi lo **standard de facto** per l'addestramento di modelli complessi,

poiché combina i vantaggi di entrambi gli approcci:

è **veloce, stabile e facilmente parallelizzabile**.



Confronto dei risultati

A questo punto possiamo confrontare in modo sintetico i tre metodi:

- il **tempo di esecuzione**,
- il **numero di passi** (cioè quante volte vengono aggiornati i parametri),
- il **numero totale di gradienti calcolati**,
- e il **valore finale della funzione di costo**.

Questo confronto ci permette di valutare l'**efficienza computazionale** e la **stabilità della convergenza** di ciascun approccio:

- **Batch GD** → più lento, ma con andamento regolare e stabile.
- **SGD** → molto veloce negli aggiornamenti, ma con fluttuazioni più ampie.
- **Mini-Batch GD** → compromesso efficace tra stabilità e rapidità.



Dulcis in fundo: come vanno questi metodi?

Abbiamo implementato e visualizzato tre varianti della **discesa del gradiente**:

- Batch GD
- Stochastic GD
- Mini-Batch GD

Abbiamo osservato tempi di esecuzione, numero di passi e andamento della funzione di costo.

👉 Ora arriva la domanda più importante: **quanto bene predicono?**

Per rispondere, calcoliamo l'**accuratezza finale** dei modelli addestrati.

Lo facciamo utilizzando i parametri θ stimati da ciascun metodo per predire le etichette del nostro dataset e confrontarle con i valori reali.

⚠ **Attenzione:** in questa fase stiamo misurando le performance **sullo stesso dataset usato per l'addestramento**.

- Questo significa che la valutazione può risultare **ottimistica**.
- In pratica dovremmo usare un **validation set** o una **cross-validation** per stimare correttamente le prestazioni su dati nuovi.

Per ora, lo scopo è **confrontare i metodi di ottimizzazione tra loro**: vedremo se, oltre a convergere bene in termini di costo, arrivano anche a buoni risultati di classificazione.

Algoritmi avanzati di ottimizzazione

I metodi di discesa del gradiente visti finora (batch, stochastic e mini-batch) rappresentano la base dell'ottimizzazione in Machine Learning. Tuttavia, nella pratica presentano limiti significativi: lenta convergenza, sensibilità al valore del learning rate e difficoltà in presenza di funzioni di costo complesse e non convesse.

Per affrontare queste criticità, sono stati sviluppati diversi **metodi avanzati di ottimizzazione**, che introducono strategie aggiuntive per migliorare stabilità ed efficienza della discesa:

- **Momento**: sfrutta l'inerzia dei passi precedenti per ridurre le oscillazioni e accelerare la convergenza.
- **Nesterov Accelerated Gradient**: anticipa la direzione di aggiornamento, ottenendo un effetto più predittivo.
- **Adagrad, RMSProp e Adadelta**: adattano dinamicamente il learning rate in base alla storia dei gradienti.
- **Adam**: combina i vantaggi di momentum e adattività, diventando l'ottimizzatore più usato nelle reti neurali moderne.
- **Metodi del secondo ordine**: sfruttano informazioni derivate dalla matrice Hessiana per una discesa più rapida e precisa.

Nelle prossime lezioni analizzeremo questi approcci, a partire dal **metodo del momento**, per capire come affrontano i limiti delle tecniche elementari.