

Machine Learning

Emanuele Galiano
Damiano Trovato

Anno Accademico 2025/2026

Indice

1	Introduzione al Machine Learning	1
1.1	Prime definizioni di Machine Learning	1
1.1.1	Arthur Samuel - 1959	1
1.1.2	Perchè abbiamo bisogno del Machine Learning	1
1.1.3	Esempio delle email spam e non-spam	2
1.1.4	Tom Mitchel - 1998	2
1.2	Definizioni fondamentali	2
1.2.1	Definizione di Task	2
1.2.2	Definizione di Esperienza	2
1.2.3	Definizione di Performance	3
1.2.4	Esempio completo	3
1.3	Task, Esempi ed Etichette	4
1.4	Features	4
1.4.1	Estrazione	4
1.4.2	Definizione formale	5
1.5	Esempio delle Email Spam e Non Spam	5
1.6	Tipologie di Task	6
1.6.1	Classificazione	6
1.6.2	Regressione	7
1.7	Tipi di Machine Learning	8
1.7.1	Supervised Learning e Unsupervised Learning	8
1.7.2	Reinforcement Learning	9
1.8	Misura di Performance (P)	9
1.8.1	Esempio	10
1.9	Elementi fondamentali del Machine Learning	10
1.9.1	Experience (E)	10
1.9.2	Dataset (D)	11
1.9.3	Design Matrix	11
1.9.4	Esempio	11
1.9.5	Learning	12
1.9.6	Il paradigma Training/Testing	13
1.9.7	Normalizzazione	14
1.10	Iperparametri	15

1.10.1	k-fold Cross Validation	15
1.11	Overfitting e underfitting	16
1.11.1	Bias e Variance	16
2	I primi modelli di Machine Learning	17
2.1	Neurone computazionale - modello di McCulloch-Pitt	17
2.1.1	Limitazioni del modello di McCulloch-Pitt	18
2.2	Percettrone - Rosenblatt	18
2.2.1	Processo generale di training del percettrone	20
2.2.2	Implementazione del training	20
2.2.3	Versione di Adaline	21
2.3	Modelli lineari	21
2.3.1	Problema della separabilità non lineare	21
2.3.2	Problema dello XOR	23
3	Regressione	25
3.1	Ingredienti del task	25
3.1.1	Tipi di regressione	25
3.2	Regressione lineare semplice	26
3.2.1	Funzione di loss	26
3.3	Regressione lineare multipla	26
3.4	Feature scaling	26
3.5	Algoritmo di discesa del gradiente	27
3.5.1	Idea dell'algoritmo	27
3.5.2	Algoritmo generale	27
3.6	Regressione polinomiale e feature mapping	28
3.7	Regolarizzazione	28
3.8	Valutazione	28
3.8.1	REC curve	29
4	Classificazione	31
4.1	Introduzione alla classificazione	31
4.1.1	Classificazione: binaria e di classe	31
4.2	Classificazione binaria	31
4.2.1	Perché non usare la regressione lineare per la classificazione?	32
4.3	Regressione logistica - Sigmoid	33
4.3.1	Vantaggi relativi all'uso della sigmoid	34
4.4	Funzione di Loss	34
4.4.1	Derivata parziale della funzione di costo	35
4.4.2	Entropia dell'informazione	35
4.4.3	Binary Cross Entropy Loss	36
4.5	Overfitting nella classificazione	37
4.6	Reject Region, regione d'incertezza	37

4.7	Classificazione multiclasse, approccio naive OVA e OVO	38
4.7.1	Metodo One vs All	38
4.7.2	Metodo One vs One	38
4.7.3	Confronto tra <i>One vs All</i> e <i>One vs One</i>	39
4.8	Stochastic Gradient Descent	39
4.9	Softmax - Classificazione Multiclasse	40
4.9.1	Definizione	40
4.9.2	Funzione Softmax	40
4.9.3	Funzione di Costo	41
4.9.4	Obiettivo di Learning	41
4.9.5	Proprietà del Softmax	41
4.9.6	Rete neurale Softmax	43
4.9.7	Funzione di Loss	43
5	Design, valutazione e scelta dei parametri di un algoritmo di ML	45
5.1	Momento e gradiente	45
5.2	Design di Algoritmi	45
5.2.1	Strategie di miglioramento	46
5.3	Valutazione	46
5.3.1	Valutazione incrociata	47
5.3.2	Bias e Varianza	48
6	Introduzione alle reti neurali	53
7	Decision Tree	55
7.1	Alberi di classificazione	55
7.1.1	Divisione ricorsiva	55
7.1.2	Migliore divisione dei nodi	56
7.2	Alberi di regressione	57
7.2.1	Costruzione dell'albero di regressione	57
7.2.2	Divisione di un nodo	58
7.3	Pruning: riduzione dell'albero	59
7.3.1	Pre-pruning	59
7.3.2	Post-pruning	60
7.4	Regole di learning	60

Capitolo 1

Introduzione al Machine Learning

1.1 Prime definizioni di Machine Learning

1.1.1 Arthur Samuel - 1959

Nel 1959, Arthur Samuel fornisce una **definizione di machine learning**: il machine learning è il campo di studi che abilita i computer ad **imparare, senza essere esplicitamente programmati**.

Il paradigma alla base è fortemente diverso da quello tipico: se un algoritmo classico è **programmato ad hoc** per risolvere un task, e si comporta in maniera prettamente **deterministica**, un algoritmo di machine learning può **imparare a risolvere problemi** associati a vasti set di dati (classificare elementi, riconoscerli, trovare correlazioni). Questo permette di spostare il nostro focus non più sullo sviluppo di tutti gli step necessari affinchè un algoritmo risolvi **un problema specifico**, ma sulla creazione di un modello che riesca ad apprendere come risolvere **un insieme di problemi simili**.

1.1.2 Perchè abbiamo bisogno del Machine Learning

L'approccio utilizzato finora per risolvere i problemi è quello di:

- Trovare una logica per risolvere il problema.
- Scrivere un programma.
- Suddividerlo in pezzi più piccoli (funzioni).
- Automatizzare l'approccio.

Questo funziona per problemi di natura fortemente univoca, che sappiamo come risolvere, ad esempio:

- Computare l'area di un poligono.
- Risolvere equazioni differenziali.

Nel caso del poligono, supponendo di voler calcolare l'area di un rombo i dati presi in input sarebbero dati dalla coppia (x_1, x_2) , contenente le lunghezze della diagonale principale

e secondaria. Questi dati, passati ad un algoritmo, permettono di calcolarne l'area $\frac{x_1 \cdot x_2}{2}$ e generarne un output

Dati → Programma che risolve un task → Output

Alcuni problemi tuttavia presentano un alto grado di **incertezza**, che li rende più difficili da affrontare. Non poter fare assunzioni sui dati in input, e non conoscere tutti i possibili task, rende impossibile l'utilizzo di algoritmi standard per compiti del tipo:

- Classificazione di email spam e non spam
- Object detection

Il machine learning rappresenta la soluzione ideale a problemi di questo tipo, proponendo una nuova pipeline:

Dati + Output Atteso → Machine Learning → Soluzioni su nuovi dati

1.1.3 Esempio delle email spam e non-spam

Vogliamo creare un algoritmo di machine learning in grado di determinare se una mail è spam o meno. Il nostro obiettivo è quindi classificare ciascuna di queste come **spam**, o **ham**¹.

- Compra prodotto a 10\$! Oferta imperdibile! → Spam
- Ciao Giovanni, come stai? → Ham

1.1.4 Tom Mitchel - 1998

Un algoritmo **apprende dall'esperienza** E rispetto a una certa classe di **Task** T e a una misura di **performance** P . Se la sua **performance** nel compito T , misurata tramite P , migliora con l'**esperienza** E , allora quel modello ha appreso con successo.

1.2 Definizioni fondamentali

1.2.1 Definizione di Task

Rappresenta il problema che deve essere risolto. Nell'esempio di determinare se una mail è spam o meno, il task è quello di **predire** l'etichetta ($Y = \text{"spam"}$ oppure $Y = \text{"ham"}$), ed è strettamente legata al modello, che rappresentiamo come funzione parametrizzata, indicata con h_θ .

1.2.2 Definizione di Esperienza

Rappresentano i dati, ovvero i valori assunti dalle **random variables**, nell'esempio X è il contenuto della mail ed Y l'etichetta. La coppia di valori:

¹Email legittima, non spam.

$$\{(X = x_i, Y = y_i)\}_{i=1}^N$$

Rappresenta l'esperienza. Generalmente vista come una collezione di elementi chiamati **esempi**.

1.2.3 Definizione di Performance

Funzione P che **valuta quanto bene** il modello è in grado di **risolvere un certo task** T . Supponiamo che il nostro algoritmo abbia previsto un insieme di etichette per un dato numero di email che indichiamo con:

$$\{\hat{y}_i\}$$

Dove il simbolo 'hat' indica che il dato non è stato osservato ma **previsto**. L'insieme delle etichette corrette è invece dato da

$$\{y_i\}$$

Per valutare la qualità del nostro metodo, dovremmo confrontare i due insiemi di previsioni utilizzando una **misura di performance**:

$$P(\{y_i\}, \{\hat{y}_i\})$$

Questa funzione restituisce un valore reale appartenente al range [0,1].

- Un **valore elevato** indica che le previsioni sono accurate
- Un **valore basso** indica che le previsioni non sono accurate.

Indichiamo con il termine **misura di errore** il valore: $1 - P$. Per risolvere problemi di machine learning ci affidiamo a modelli statistici che dipendono dal task.

1.2.4 Esempio completo

Siano:

- $x^{(1)}$: Il testo dell'email 1: "Compra prodotto a 10\$! Oferta imperdibile!"
- $x^{(2)}$: Il testo dell'email 2: "Ciao Giovanni, come stai?"
- $y^{(1)}$: L'etichetta **spam**
- $y^{(2)}$: L'etichetta **ham**
- h_θ : Il modello

Allora

$$h_\theta(x^{(1)}) = \hat{y}^{(1)}$$

e

$$h_{\theta}(x^{(2)}) = \hat{y}^{(2)}$$

1.3 Task, Esempi ed Etichette

Un esempio (o osservazione) è generalmente espresso come una raccolta di valori che sono stati misurati quantitativamente da un evento osservato. Un esempio è generato da un vettore:

$$x \in \mathbb{R}^d$$

Scritto anche come:

$$x = (x_1, x_2, \dots, x_d)$$

I valori del vettore x sono detti **features**, in quanto rappresentano **proprietà specifiche** degli esempi in input. Se la dimensionalità di x è 10, diremo che ha 10 features. Nella maggior parte dei casi, ogni esempio x è anche abbinato a un output desiderato y . Tali output desiderati, sono anche chiamati **etichette**. Un'attività può quindi essere definita come un certo modo di elaborare un esempio di input per ottenere un output.

Torniamo al nostro esempio: determinare se un'e-mail è spam o ham. In questo caso, l'input è l'email, le features possono essere caratteristiche dell'email, come il numero di errori ortografici o la presenza di alcune parole chiave, mentre l'output atteso è l'etichetta (spam o ham).

1.4 Features

1.4.1 Estrazione

Per gestire le email, dobbiamo prima trasformarle in un'**entità quantificabile**. Questo di solito viene fatto **identificando alcune caratteristiche** dei dati che sono **rilevanti per il compito dato** (numero di errori ortografici o la presenza di alcune parole chiave). In pratica, stiamo cercando una funzione f che trasformi l'entità dalla sua forma originale a una forma di destinazione, che è buona per risolvere un compito specifico:

$$x \rightsquigarrow f(x) \rightsquigarrow \bar{x}$$

Dove x è il dato grezzo di input (ad esempio, il messaggio di posta elettronica completo), f è la funzione di trasformazione e \bar{x} ² è l'output della trasformazione, che sarà l'input dell'algoritmo di apprendimento automatico.

La funzione f è chiamata *rappresentazione*. L'output della trasformazione x è anche chiamato rappresentazione. Poiché rappresentando i dati otteniamo un vettore di funzionalità, il processo di rappresentazione dei dati è talvolta chiamato **features extraction**.

²Su questa notazione: da ora in poi, quando ci riferiremo all'output della trasformazione, non useremo più \bar{x} , ma direttamente x , dando per scontato il passaggio di rappresentazione $f(x)$.

Non ci sono «rappresentazioni universali», ma solo rappresentazioni che servono a qualche compito.

Le rappresentazioni sono di 2 tipi:

- Create a mano
- Apprese

L'estrazione delle features **mette in luce caratteristiche salienti** trascurandone altre.

1.4.2 Definizione formale

Generalmente, l'output di una funzione di rappresentazione è nella forma:

$$x = (x_1, x_2, \dots, x_d), \quad x \in \mathbb{R}^d$$

Questo, è composto da un insieme di features . **Una feature è la specifica di un attributo.** Si tratta di una misura che rappresenta **aspetti dei dati** che è utile **evidenziare per risolvere il problema considerato.** Ad esempio, il colore può essere un attributo. "Il colore è blu" è una funzionalità estratta da un esempio.

Le caratteristiche possono essere di due tipi principali:

Categoriche: un numero finito di valori discreti. Questi possono essere:

- **Nominali:** a indicare che non esiste **alcun ordinamento** tra i valori, ad esempio cognomi e colori.
- **Ordinali:** a indicare che esiste un **ordinamento rilevante**, ad esempio in un attributo che assume i valori basso, medio o alto.

Continue: comunemente, **sottoinsieme di numeri reali**, dove c'è una differenza misurabile tra i valori possibili. I numeri interi sono solitamente trattati come continui nei problemi pratici.

1.5 Esempio delle Email Spam e Non Spam

Consideriamo il nostro esempio in cui vogliamo distinguere le e-mail spam da quelle non spam. L'input del processo sono i messaggi di posta elettronica, quindi dobbiamo trasformarli in vettori di features:

$$x = (x_1, x_2, \dots, x_n)$$

con un processo di *features extraction*.

Naturalmente, ci aspettiamo che le funzionalità estratte siano **utili per risolvere il nostro compito** di determinare se un'e-mail è spam o ham. Possiamo notare che le e-mail di spam spesso includono errori ortografici e parole come "Acquista", "occasione" e "10\$". Quindi, potremmo decidere di rappresentare ogni messaggio di posta elettronica con due numeri:

- Il conteggio degli errori ortografici.
- Il numero di volte in cui alcune parole o pattern specifici appaiono nel testo.

Una volta che i messaggi di input sono stati convertiti in **vettori di funzionalità**, possono essere visti come **vettori nello spazio \mathbb{R}^2** .

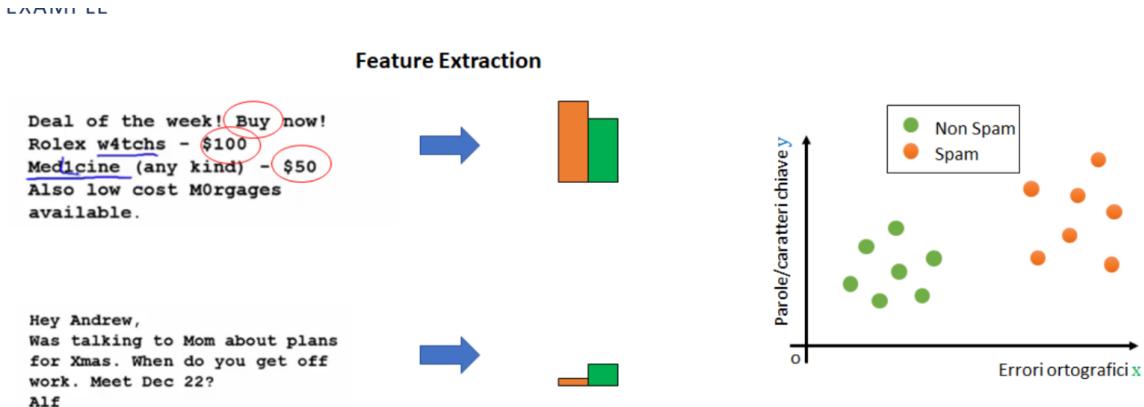


Figura 1.1: Estrazione delle feature: le e-mail vengono trasformate in vettori 2D, dove x = errori ortografici e y = pattern ripetibili, e proiettate nello spazio delle feature, , dove la separazione tra spam (arancione) e non spam (verde) risulta evidente.

1.6 Tipologie di Task

Le attività possono essere di diversi tipi. Di seguito, discuteremo due compiti principali:

- **Classificazione**
- **Regressione**

Assumeremo che ogni algoritmo di apprendimento automatico prenda come input esempi che sono già stati rappresentati con una funzione di rappresentazione adeguata.

1.6.1 Classificazione

In questo tipo di attività, alla macchina viene chiesto di specificare a quale di un insieme predefinito di categorie K appartiene l'input.

Esempi di questo compito sono:

- Classificare i post di Facebook come riguardanti la politica o qualcos'altro (classificazione politica vs non politica).
- Rilevamento delle e-mail di spam (classificazione dello spam vs legittima delle e-mail).
- Riconoscimento dell'oggetto raffigurato in un'immagine tra 1000 oggetti diversi (riconoscimento dell'oggetto).

L'algoritmo di apprendimento è solitamente fornito con un insieme di esempi:

$$\{x^{(1)}, x^{(2)}, \dots, x^{(n)}\} \text{ dove: } x^{(j)} \in \mathbb{R}^N \forall j$$

e un insieme di etichette corrispondenti

$$\{y^{(1)}, y^{(2)}, \dots, y^{(n)}\} \text{ dove: } y^{(j)} \in \{1, \dots, k\} \forall j$$

che specificano a quale delle categorie K appartiene ogni esempio.

Ad esempio, se $y^{(j)} = 3$, allora $x^{(j)}$ appartiene alla classe "3".

Nel caso della classificazione binaria (ad esempio, spam vs non spam), $y^{(j)} \in \{0, 1\}$. Per risolvere questo compito, l'algoritmo di apprendimento automatico assume la forma di una funzione:

$$h_\theta : \mathbb{R}^N \rightarrow \{1, \dots, K\}$$

tale che:

$$y^{(j)} = h_\theta(x^{(j)})$$

Esempio:

- **Classification Task:** data un'e-mail, classificarla come spam o non spam.
- **Input:** esempi n-dimensional $x = (x_1, x_2, \dots, x_n)$ contenenti le caratteristiche dell'email, come il numero di errori ortografici e l'occorrenza di parole specifiche.
- **Output:** etichette $y \in \{0, 1\}$ che indicano se l'e-mail è legittima o spam.

Alcuni algoritmi di classificazione **non prevedono un output discreto**, ma un vettore di **probabilità**, contenente la probabilità relativa a ciascuna delle etichette possibili. In questo caso, puntiamo ad avere la probabilità massima nello slot del vettore relativo all'etichetta vera.

1.6.2 Regressione

In questo tipo di compito, al programma del computer viene chiesto di **prevedere un valore numerico dato un input**, tipo:

- Prevedere il prezzo delle case date alcune caratteristiche come la città, l'età, la zona, ecc.
- Prevedere il valore futuro delle azioni di una società dai valori di altre società o da altre statistiche sul mercato (previsione del mercato azionario).
- Conta il numero di auto presenti in un'immagine.

Analogamente alla classificazione, l'algoritmo viene fornito con esempi di training $x \in \mathbb{R}^N$ e con gli output desiderati $y \in \mathbb{R}$. L'algoritmo di apprendimento automatico assume la forma di una funzione $h_\theta : \mathbb{R}^N \rightarrow \mathbb{R}$ tale che $y^{(j)} = h_\theta(x^{(j)})$.

Esempio:

- **Regression task:** Predire il prezzo di una casa in base ai suoi metri quadrati.
- **Input:** Dimensione della casa x (valore scalare)
- **Output:** Prezzo y .

Sono algoritmi ottimi per trovare relazioni tra i dati ed effettuare predizioni.

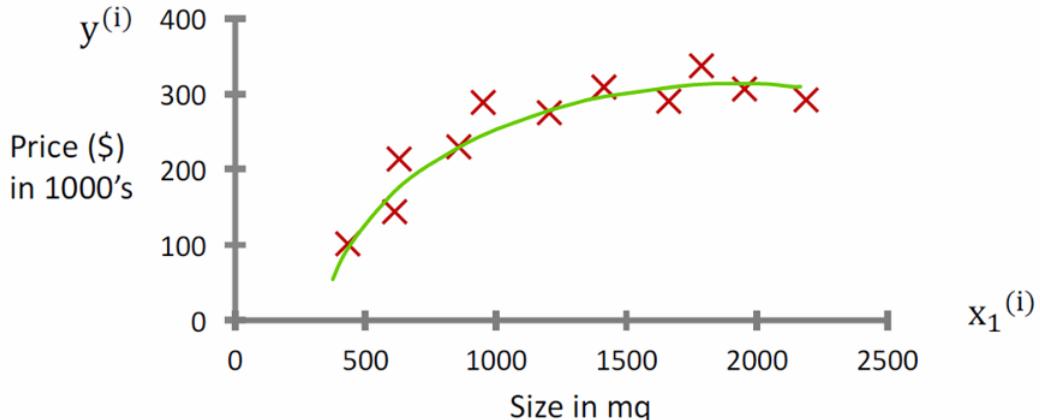


Figura 1.2: Relazione tra dimensione dell’immobile ($x_1^{(i)}$, in mq) e prezzo ($y^{(i)}$, in migliaia di \$): i punti rossi sono i dati osservati, la linea blu rappresenta un modello di regressione lineare che non approssima bene l’andamento non lineare.

1.7 Tipi di Machine Learning

1.7.1 Supervised Learning e Unsupervised Learning

Gli approcci di Machine Learning possono essere approssimativamente divisi in **supervised** e **unsupervised learning**.

Supervised Learning: L’algoritmo viene addestrato su un insieme di esempi di input e **output desiderati**. L’obiettivo è allenare un modello a mappare gli input agli output corretti. Il punto chiave qui è il conoscere gli output desiderati: i dati del nostro dataset dovranno quindi essere preventivamente **etichettati**.

Unsupervised Learning: L’algoritmo viene addestrato solo su esempi di input, senza output desiderati. L’obiettivo è trovare struttura, pattern e associazioni nei dati, spesso molto eterogenei.

$$\{x^{(1)}, x^{(2)}, \dots, x^{(n)}\} \quad \text{dove: } x^{(j)} \in \mathbb{R}^N \forall j$$

Questi tipi di compiti mirano generalmente a **modellare la struttura dei dati**. Un esempio di unsupervised learning è il **clustering**, in cui non viene fornita alcuna informazione aggiuntiva oltre agli esempi.

Gli approcci supervised sono generalmente più facili da gestire, ma richiedono la presenza di **labels**. Ottenerne labels è spesso un problema costoso in termini di tempo, poiché richiede che le persone annotino manualmente i dati. Ad esempio, se dobbiamo costruire un spam-detector utilizzando un approccio supervised, è necessario che qualcuno etichetti manualmente diverse email come ‘spam’ o ‘non-spam’.

1.7.2 Reinforcement Learning

Alcuni autori fanno riferimento anche a una terza classe di algoritmi di Machine Learning: il **Reinforcement Learning**.

Il Reinforcement Learning mira a **scoprire la soluzione a un problema** attraverso il metodo *trial and error*, piuttosto che tramite istruzioni esplicite su come risolvere il compito. Questo avviene permettendo all'algoritmo di **interagire con un environment** e ricevere **positive rewards** quando compie azioni che portano a un buon risultato (rispetto al problema da risolvere) e **negative rewards** quando compie azioni che portano a un risultato negativo.

L'obiettivo degli algoritmi di Reinforcement Learning è apprendere una policy π , che possa essere utilizzata per **determinare quale azione a intraprendere** quando si acquisisce un'**osservazione** del mondo o . Questo processo **ricorda il modo naturale in cui gli animali imparano a risolvere problemi**. Ad esempio, si può pensare a un topo che deve trovare l'uscita da un labirinto (immagine 1.3).

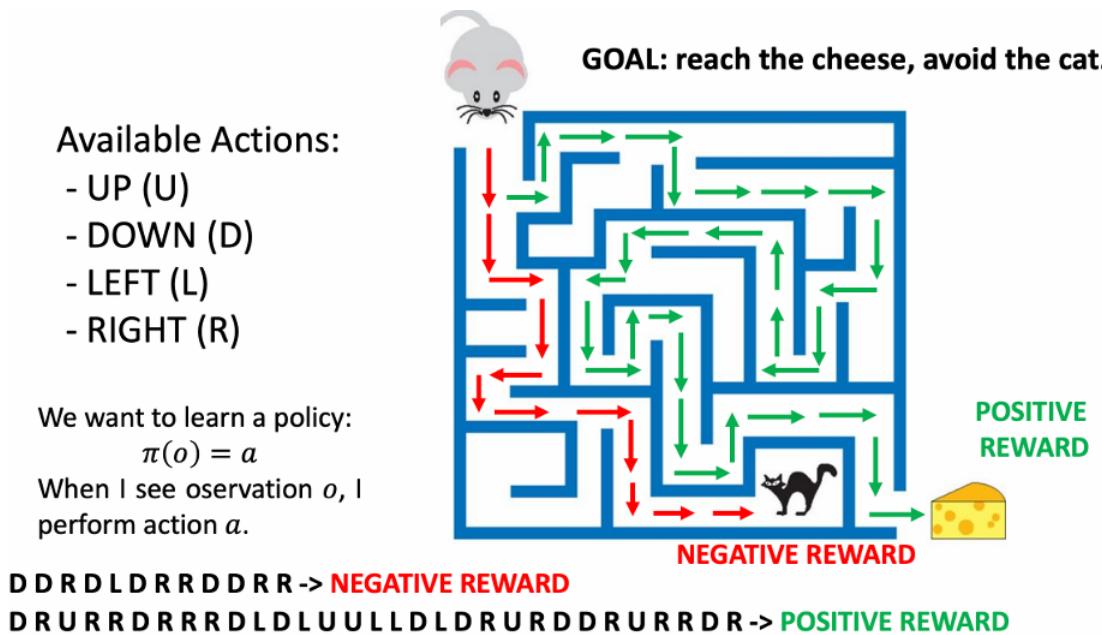


Figura 1.3: Reinforcement Learning nel labirinto: l'agente sceglie tra azioni U/D/L/R e apprende una policy $\pi(o) = a$ che massimizza la ricompensa, raggiungendo il formaggio (positiva) ed evitando il gatto (negativa).

1.8 Misura di Performance (P)

Per valutare le capacità di un algoritmo di Machine Learning nel risolvere un determinato compito, è necessaria una misura quantitativa delle sue prestazioni. Solitamente, questa **performance measure P** è **specificata per il task T** che il sistema sta eseguendo.

Per compiti come la classification, spesso si misura la performance utilizzando l'**accuracy**, ovvero la percentuale di esempi classificati correttamente dal modello. Nel caso della regression, invece, si possono usare altre metriche come il mean squared error.

Le misure di performance sono utilizzate per due motivi principali:

- Capire quando un algoritmo di Machine Learning sta migliorando in un determinato compito.
- Valutare la performance dell'algoritmo una volta finalizzato.

Una performance measure può anche essere vista in termini di error. Ad esempio, l'**accuracy** corrisponde a un error rate (la percentuale di esempi classificati in modo errato), calcolato come $1 - \text{accuracy}$.

1.8.1 Esempio

Un spam detector analizza cinque email. Le prime tre sono spam, le ultime due non lo sono. L'algoritmo classifica come spam le prime due email e come non spam le ultime tre. In questo caso, la prima e le ultime due classificazioni sono corrette, mentre la terza è errata. La accuracy si calcola come la percentuale di esempi classificati correttamente:

$$\frac{4}{5} = 0.8 \quad \text{ovvero} \quad 80\%$$

1.9 Elementi fondamentali del Machine Learning

1.9.1 Experience (E)

Un algoritmo di Machine Learning apprende dall'**experience** per migliorare una performance measure su un determinato task.

L'**experience** è costituita da una raccolta di esempi

$$x^{(i)}$$

(noti anche come data points, poiché possono essere mappati in uno spazio multi-dimensionale tramite una funzione di rappresentazione), eventualmente accompagnati dalle relative labels

$$y^{(i)}$$

(a seconda del task considerato).

Esistono due principali tipi di algoritmi di Machine Learning:

- Supervised approaches (quando abbiamo le paired labels, ad esempio nella classification e nella **regression**).
- Unsupervised approaches (quando non abbiamo paired labels, come nel **clustering**).

L'**experience** assume forme diverse a seconda del tipo di approccio di Machine Learning utilizzato.

1.9.2 Dataset (D)

Le performance measures vengono generalmente calcolate rispetto a un insieme di esempi, piuttosto che su singoli esempi. Un insieme di esempi (eventualmente con labels) è chiamato **dataset**. I datasets sono generalmente omogenei, nel senso che i dati contenuti al loro interno hanno un formato simile. Ad esempio:

- Nel Fisher's Iris dataset, tutti gli esempi hanno 4 features e una label corrispondente a una delle tre classi.
- In un dataset di immagini di food, ogni immagine è associata a una class che indica il piatto specifico.

1.9.3 Design Matrix

Un modo comune per rappresentare un dataset è utilizzare una design matrix. Poiché ogni esempio è una collezione di n features, un dataset di m elementi può essere rappresentato tramite una matrice

$$X \in \mathbb{R}^{m \times n}, m, n \in \mathbb{N}$$

di dimensione $m \times n$.

- Ogni riga della design matrix rappresenta un esempio.
- Ogni colonna rappresenta una delle features.

Nel caso del supervised learning, si considera spesso anche un'altra matrice

$$Y \in \mathbb{A}^{m \times k}, k \in \mathbb{N}$$

dove k è la dimensionalità degli output desiderati.

Ad esempio, nel caso della classification,

$$\mathbb{A} = \{1, \dots, M\}$$

dove M è il numero di classi e k è spesso uguale a 1.

1.9.4 Esempio

Supponiamo di avere un dataset composto da 1000 email, alcune classificate come spam e altre come not spam. Assumiamo che ogni email sia rappresentata da due features, come discusso nei precedenti esempi.

La design matrix che rappresenta il dataset è una matrice

$$X \in \mathbb{R}^{1000 \times 2}$$

- Ogni elemento della matrice rappresenta una delle features di un esempio nel dataset.

j^{th} feature											
$X =$ i^{th} example { }	2.2	4.7	3.8	9.2	4.7	3.2	-1.2	8.9	-0.11	4.12	2
	-0.11	-1.2	-0.11	-1.2	-0.11	4.7	-1.2	4.7	-0.11	9.2	3
	9.2	-0.11	9.2	4.12	9.2	9.2	-1.2	6.9	-1.2	9.2	8
	9.2	9.2	-0.11	-1.2	4.12	-8.6	9.2	-0.11	4.7	9.2	2
	-0.15	9.2	-1.2	-0.11	4.7	-0.11	-1.2	8.7	9.2	-87.5	1
	-0.11	-1.2	4.7	4.7	9.2	4.7	4.12	9.2	9.2	-1.2	0
	9.2	-0.11	9.2	9.2	-1.2	4.7	4.7	-0.11	-1.2	-0.11	8
	9.2	-0.11	4.12	-1.2	32.5	-1.2	9.2	9.8	4.12	9.2	1

Figura 1.4: Design Matrix: rappresentazione di un dataset con m esempi e n features. Ogni riga corrisponde a un esempio, ogni colonna a una feature.

- Ad esempio, $X_{i,1}$ indica il numero di **errori** ortografici nell' **i -esima email**, mentre $X_{j,2}$ rappresenta il numero di **occorrenze** di parole chiave nell' **j -esima email**, e così via.

Le labels sono contenute in un vettore

$$Y \in \{0, 1\}^{1000}$$

dove Y_i rappresenta la label associata all' **i -esimo esempio** (ad esempio, 0 = not spam, 1 = spam).

1.9.5 Learning

Un algoritmo di Machine Learning **utilizza un dataset di esempi per migliorare la sua performance** in un determinato **task**. Il processo di miglioramento della performance dell'algoritmo è chiamato **learning** o **training**.

In cosa consiste il training?

- Un algoritmo di Machine Learning ha alcuni parametri chiamati parameters, che possono essere regolati per modificarne il comportamento. Questi parametri sono legati a un model (una funzione matematica) utilizzata per risolvere il task.
- Un algoritmo chiamato training procedure utilizza gli esempi forniti per trovare i valori ottimali per questi parameters.

- Alcuni parametri non possono essere regolati automaticamente dal training. Questi sono detti hyperparameters e devono essere ottimizzati al di fuori della training procedure, spesso attraverso un metodo trial and error.

Esempio. Consideriamo un semplice spam detector che classifica le email come spam o non-spam in base al numero di errori ortografici.

L'algoritmo può essere scritto come segue:

```

1 def classify(x):
2     if x > a:
3         return 1 # Spam
4     else:
5         return 0 # Non-spam

```

Listing 1.1: Soglia sul numero di errori ortografici, chiaramente un approccio naive

L'algoritmo dipende da un singolo parametro a . La domanda è: quale valore dovremmo assegnare ad a ? La *training procedure* permette di trovare un valore ottimo presumibilmente adatto per a . Una semplice training procedure consisterebbe nel provare diversi valori per a e registrare le performance dell'algoritmo per ciascun valore di a . Alla fine, possiamo scegliere il valore di a che massimizza la performance measure P .

1.9.6 Il paradigma Training/Testing

Ciascun algoritmo di Machine Learning presenta al suo interno dei parametri. Una scelta opportuna dei parametri, dovrebbe garantire il funzionamento del modello. La procedura generale, per la scelta e la verifica dei parametri dell'algoritmo, è suddivisa in due parti:

1. Training dell'algoritmo.

Effettuato usando i dati dell'omonimo **training set**, consiste nella scelta dei parametri del modello, vedendo ogni volta i valori della *loss function*, o funzione di costo, che offre un'indice dell'errore da parte del modello, permettendo di capire se i parametri vanno modificati, e di quanto.

2. Testing dell'algoritmo - Inferenza.

Passiamo ora ai dati di testing, usati per verificare se i parametri scelti sul dataset di training, sono opportuni su dati mai visti prima. Andremo a valutare quindi il nostro modello, secondo quelle che chiamiamo **misure di valutazione**.

Suddivisione dei dati

Come accennato in precedenza, distinguiamo due insiemi di dati: i **dati di training** e i **dati di testing**. È importante che i dati usati in fase di training, e i dati usati in fase di inferenza, siano **totalmente disgiunti**.

$$\text{Training} \cap \text{Testing} = \emptyset$$

Tuttavia, entrambi i set di dati appartengono in origine all'insieme X . La suddivisione dei dati in training e testing sarà effettuata randomicamente, in modo da evitare bias di qualsiasi tipo. Dobbiamo inoltre assicurarsi che il numero di dati, per ogni etichetta, sia uguale. Prendere l'80% di un dataset in fase di testing, con due classi (Cane, non-cane), significa prendere l'80% di ciascuna delle due classi. L'obiettivo è **evitare favoritismi** durante la classificazione. Parleremo più avanti di tecniche che stabiliscono come effettuare la suddivisione tra training e testing, sfruttando al meglio i propri dati, come la k -fold cross-validation.

1.9.7 Normalizzazione

I nostri dati $x \in \mathbb{R}^d$ possono avere valori appartenenti a range molto ampi, con range diversi per ciascuna di queste features. Per creare spazi in cui le features hanno più o meno la stessa dimensione, si usano delle strategie di normalizzazione, come di seguito.

Normalizzazione in $[0, 1]$. Per ogni j -esima feature, dobbiamo andare a stabilire il minimo e il massimo del dataset. Dato un dataset di dimensione m :

$$\begin{aligned} x_j^{\min} &= \min\{x_j^{(i)} \mid i = 1, 2, \dots, m\}, \\ x_j^{\max} &= \max\{x_j^{(i)} \mid i = 1, 2, \dots, m\}. \end{aligned}$$

Possiamo quindi normalizzare secondo la seguente formula:

$$\hat{x}_j = \frac{x_j - x_j^{\min}}{x_j^{\max} - x_j^{\min}},$$

e ogni dato sarà $\hat{x}_j \in [0, 1]$.

Standardizzazione in $[-1, 1]$.

$$\hat{x}_j = \frac{x_j - \mu_j}{\sigma_j}$$

con σ_j varianza. Il risultato è un centramento a $(0, 0)$ dei nostri dati. L'origine coinciderà con la nostra media. Ogni dato sarà $\hat{x}_j \in [-1, 1]$.

Quantità statistiche per la j -esima feature

Per completezza, riportiamo le definizioni delle quantità usate sopra:

$$\text{Media: } \mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)},$$

$$\text{Varianza: } \text{Var}(x_j) = \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2,$$

$$\text{Deviazione standard: } \sigma_j = \sqrt{\frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2} = \sqrt{\text{Var}(x_j)}.$$

Bisognerà conservare rispettivamente x_j^{\min}, x_j^{\max} o μ_j, σ_j , per permettere la normalizzazione del testing set, in quanto la normalizzazione è fatta in fase di training.

1.10 Iperparametri

Sono dei parametri che stabiliscono aspetti del nostro modello, e che non sono calcolati in fase di training. In fase di training, questi vengono fissati, e **non influiscono sul training**. Nonostante ciò, gli iperparametri avranno un effetto sulla nostra loss-function. Bisognerà variare gli iperparametri usando un **validation set**, un terzo set di dati sempre ottenuto dal dataset X .

1.10.1 k-fold Cross Validation

È una strategia che permette di ottenere ottimi risultati anche con dataset più piccoli. Dividiamo in due parti il nostro dataset X , ottenendo l'insieme di training e l'insieme di testing. Dividiamo nuovamente l'insieme di **training** in k parti. Supporremo $k = 3$. Utilizzeremo un $\frac{1}{k}$ per la validazione e le $\frac{k-1}{k}$ rimanenti come training. In questo caso, $1/3$ e $2/3$. Effettueremo tre volte training e validazione, variando ogni volta il subset usato per la validazione e quelli usati per il testing. Da ciascuno dei processi di training, emergerà un valore della loss function: sceglierò gli iperparametri associati a valori della loss function più bassi.

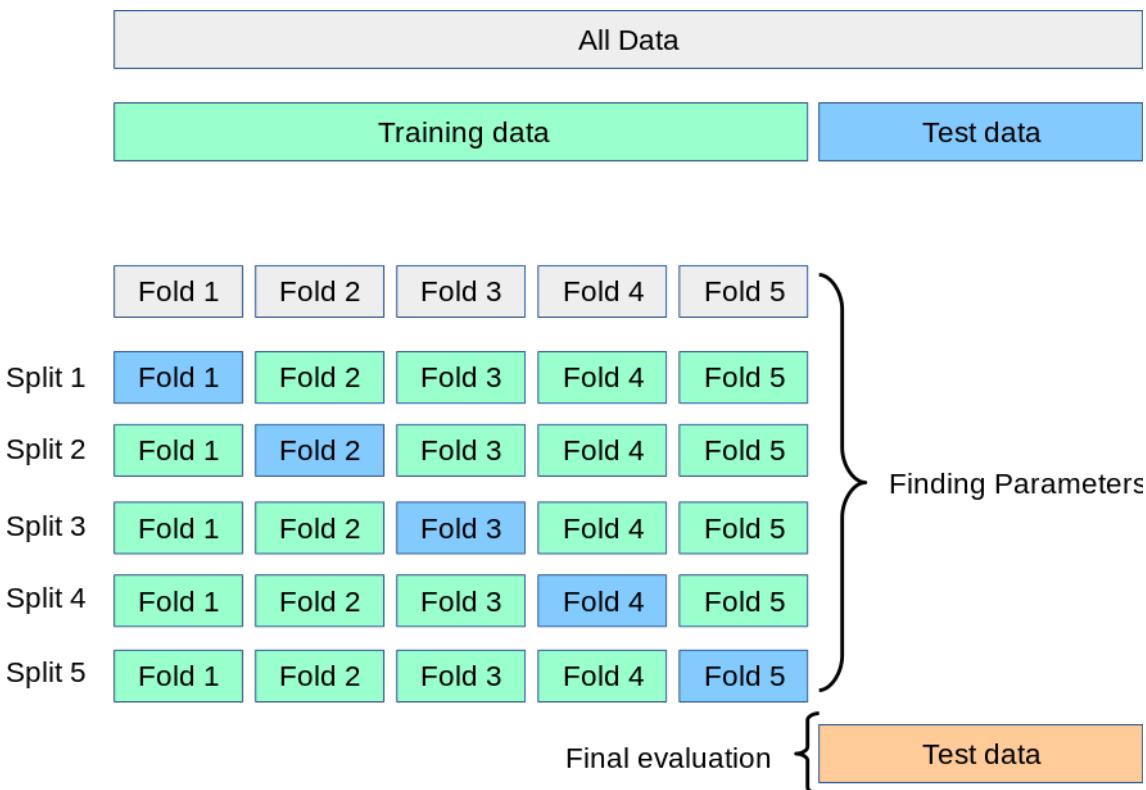


Figura 1.5: Procedura di cross-validation con $k = 5$.

1.11 Overfitting e underfitting

Sono tra i problemi più grandi che possono emergere con i propri algoritmi di machine learning, e riguardano spesso la **scelta del modello**.

Overfitting Generalmente si parla di overfitting, quando il modello è così complesso da offrire performance ottime in fase di training (impara facilmente e bene), ma fortemente **peggiori in fase di testing**: il modello è così complesso da imparare troppo bene i dati su cui ha appreso, ma da pessime performance sui dati inediti.

Underfitting Caratterizzato da **pessime performance in fase di training**, troppe poche variabili, modello troppo semplice. L'apprendimento non è tale da permettere al modello di conoscere il proprio dataset. Il modello non impara!

Bisogna quindi trovare lo *sweetspot*, il compromesso, tra i modelli più complessi, e quelli più semplici. Il numero di parametri è un fattore da tenere sempre in considerazione.

1.11.1 Bias e Variance

Osservando i risultati dei nostri modelli, sarà possibile individuare facilmente due tipologie di errori:

Bias Error. Le previsioni non sono corrette a causa di assunzioni errate nel modello. Un alto bias può portare il modello a non rilevare importanti relazioni tra le features e gli output target, portando a underfitting (es. un modello che cerca di approssimare una funzione polinomiale con una lineare). Usare modelli con alto bias causa underfitting.

Variance Error. Le previsioni del modello sono fortemente influenzate da piccole fluttuazioni nel training set. Ad esempio, se rimuoviamo un punto dati e riaddestriamo l'algoritmo, otteniamo previsioni molto diverse. Questo può essere dovuto al fatto che l'algoritmo sta cercando di modellare il rumore casuale presente nei dati di training, il che di solito porta a overfitting.

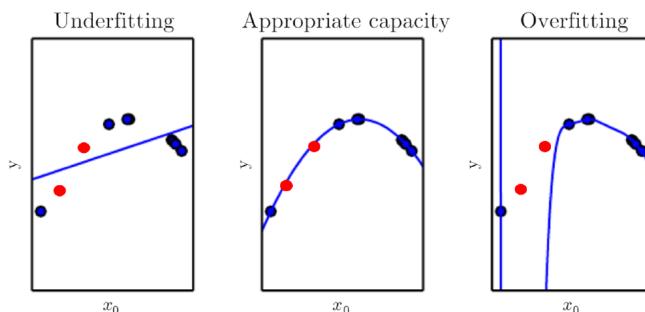


Figura 1.6: In blu i dati di training, in rosso quelli di testing. Nel caso dell'underfitting, vediamo una funzione troppo semplice (una funzione lineare) per approssimare una funzione più complessa. Nel caso dell'overfitting, il modello da output assurdi.

Capitolo 2

I primi modelli di Machine Learning

2.1 Neurone computazionale - modello di McCulloch-Pitt

L'idea iniziale era quella di replicare la struttura di un neurone biologico con un modello matematico, in modo da poter simulare il funzionamento del cervello umano.

Neurone biologico. Un neurone biologico è costituito da:

- Dendriti: ricevono segnali da altri neuroni.
- Corpo cellulare: elabora i segnali ricevuti.
- Assone: trasmette il segnale elaborato ad altri neuroni.

Il primo modello di neurone computazionale fu proposto da McCulloch e Pitts nel 1943. Si basa su tre componenti principali:

- Degli input x_1, x_2, \dots, x_d binari, che possono essere **eccitatori** e **inibitori**.
- Una threshold v , una soglia
- Un output binario.

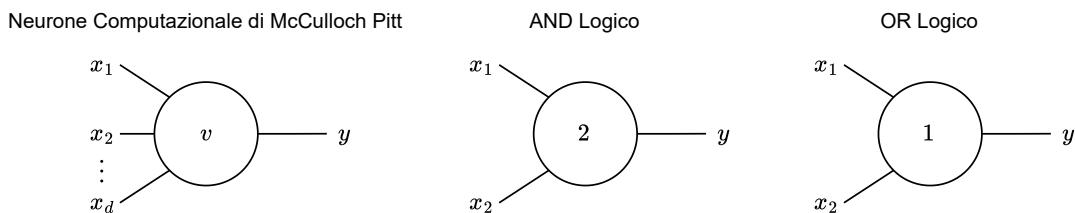


Figura 2.1: Modello di McCulloch Pitt e due computazioni possibili: AND logico e OR logico

Supponiamo che x_1, \dots, x_j siano eccitatori, x_{j+1}, \dots, x_d sono inibitori. Se $j \geq 1$ e almeno un inibitore è = 1, allora il neurone ritorna 0. Un inibitore è sufficiente per bloccare

l'output. Altrimenti, si calcola $z = x_1 + \dots + x_j = x_1 + \dots + x_d$, ossia la somma di tutti gli input¹. Se la somma è $\geq v$, allora l'output sarà 1, altrimenti 0.

Questo modello è tale da poter computare un *AND* ($v = n$ e n input eccitatori), un *OR* ($v = 1$ e n input eccitatori, vedere figura 2.1), ma non uno *XOR*!

2.1.1 Limitazioni del modello di McCulloch-Pitt

- Non esiste un modo automatico di fare training.
- Gli input sono binari.
- Gli input hanno tutti lo stesso peso.
- Tutte le funzioni computabili sono linearmente separabili.

2.2 Percettrone - Rosenblatt

Il successore del modello di McCulloch-Pitt è il **percettrone**. Questo modello risolve il problema dei pesi, assegnando ad ogni ingresso un peso differente, e introduce una procedura di apprendimento automatico per stabilire i pesi in maniera opportuna.

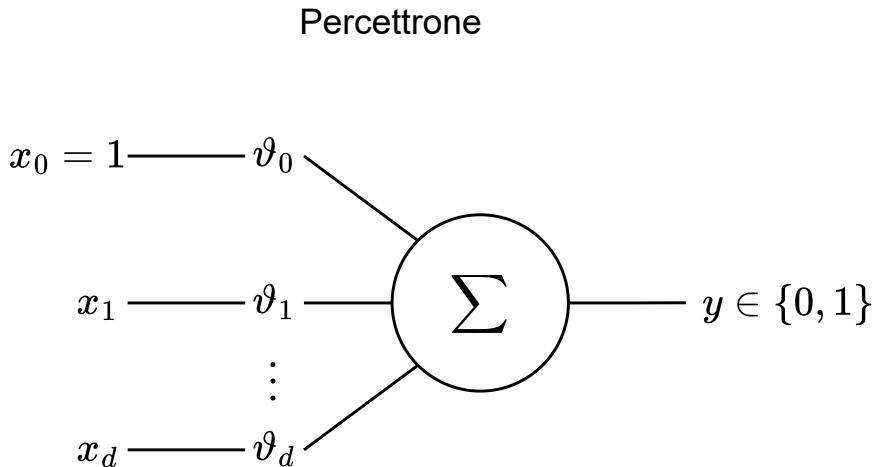


Figura 2.2: Modello di Rosenblatt, noto come Percettrone

Le componenti sono:

- Features x_1, x_2, \dots, x_d normalizzate in $[0, 1]$. Ciascuno di questi input ha associato un peso $\vartheta_1, \dots, \vartheta_d$.
- Una threshold ϑ_0 , una soglia.
- Un output binario.

¹posso considerare anche gli input degli inibitori in quanto = 0.

- Una **procedura di learning automatico** per stabilire i parametri (il peso di ciascun input).

Il comportamento del percettrone sarà il seguente:

$$f(x_1, \dots, x_d) = \begin{cases} 1 & \sum_{j=1}^d x_j v_j \geq \vartheta_0 \\ 0 & \text{altrimenti} \end{cases}$$

Per semplificare la computazione, verrà aggiunta una feature $x_0 = 1$ con peso ϑ_0 , uguale alla threshold. Questo ci aiuterà a scrivere la funzione con la seguente notazione:

$$f(x_1, \dots, x_d) = \begin{cases} 1 & \sum_{j=1}^d x_j v_j \geq 0 \\ 0 & \text{altrimenti} \end{cases}$$

E poi come prodotto matriciale:

$$f(x) = [\vartheta^T \bar{x} > 0]$$

dove ϑ^T è il vettore dei pesi. Supponiamo ora di avere 2 parametri, andremo a ottenere:

$$\vartheta_0 + x_1 \vartheta_1 + x_2 \vartheta_2 \geq 0$$

e con dei semplici passaggi, possiamo tracciare una retta nello spazio, chiamata **decision boundary**, che dividerà lo spazio in due parti, quella per cui $f(x_1, \dots, x_d) = 1$, e quella per cui $f(x_1, \dots, x_d) = 0$

$$x_2 = x_1 \frac{\vartheta_1}{\vartheta_2} + \frac{\vartheta_0}{\vartheta_2}$$

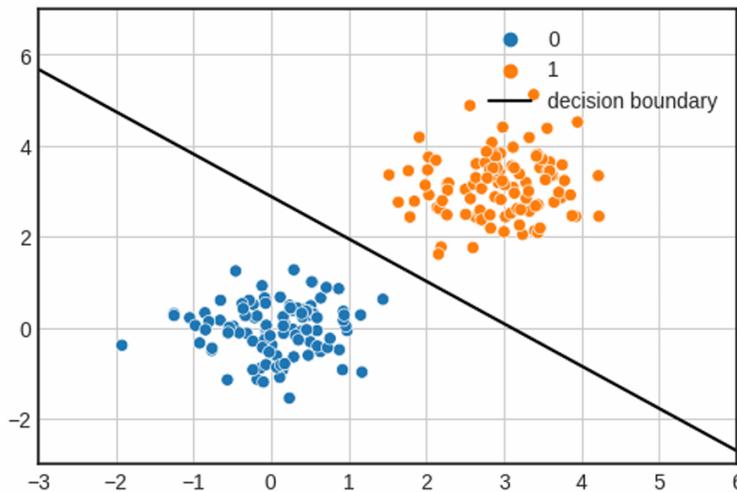


Figura 2.3: Decision boundary del percettrone in 2D. I punti arancioni sono classificati come 1, quelli blu come 0 e la retta al centro è la *decision boundary*.

Questa retta, in un task di classificazione, suddivide lo spazio in due classi. In uno spazio 2D è una retta, in uno spazio 3D un piano. Lo spazio dovrà essere **linearmente separabile**.

2.2.1 Processo generale di training del percettrone

Descriviamo la procedura di training nel seguente modo:

1. Inizializzazione casuale dei pesi $\vartheta_1, \dots, \vartheta_d \in \mathbb{R}$.
2. Computa $\forall x^{(i)}$ il valore di $\hat{y}^{(i)}$.
3. Confronta $\hat{y}^{(i)}$ con $y^{(i)}$
 - Se $\hat{y}^{(i)} = y^{(i)}$: non fare nulla.
 - Se $\hat{y}^{(i)} \neq y^{(i)}$: vanno aggiornati i pesi. Analizziamo come modificare i pesi, in funzione dei risultati.
4. Aggiornamento dei pesi:

$\hat{y}^{(i)}$	$y^{(i)}$	Cosa fare
0	0	ok!
0	1	riduci i pesi.
1	0	aumenta i pesi.
1	1	ok!

Questo sposterà la retta in maniera opportuna, convergendo ad un risultato opportuno per il dataset di training.

Questa descrizione generale riflette a grandi linee il processo, tuttavia è opportuno conoscere la procedura dal punto di vista matematico, che riflette poi l'implementazione effettiva.

2.2.2 Implementazione del training

L'aggiornamento dei pesi avviene nel seguente modo:

$$\underbrace{\hat{\vartheta}_j}_{\text{nuovo peso}} \leftarrow \vartheta_j + (y^{(i)} - \hat{y}^{(i)})x_j$$

Chiaramente, il percettrone non dovrà modificare i suoi pesi se $\hat{y}^{(i)} = y^{(i)}$. Questa formula lo contempla, in quanto $y^{(i)} - \hat{y}^{(i)} = 0$: il peso non verrà aggiornato. Si può aggiungere un parametro α , detto **learning rate**. Il percettrone standard non lo prevede.

$$\hat{\vartheta}_j \leftarrow \vartheta_j + \alpha(y^{(i)} - \hat{y}^{(i)})x_j$$

Scelto in maniera opportuna, potrebbe migliorare l'apprendimento e la convergenza.

2.2.3 Versione di Adaline

Una versione successiva dell'algoritmo di ricalcolo dei pesi, per stabilire con maggiore precisione di quanto incrementare o decrementare i pesi, stabilisce che

$$\hat{\vartheta}_j \leftarrow \vartheta_j + \left(y^{(i)} - \sum_i \vartheta_i \cdot x_i \right) x_j$$

Osserviamo che, per ogni esempio i -esimo, l'uscita desiderata $y^{(i)}$ può assumere solo due valori, 0 oppure 1. Allo stesso tempo, la somma pesata

$$\sum_i \vartheta_i \cdot x_i$$

rappresenta una combinazione lineare normalizzata degli ingressi, e pertanto assume valori compresi nell'intervallo $[0, 1]$. Ne consegue che la differenza tra il valore target $y^{(i)}$ e la somma pesata non potrà che appartenere all'intervallo $[-1, 1]$. In altre parole, se il neurone commette l'errore massimo possibile, questo sarà pari a 1 in valore assoluto, cioè la distanza più grande tra un'uscita binaria (0 o 1) e un valore previsto all'interno di $[0, 1]$.

2.3 Modelli lineari

Il percettrone funziona correttamente solo se i dati sono *linearmente separabili*, ovvero se esiste una frontiera lineare che separa le due classi. Ad esempio, il problema AND è linearmente separabile.

2.3.1 Problema della separabilità non lineare

Questa parte è un approfondimento teorico sulla separabilità lineare. Può essere saltata senza problemi.

Definizione (separabilità lineare). Sia $X = \{x^{(1)}, \dots, x^{(m)}\} \subset \mathbb{R}^d$ e sia $y \in \{0, 1\}^m$ un insieme di etichette. Diciamo che i due insiemi di punti $A = \{x^{(i)} : y^{(i)} = 1\}$ e $B = \{x^{(i)} : y^{(i)} = 0\}$ sono *linearmente separabili* se esistono $w \in \mathbb{R}^d$ e $b \in \mathbb{R}$ tali che

$$w^\top x > b \quad \forall x \in A, \quad w^\top x < b \quad \forall x \in B.$$

Equivalentemente: esiste un iperpiano $w^\top x = b$ la cui parte positiva contiene tutti i punti di A e la parte negativa tutti i punti di B .

Quante funzioni booleane sono linearmente separabili? Per m argomenti binari esistono 2^{2^m} funzioni booleane possibili; solo una parte è realizzabile con un singolo

classificatore lineare. Dati noti:

$$\begin{aligned} m = 2 &\Rightarrow 14 \text{ su } 16 \text{ sono separabili linearmente,} \\ m = 3 &\Rightarrow 104 \text{ su } 256 \text{ sono separabili linearmente,} \\ m = 4 &\Rightarrow 1882 \text{ su } 65536 \text{ sono separabili linearmente.} \end{aligned}$$

Non esiste una formula chiusa semplice che dia, in funzione di m , quante funzioni sono linearmente separabili; tuttavia si osserva che, all'aumentare di m , la *frazione* di funzioni separabili decresce rapidamente.

Dicotomie di un insieme di punti. Dato $X = \{x^{(1)}, \dots, x^{(m)}\} \subset \mathbb{R}^d$, le possibili etichettature (o *dicotomie*) sono 2^m :

$$\mathcal{Y} = \{0, 1\}^m.$$

Solo una parte di queste dicotomie è realizzabile mediante una frontiera lineare. Con d fissata, il numero di dicotomie realizzabili cresce in modo polinomiale in m (ordine al più m^d), mentre il totale cresce esponenzialmente (2^m); di conseguenza,

$$\Pr\{\text{dicotomia separabile linearmente}\} \rightarrow 0 \quad \text{quando } m \gg d.$$

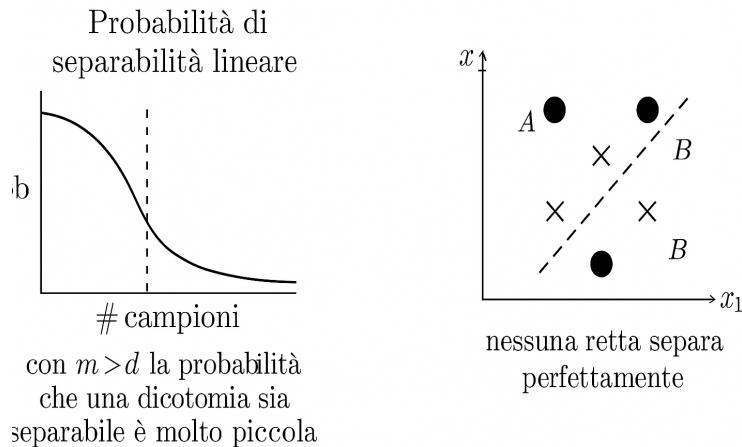


Figura 2.4: Separabilità non lineare. A sinistra: andamento qualitativo della probabilità che una dicotomia sia linearmente separabile al crescere del numero di campioni m (con dimensione d fissata); a destra: esempio in \mathbb{R}^2 di punti non separabili con un singolo iperpiano.

Intuitivamente: se il numero di campioni cresce molto a parità di dimensione d , è sempre meno probabile che una separazione perfetta con una sola retta/iperpiano esista.

2.3.2 Problema dello XOR

Supponiamo però di voler computare lo XOR logico. Per farlo, costruiamo la tabella:

x_1	x_2	$XOR(x_1, x_2)$
0	0	0
0	1	1
1	0	1
1	1	0

Il problema XOR non è linearmente separabile (immagine 2.5): i quattro punti $(0, 0), (1, 0), (0, 1), (1, 1)$ con etichetta di uscita 0, 1, 1, 0 non possono essere separati da una singola frontiera lineare. Un percepitrone (che produce solo confini di decisione lineari) non riesce quindi a risolverlo.

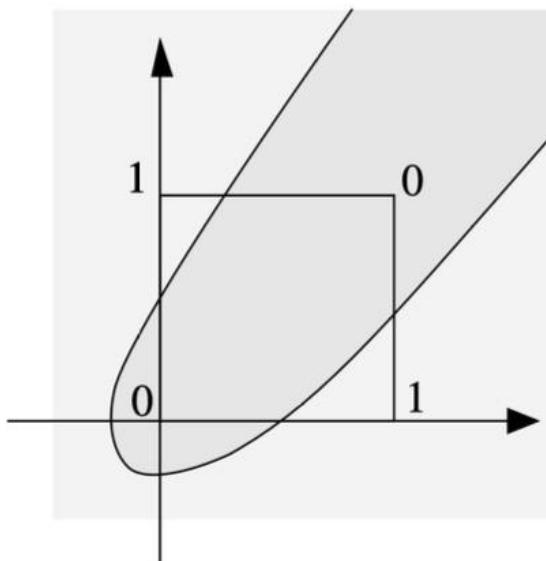


Figura 2.5: Schema del problema XOR: i vertici $(0, 1)$ e $(1, 0)$ appartengono alla classe 1, $(0, 0)$ e $(1, 1)$ alla classe 0. Nessuna frontiera lineare può separare le classi; è necessaria una frontiera non lineare (ottenibile con uno strato nascosto).

Dimostrazione della non-linearità dello XOR. Assumendo la tabella di verità dello XOR e un percepitrone con pesi v_1, v_2 e soglia v_0 , le condizioni di attivazione diventano

$$\begin{aligned} 1 \cdot v_1 + 0 \cdot v_2 &> v_0, \\ 1 \cdot v_1 + 1 \cdot v_2 &< v_0, \\ 0 \cdot v_1 + 1 \cdot v_2 &> v_0, \\ 0 \cdot v_1 + 0 \cdot v_2 &< v_0. \end{aligned}$$

Da cui si ottiene

$$2v_0 < v_1 + v_2 < v_0,$$

ovvero una contraddizione: non esiste configurazione dei parametri di un singolo percettrone che risolva XOR.

Rete di percetroni. La soluzione consiste nell'usare una *rete di percetroni* (uno strato nascosto) che trasformi lo spazio in modo da rendere il problema linearmente separabile all'uscita. Un esempio minimale è:

$$\begin{aligned}f_1(x_1, x_2) &= [x_1 - x_2 \geq 0.5], \\f_2(x_1, x_2) &= [x_2 - x_1 \geq 0.5], \\f_3(f_1, f_2) &= [1 \cdot f_1 + 1 \cdot f_2 \geq 0.5],\end{aligned}$$

dove $[\cdot]$ indica una funzione soglia (vale 1 se la condizione è vera, 0 altrimenti). Le funzioni f_1 e f_2 realizzano una trasformazione non lineare degli input; nello spazio così trasformato l'uscita f_3 è ottenuta tramite una semplice soglia lineare.

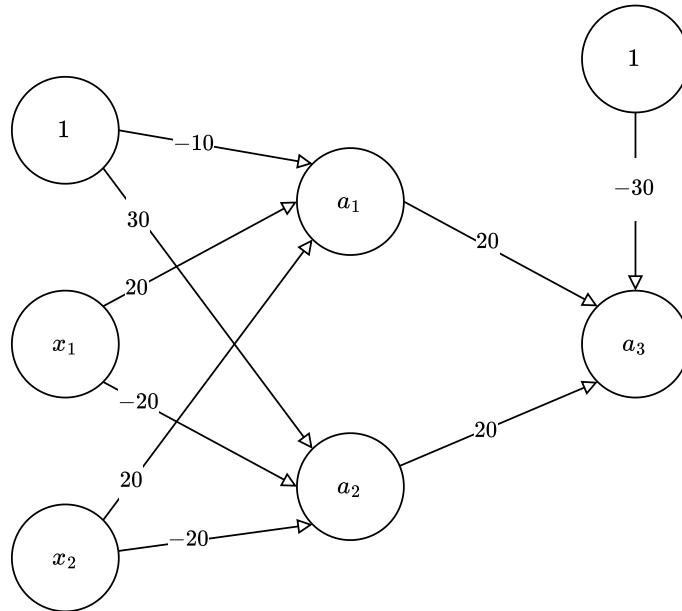


Figura 2.6: Rete di percetroni per computare lo *XOR*

Infine, sebbene reti di percetroni possano risolvere *XOR*, l'algoritmo di apprendimento del singolo percettrone non è direttamente applicabile a reti multistrato; storicamente ciò contribuì al primo *AI winter* e la situazione migliorò con la formalizzazione della *retropropagazione* (*backpropagation*).

Capitolo 3

Regressione

Definiamo il compito di regressione in termini di come un algoritmo elabora un esempio di input e produce un valore (o vettore) reale di output. Formalmente, vogliamo apprendere una funzione

$$h_\vartheta : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}, \quad d_1 \geq 1, d_2 \geq 1,$$

a partire da un dataset supervisionato

$$D = \{(x^{(i)}, y^{(i)})\}_{i=1}^m, \quad x^{(i)} \in \mathbb{R}^{d_1}, y^{(i)} \in \mathbb{R}^{d_2}.$$

In altre parole, individuare una funzione nello spazio dei dati, passante per tutti i valori del dataset in input, e presumibilmente capace di predire il valore associato ai dati di testing. Permette quindi di **stabilire relazioni tra i dati, individuare trend** e fare predizioni.

3.1 Ingredienti del task

- **Task:** predire valori reali a partire da input reali.
- **Modello:** ipotesi parametrica h_ϑ che mappa input in output.
- **Dati:** coppie etichettate $(x^{(i)}, y^{(i)})$.
- **Algoritmo di learning:** metodo per stimare ϑ (ottimizzazione).
- **Funzione di loss:** misura lo scarto tra predizioni e target.
- **Valutazione:** metriche su validation/test per giudicare il modello.

3.1.1 Tipi di regressione

1. **Regressione lineare**, $h_\vartheta : \mathbb{R} \rightarrow \mathbb{R}$, da spazio a una dimensione a una dimensione.
2. **Multiple regression**, $h_\vartheta : \mathbb{R}^{d_1} \rightarrow \mathbb{R}$, $d_1 > 1$, con d_1 features di input e un output.
3. **Multivariate regression**, $h_\vartheta : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}$, $d_1, d_2 > 1$, con d_1 features di input e d_2 output. Un modello di multivariate regression, viene allenato come un modello di multiple regression su ciascuno degli output.

3.2 Regressione lineare semplice

Nel caso $d_1 = d_2 = 1$ modelliamo la relazione tra un singolo ingresso $x \in \mathbb{R}$ e un'uscita reale $y \in \mathbb{R}$ con una retta:

$$h_{\vartheta}(x) = \vartheta_0 + \vartheta_1 x,$$

dove ϑ_0 è l'intercetta e ϑ_1 la pendenza. “Imparare” significa scegliere $\vartheta = (\vartheta_0, \vartheta_1)$ in modo che le predizioni siano vicine ai corrispondenti target.

3.2.1 Funzione di loss

Misuriamo la qualità della retta con l'errore medio quadratico (MSE):

$$J(\vartheta) = \frac{1}{2m} \sum_{i=1}^m (h_{\vartheta}(x^{(i)}) - y^{(i)})^2.$$

Il fattore $1/2$ è una costante di comodo che semplifica le derivate; il quadrato rende positivi tutti i contributi ed enfatizza gli errori grandi. In regressione lineare J è convessa rispetto ai parametri, quindi il minimo è unico (se c'è variabilità negli $x^{(i)}$). Otteniamo un grafico a tazza.

3.3 Regressione lineare multipla

Estendiamo al caso con più variabili in ingresso. Dato il vettore $x = (x_1, \dots, x_n)$, usiamo un parametro per ogni dimensione più il bias:

$$f(x) = \vartheta_0 + \vartheta_1 x_1 + \vartheta_2 x_2 + \dots + \vartheta_n x_n.$$

Per semplicità poniamo $x_0 = 1$ e definiamo il vettore esteso $x = (x_0, x_1, \dots, x_n)^\top$; allora

$$f(x) = \sum_{i=0}^n \vartheta_i x_i = \boldsymbol{\vartheta}^\top x.$$

Questo modello è, in sostanza, un *perceptron senza soglia*: la combinazione lineare $\boldsymbol{\vartheta}^\top x$ è l'uscita continua del regressore.

3.4 Feature scaling

Per far funzionare bene (e in fretta) la discesa del gradiente, le feature vanno portate su scale simili. Due opzioni comuni:

$$\text{z-scoring: } x_j \leftarrow \frac{x_j - \mu_j}{\sigma_j}, \quad \text{min-max: } x_j \leftarrow \frac{x_j - x_j^{\min}}{x_j^{\max} - x_j^{\min}}.$$

Le statistiche $(\mu_j, \sigma_j, x_j^{\min}, x_j^{\max})$ si calcolano solo sul training e si riusano (senza ricalcolarle) su validation/test, per evitare data leakage.

3.5 Algoritmo di discesa del gradiente

L'obiettivo è scegliere ϑ che minimizza $J(\vartheta)$ funzione di loss. La *discesa del gradiente* è un metodo iterativo che aggiorna i parametri nella direzione di massima diminuzione di J . Nel prossimo esempio a due parametri $(\vartheta_0, \vartheta_1)$, supporremo l'intercetta $\vartheta_0 = 0$.

3.5.1 Idea dell'algoritmo

Partiremo da dei valori casuali dei parametri, in questo caso ϑ_1 , in quanto ϑ_0 è già fissata a 0 per semplicità. Osserveremo il coefficiente angolare della retta tangente alla funzione di loss, ossia la sua derivata. Quando questa è:

- negativa, incrementare il valore.
- positiva, decrementare il valore,

Il numero di iterazioni da ripetere dipende dalla convergenza che si vuole ottenere.

3.5.2 Algoritmo generale

Sia $X \in \mathbb{R}^{m \times (n+1)}$ la design matrix con prima colonna di 1, $y \in \mathbb{R}^m$ il vettore dei target e $\hat{y} = X\vartheta$ le predizioni. La loss è

$$J(\vartheta) = \frac{1}{2m} \|X\vartheta - y\|_2^2, \quad \nabla_{\vartheta} J(\vartheta) = \frac{1}{m} X^\top (X\vartheta - y).$$

Pseudocodice.

1. **Inizializza** in modo casuale: $\vartheta^{(0)} = (\vartheta_0, \dots, \vartheta_n)^\top$.
2. **Calcola le derivate parziali:**

$$\frac{\partial J(\vartheta)}{\partial \vartheta_j} = \frac{1}{m} \sum_{i=1}^m (\vartheta^\top \tilde{x}^{(i)} - y^{(i)}) \tilde{x}_j^{(i)}, \quad j = 0, \dots, n,$$

dove $\tilde{x}^{(i)} = (1, x_1^{(i)}, \dots, x_n^{(i)})^\top$.

3. **Aggiorna** i parametri (passo di ampiezza $\alpha > 0$):

$$\vartheta_j \leftarrow \vartheta_j - \alpha \frac{\partial J(\vartheta)}{\partial \vartheta_j}, \quad j = 0, \dots, n.$$

4. **Ripeti** i passi 2–3 finché non è soddisfatto un criterio d'arresto (iterazioni massime, $\|\nabla J(\vartheta)\|$ sotto soglia, variazione di J piccola).

In forma compatta: $\vartheta \leftarrow \vartheta - \frac{\alpha}{m} X^\top (X\vartheta - y)$.

3.6 Regressione polinomiale e feature mapping

Parlando della rete di perceptroni per computare lo *XOR*, abbiamo già anticipato l'utilizzo di trasformazioni per semplificare spazi di partenza non linearmente separabili, col fine ultimo di renderli compatibili a modelli con parametri di tipo lineare. Indichiamo la trasformazione con $\Phi(x) \rightarrow (x')$. Il nostro modello lavorerà su $h_\theta(\Phi(x))$. Prendiamo come esempio la seguente trasformazione, che ci fa passare da uno spazio a 2 dimensioni, ad uno a 5 dimensioni.

$$\phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightsquigarrow \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{pmatrix}$$

Ottenendo un polinomio non lineare rispetto alle features, ma lineare rispetto ai parametri.

$$\vartheta_0 + \vartheta_1 x_1 + \vartheta_2 x_2 + \vartheta_3 x_1^2 + \vartheta_4 x_2^2 + \vartheta_5 x_1 x_2$$

3.7 Regolarizzazione

Per ridurre l'overfitting penalizziamo pesi grandi. In **Ridge** (L2) la loss è

$$J_\lambda(\vartheta) = \frac{1}{2m} \|X\vartheta - y\|_2^2 + \underbrace{\frac{\lambda}{2m} \sum_{j=1}^n \vartheta_j^2}_{\text{Termine di regolarizzazione}},$$

dove tipicamente ϑ_0 non si penalizza. Aggiornamenti:

$$\vartheta_0 \leftarrow \vartheta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}), \quad \vartheta_j \leftarrow \vartheta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) \tilde{x}_j^{(i)} + \frac{\lambda}{m} \vartheta_j \right] \quad (j \geq 1).$$

3.8 Valutazione

Metriche tipiche:

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})^2, \quad \text{RMSE} = \sqrt{\text{MSE}}, \quad \text{MAE} = \frac{1}{m} \sum_{i=1}^m |\hat{y}^{(i)} - y^{(i)}|.$$

R^2 (coefficiente di determinazione):

$$R^2 = 1 - \frac{\sum_i (y^{(i)} - \hat{y}^{(i)})^2}{\sum_i (y^{(i)} - \bar{y})^2}.$$

3.8.1 REC curve

Ordinando gli errori assoluti $e_i = |\hat{y}^{(i)} - y^{(i)}|$ e tracciando, al variare di ε , la frazione cumulativa di esempi con errore $\leq \varepsilon$, si ottiene una curva di facilissima lettura; un'area sotto la curva¹ maggiore indica in genere prestazioni migliori. Analogamente, un'area sopra la curva² maggiore indica maggiore errore atteso. Quanto più velocemente cresce la curva, tanto più il modello dovrebbe essere accurato.

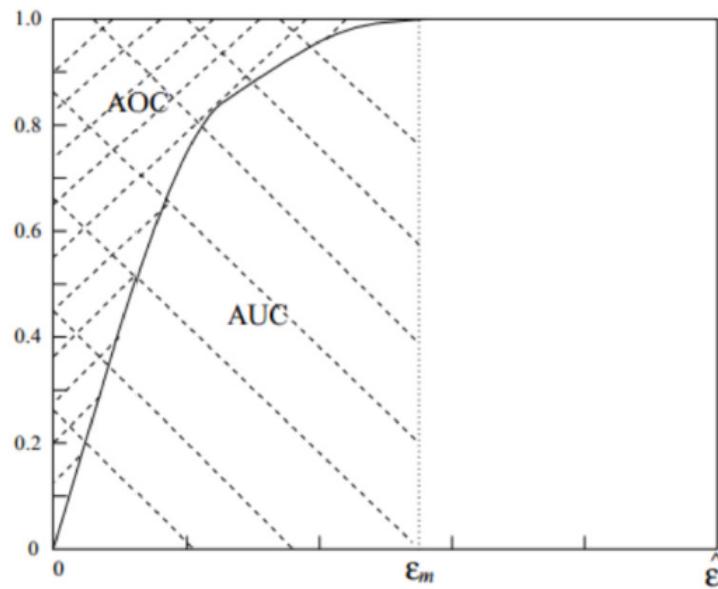


Figura 3.1: Esempio di curva REC.

¹AUC, area under the curve

²AOC, area over the curve

Capitolo 4

Classificazione

4.1 Introduzione alla classificazione

La **classificazione** è un altro dei problemi tipici che può essere risolto tramite un algoritmo di Machine Learning. Riguarda lo **stabilire se un dato input appartiene ad una tra delle classi** prestabilite dal problema. Il **target non sarà più un valore** $\in \mathbb{R}$, ma una tra k classi. Per studiare la classificazione, andremo a stabilire i requisiti e i nostri obiettivi.

- Un modello ben definito.
- Una funzione di loss per stabilire lo scarto con i risultati attesi.
- Un algoritmo di learning per trovare i parametri.
- Delle misure di valutazione.
- Niente overfitting!

4.1.1 Classificazione: binaria e di classe

Identifichiamo due tipi di task di classificazione:

- **Classificazione binaria.**
Le etichette $y \in \{0, 1\}$. Va associata ad x una delle due etichette, o le probabilità relativa a ciascuna delle due.
- **Classificazione di classe.**
Le etichette $y \in \{0, 1, \dots, k-1\}$. Va associata ad x una delle etichette, o la probabilità relativa a ciascuna di esse.

È fondamentale sottolineare l'importanza che gli output di questi algoritmi siano sempre $\in [0, 1]$, in quanto da interpretare come **probabilità**.

4.2 Classificazione binaria

Nonostante i classificatori binari possano sembrare limitati, questi trovano applicazioni in vari task: solitamente, questi si basano sulla suddivisione di immagini più grandi in *patches* più piccole, con una successiva, appunto, classificazione, per individuare determinati

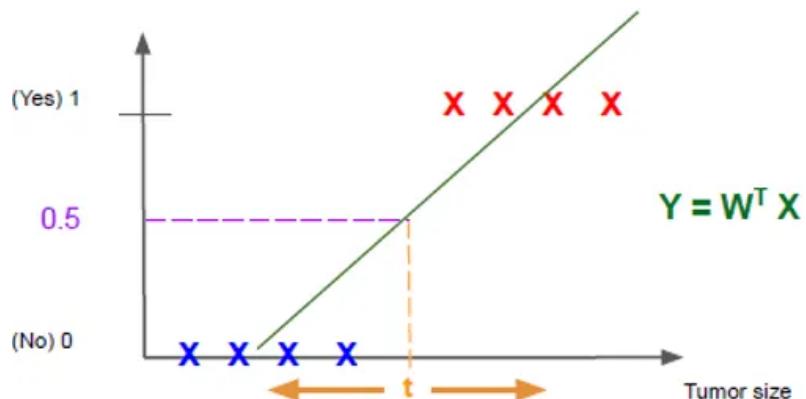
elementi. Nel **medical imaging**, i classificatori binari possono individuare piccoli tumori. Nei **controlli qualità** in ambito industriale, possono individuare **imperfezioni** nei materiali. Nel campo più generale della **computer vision**, gli algoritmi di classificazione (binaria e non) sono fondamentali.

4.2.1 Perché non usare la regressione lineare per la classificazione?

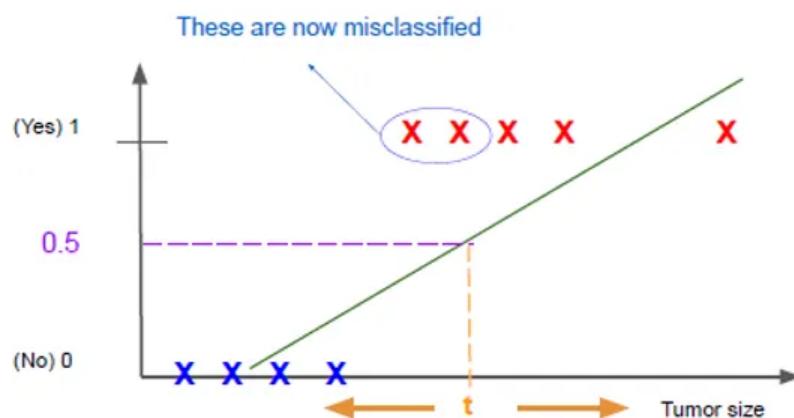
Si potrebbe pensare di **usare un modello di regressione lineare per effettuare classificazione binaria**, e l'intuizione non sarebbe nemmeno totalmente sbagliata: potremmo usare, ad esempio, la regressione per **trovare una retta che unisce i dati** in maniera opportuna, e in funzione della pendenza di questa retta, **stabilire un valore di sogliatura** per distinguere due classi.

Forniamo il seguente esempio: vogliamo addestrare un modello per prevedere se un tumore è benigno o maligno usando una singola feature (dimensione del tumore).

Usiamo la regressione lineare e facciamo il fitting di una retta nello spazio (troviamo la retta, cioè la funzione che meglio si posiziona a media tra tutti i punti). Possiamo trovare il miglior $y = \vartheta^T x$ dal nostro set di dati e quindi selezionare una **soglia** su y per classificare quando il tumore è maligno o meno: $h_{\vartheta}(x) \leq 0.5 \rightarrow 0, \quad h_{\vartheta}(x) \geq 0.5 \rightarrow 1$.



Ci sono **due criticità** relative all'utilizzo della regressione. La prima, riguarda la versatilità del modello: il fitting della retta non è opportuno, in quanto la regressione lineare dipende fortemente dalla distribuzione dei dati.



La seconda criticità riguarda invece la y , non sempre limitata tra 0 e 1, come ci aspettiamo in un modello di classificazione binaria. Come ci comportiamo con valori più grandi di 1? E con valori minori di 0? Possiamo risolvere il problema introducendo delle funzioni, dette funzioni **di linking**. Queste ci permettono di **adattare i nostri valori** a modelli di regressione che altrimenti non sarebbero opportuni per la classificazione. Vediamo un esempio relativo alla regressione logistica usando la funzione **sigmoide**.

4.3 Regressione logistica - Sigmoide

Se la regressione lineare restituisce una y non limitata tra 0 e 1, la **regressione logistica** risolve il problema con una funzione, detta **sigmoide**, applicata sul risultato $z = \vartheta^T X$ del modello di regressione. La **sigmoide** è una funzione del tipo

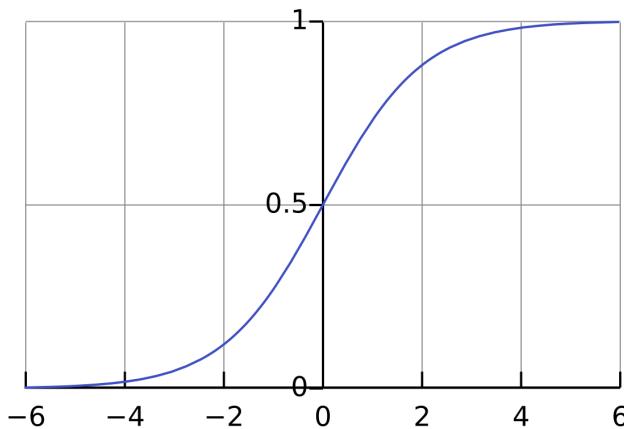
$$\bar{\sigma}(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-\vartheta^T X}}$$

con $z = \vartheta_0 + \vartheta_1 x_1 + \dots + \vartheta_d x_d$. Questa funzione ha un'interpretazione probabilistica molto banale, opportuna per **adattare i modelli di regressione lineare su problemi di classificazione**¹. Osserviamo che:

- $1 - \bar{\sigma} = \sigma(z)$
- $\frac{\partial \bar{\sigma}(z)}{\partial t} = \bar{\sigma}(z)(1 - \bar{\sigma}(z)) = \bar{\sigma}(z)\bar{\sigma}(-z)$
- È inoltre dimostrabile che $\bar{\sigma} = \frac{1}{2} + \frac{1}{2} \tanh\left(\frac{z}{2}\right)$. Questo ci apre le porte a implementazioni molto più semplici.

Concludiamo che la regressione logistica, è ottimale per trovare opportuni parametri di ϑ^T , che moltiplicati a x , ci permetteranno di stabilire il **decision boundary**, nella ben nota forma:

$$\vartheta_0 x_0 + \vartheta_1 x_1 + \vartheta_2 x_2 + \dots + \vartheta_d x_d$$



¹Non si può pretendere di usare un modello di regressione lineare su task di classificazione, senza fare alcun tipo di modifica, come quelle che introduciamo con la sigmoide.

4.3.1 Vantaggi relativi all'uso della sigmoide

- Interpretazione probabilistica semplice. Da valori $\in [0, 1]$.
- Derivabile, più liscia rispetto ad una step function.
- Introduce non linearietà, migliorando la classificazione.

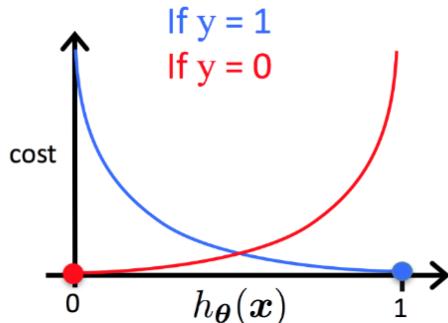
4.4 Funzione di Loss

Avere una funzione di Loss associata al modello è fondamentale per capire come effettuare il training, e quali sono i migliori parametri di ϑ^T per la classificazione. Per modellare questa funzione (da minimizzare secondo l'algoritmo di discesa del gradiente), utilizzeremo due funzioni logaritmiche. La funzione di Loss sarà definita in funzione dell'etichetta \hat{y} .

$$\text{Loss}(h_\vartheta(x), y) = \begin{cases} -\log(h_\vartheta(x)) & \text{se } \hat{y} = 1 \\ -\log(1 - h_\vartheta(x)) & \text{se } \hat{y} = 0 \end{cases}$$

Questo, perché l'errore deve essere 1 quando $\hat{y} = 0$ e $y = 1$, o quando $\hat{y} = 1$ e $y = 0$.

Ai fini della semplicità e univocità della spiegazione, useremo y per indicare ciò che fino ad ora abbiamo indicato con \hat{y} .



Possiamo esprimere la funzione di Loss anche in questo modo, rendendo unica la definizione della funzione.

$$\text{Loss}(h_\vartheta(x), y) = -[y \log(h_\vartheta(x)) + (1 - y) \log(1 - h_\vartheta(x))]$$

Definita la funzione di Loss, andremo a definire la funzione di costo

$$J(\vartheta) = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log(h_\vartheta(x)) + (1 - y^{(i)}) \log(1 - h_\vartheta(x)))$$

su cui sarà possibile applicare l'algoritmo di discesa del gradiente, fino alla convergenza del modello.

$$\vartheta_j^{\text{new}} = \vartheta_j^{\text{old}} - \alpha \frac{\partial J(\vartheta)}{\partial \vartheta_j} \quad \forall j$$

4.4.1 Derivata parziale della funzione di costo

Necessaria per applicare la discesa del gradiente.

$$\frac{\partial J(\vartheta)}{\partial \vartheta_j} = \frac{1}{m} \sum_{i=1}^m (\underbrace{h_\vartheta(x^{(i)}) - y^{(i)}}_{(*)}) x_j^{(i)} \quad \forall j = 0, \dots, d$$

(*) Questo membro nasconde la funzione sigmoide $\bar{\sigma}(\vartheta^T X^{(i)})$.

4.4.2 Entropia dell'informazione

La loss function di questo modello presenta un'interpretazione probabilistica basata sul concetto di **entropia** dell'informazione. Parleremo quindi **Binary Cross Entropy Loss**. La seguente, è la formula dell'entropia:

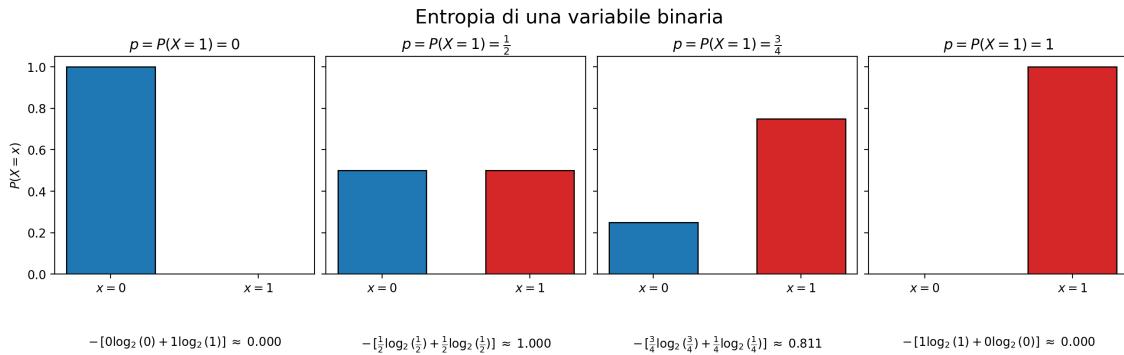
$$H(X) = - \sum_{k=1}^K p_k \log_2(p_k)$$

con p_k probabilità di essere di classe k . La distribuzione di probabilità (e quindi l'entropia). Queste distribuzioni informazioni su:

- Incertezza.
- Misura del disordine delle classi.
- Quanto la probabilità è concentrata su una classe.
- Informazione (la distribuzione è molto informativa se è capace di distinguere in maniera opportuna).

Un esempio su $k = 2$ è la funzione

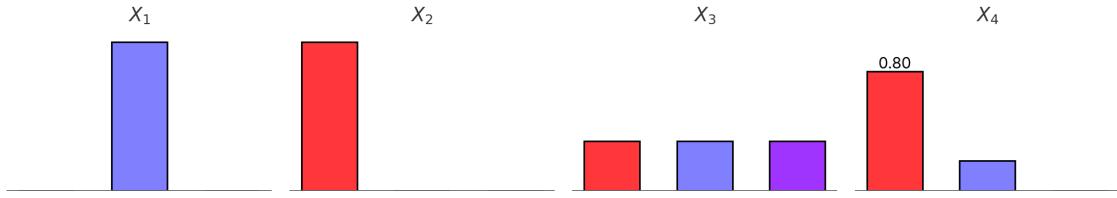
$$H(X) = -p_1 \log_2(p_1) - p_2 \log_2(p_2) = -[p_1 \log_2(p_1) + p_2 \log_2(p_2)]$$



$\log_2(0)$ è *indefinito*. Quando facciamo calcoli relativi alla cross-entropy, consideriamo valori piccoli, mai nulli. In questo esempio e nel prossimo esercizio, quando utilizziamo 0, stiamo approssimando un valore molto piccolo, ma mai effettivamente nullo. Inoltre, in un modello reale, avere una $p_k = 0$ o $p_k = 1$ significa probabile overfitting del modello.

4.4.2.1 Esercizio sull'entropia

Riordina le seguenti distribuzioni in ordine crescente di entropia.



$$H(X) = -p_1 \log_2(p_1) - p_2 \log_2(p_2) = -[p_1 \log_2(p_1) + p_2 \log_2(p_2) + p_3 \log_2(p_3)]$$

1. $H(X_1) = -[0 \log(0) + 1 \log(1) + 0 \log(0)] = 0$
2. $H(X_2) = -[1 \log(1) + 0 \log(0) + 0 \log(0)] = 0$
3. $H(X_3) = -[0,33 \log(0,33) + 0,33 \log(0,33) + 0,33 \log(0,33)] \approx 1,5848$
4. $H(X_4) = -[0,80 \log(0,80) + 0,20 \log(0,20) + 0 \log(0)] \approx 0,722$

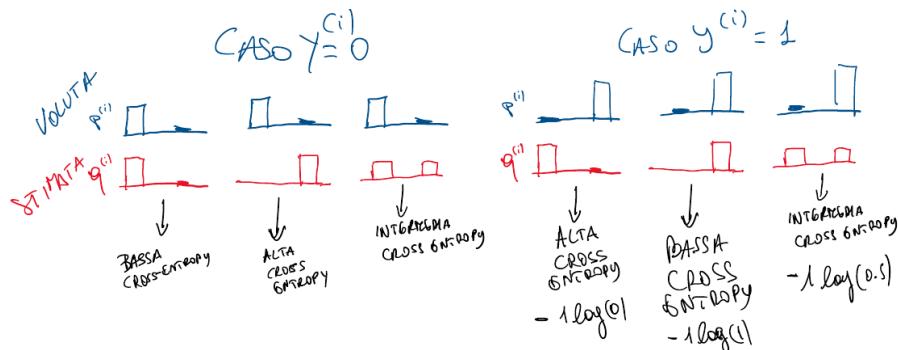
$$X_1 = X_2 < X_4 < X_3$$

4.4.3 Binary Cross Entropy Loss

La seguente formula è equivalente alla nostra funzione di Loss, ma fornisce un'interpretazione più intuitiva. Definiamo la **Binary Cross Entropy Loss** come:

$$H_{BCE}(X) = \frac{1}{m} \sum_{i=1}^m \left(- \sum_{k=1}^K \underbrace{p_k^{(i)}}_{(a)} \log_2 \underbrace{q_k^{(i)}}_{(b)} \right)$$

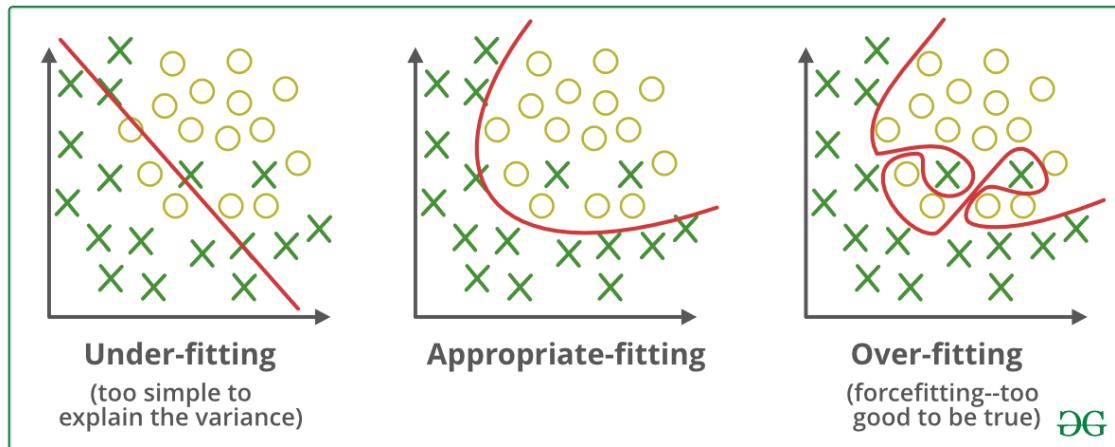
Dove (a) è la distribuzione vera della classificazione, detta **ground truth**, ovvero quella con probabilità massima sulla classe attesa, mentre (b) è la distribuzione data dal classificatore. Misura quindi la **distanza tra le due distribuzioni**. Minimizzare $H_{BCE}(X)$, significherà rendere le distribuzioni ideali e quelle effettive del classificatore, quanto più simili possibile.



4.5 Overfitting nella classificazione

Rischiamo di cadere in overfitting nell'utilizzo di classificatori polinomiali. Basterà usare i termini di regolarizzazione per diminuire o annullare l'impatto di alcuni termini di grado eccessivamente alto, ottenendo così:

$$J(\vartheta) = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log(h_\vartheta(x)) + (1 - y^{(i)}) \log(1 - h_\vartheta(x))) + \frac{\lambda}{2m} \sum_{j=1}^n \vartheta_j^2$$



4.6 Reject Region, regione d'incertezza

Sia nella classificazione binaria, che in quella multclasse, è possibile osservare casi in cui la probabilità stimata che un input appartenga ad una tra le classi specificate, non superi un determinato valore di certezza. Sono dei casi limite che vanno gestiti in maniera opportuna:

1. Prendere il valore con certezza più alta, se è proprio obbligatorio stabilire una classe, e il task non è particolarmente delicato.
2. Scartare l'input. Un'opzione migliore in campi più delicati, come quello medico.

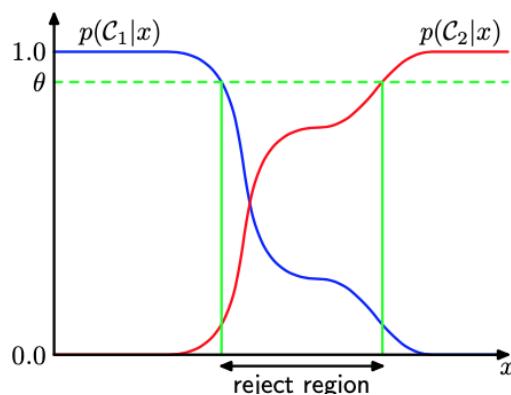


Figura 4.1: La zona d'incertezza (in cui $P(C_1|x) \sim P(C_2|x)$)

4.7 Classificazione multiclasse, approccio naive OVA e OVO

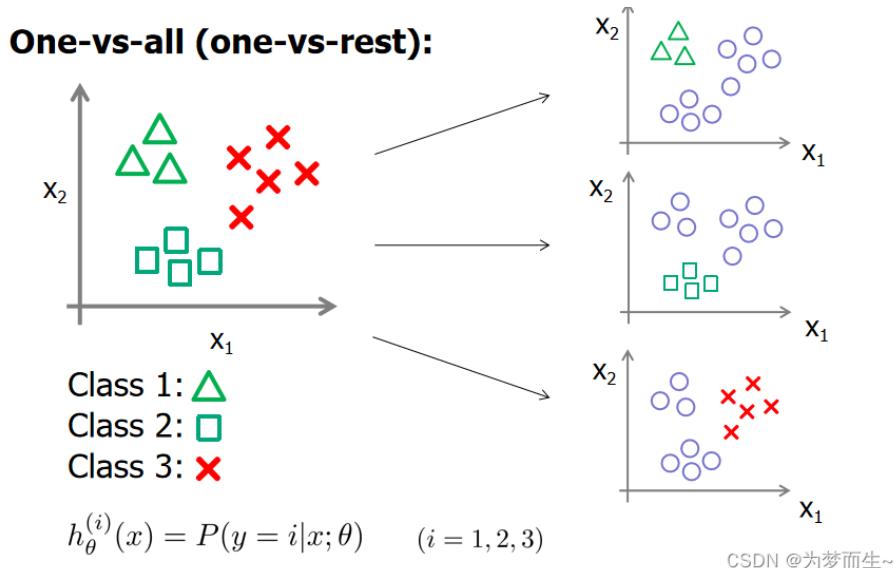
La classificazione multiclasse si distingue da quella binaria per il numero di classi. Il dataset di input non avrà più etichette binarie, ma k etichette. A ogni classe si associa un numero, ai fini di semplicità.

4.7.1 Metodo One vs All

Possiamo ottenere una classificazione multiclasse da dei classificatori binari, usando la metodologia **one vs all**. Immaginiamo di avere un sistema di classificazione figure geometriche, con le classi *triangolo*, *quadrato* e *cerchio* (c_1, c_2, c_3). Avremo bisogno di tre classificatori, $h_\theta^1, h_\theta^2, h_\theta^3$, capaci di misurare $P(\text{triangolo}|x), P(\text{quadrato}|x), P(\text{cerchio}|x)$. Prenderemo poi l'etichetta associata al valore di probabilità più alto

$$\hat{k} = \arg \max_k h_\theta^k(\bar{x})$$

ottenendo effettivamente una classificazione multiclasse.



Il training è effettuato sui singoli classificatori binari. Osserviamo inoltre che, con k classi:

Numero di classificatori richiesti in one vs all = k

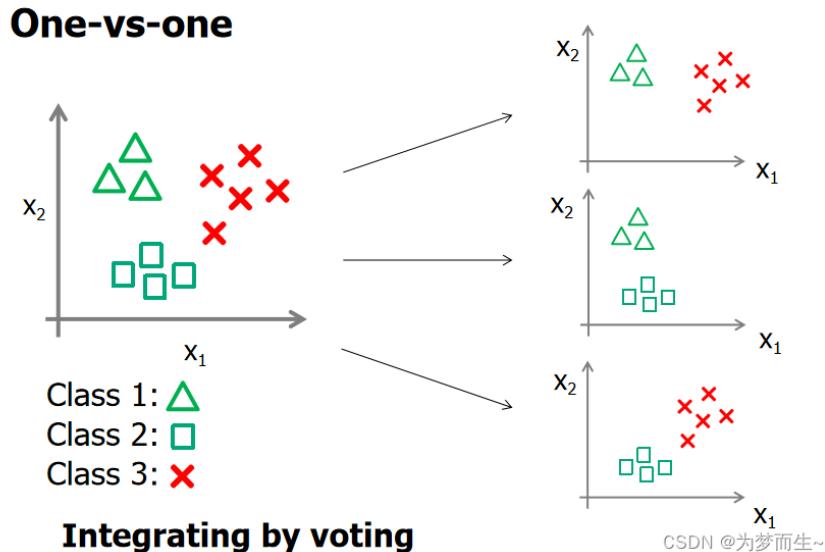
4.7.2 Metodo One vs One

In questo caso, i classificatori distinguono classi a due a due. Si fanno poi le opportune valutazioni per capire quale classe assegnare. Il numero di classificatori risulta essere più alto. Con k classi abbiamo:

$$\text{Numero di classificatori richiesti in one vs one} = \frac{k(k - 1)}{2}$$

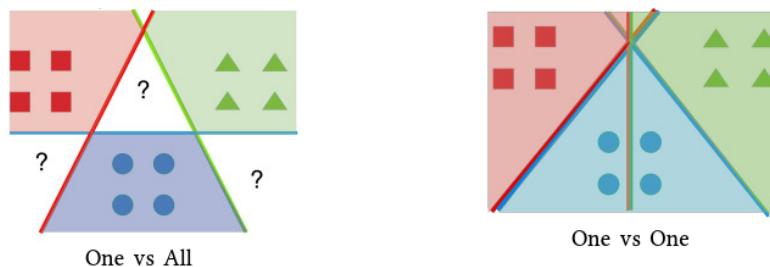
Immaginiamo un classificatore per lo stesso problema precedentemente esposto: otteniamo le classi (c_1, c_2, c_3) . Questi classificatori su coppie, daranno poi dei risultati. La classe più votata vince:

$$c_1 \text{ vs } c_2, \quad c_2 \text{ vs } c_3, \quad c_1 \text{ vs } c_3$$



4.7.3 Confronto tra *One vs All* e *One vs One*

Il modello one vs one, per quanto più oneroso in termini di numero di classificatori binari richiesti, non presenta l'area di incertezza su tutte le classi, che invece possiamo trovare nel classificatore one vs all.



4.8 Stochastic Gradient Descent

L'algoritmo di discesa del gradiente, computa $J(\vartheta)$ su **tutti** i campioni del training set: associando al numero di campioni il valore m , a d il numero di feature e ad e le epoche di training, otteniamo un numero di operazioni complessive pari a

$$m \times d \times e$$

Questo prodotto raggiunge facilmente valori molto alti. Un modo per ridurre il numero di operazioni, senza intaccare il risultato del training, e arrivare a convergenza, è l'uso della **SGD**, ossia della **discesa del gradiente stocastica**. Ad ogni iterazione del training si campiona un **mini-batch** di campioni di dimensione $b \leq m$. Questo campione è scelto randomicamente a ogni iterazione.

$$b \times d \times e$$

L'algoritmo deve arrivare a convergenza: se non arriva a convergenza, si ripete il campionamento e si ripete il training.

4.9 Softmax - Classificazione Multiclasse

Abbiamo quindi visto com'è possibile implementare un classificatore multiclassa utilizzando più classificatori binari. Tuttavia, questa strategia (a prescindere da OVA e OVO) presenta delle criticità:

- Numero di classificatori elevato, che porta a tempi di training lunghi.
- Possibili aree di incertezza.
- Difficoltà nell'interpretare le probabilità restituite dai classificatori.

Si può ovviare a questi problemi con il modello **Softmax**, che estende la regressione logistica alla classificazione multiclassa.

4.9.1 Definizione

Sia K il numero di classi da prevedere. Il modello Softmax restituisce un vettore $P_{\vartheta}^{(k)}$ di lunghezza K , in cui ogni elemento $P_{\vartheta}^{(i)}$ rappresenta la **probabilità** che l'input x appartenga alla classe i . L'obiettivo è quindi, cercare di **stimare**:

$$P_{\vartheta}^{(k)}(y = k|x) \quad \forall k \in \{1, \dots, K\}$$

la somma di ogni elemento del vettore, è sempre = 1.

4.9.2 Funzione Softmax

Partiamo dall'idea che il softmax restituisce un vettore di probabilità:

$$h_{\vartheta}^{(c)}(x) = \begin{pmatrix} P_{\vartheta}^{(1)}(y = 1|x) \\ P_{\vartheta}^{(2)}(y = 2|x) \\ \vdots \\ P_{\vartheta}^{(K)}(y = K|x) \end{pmatrix} = \frac{1}{\sum_{k=0}^{K-1} e^{\vartheta^{(k)}T x}} \cdot \begin{pmatrix} e^{\vartheta^{(1)}T x} \\ e^{\vartheta^{(2)}T x} \\ \vdots \\ e^{\vartheta^{(K)}T x} \end{pmatrix} = \frac{e^{\vartheta^{(c)}T x}}{\sum_{k=0}^{K-1} e^{\vartheta^{(k)}T x}}$$

Normalizzatore della distribuzione

Dove c è la classe di cui vogliamo calcolare la probabilità, k è l'indice che scorre tutte le classi, e $\vartheta^{(k)}$ è il vettore dei parametri associati alla classe k .

N.B. Ogni classe k ha il proprio vettore di parametri $\vartheta^{(k)}$ (i pesi del modello). Inoltre, una volta finito il learning del modello, la somma delle probabilità restituite dal softmax sarà sempre pari a 1 ($\sum_{k=1}^K P_{\vartheta}^{(k)}(y = k|x) = 1$).

4.9.3 Funzione di Costo

La funzione di costo del softmax è una funzione chiamata Cross Entropy Loss. Prima di poter definire la funzione di costo, definiamo una variabile ausiliaria $1\{\text{proposizione}\}$ che vale 1 se la proposizione è vera, 0 altrimenti. La funzione di costo del softmax è quindi:

$$J(\vartheta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{c=1}^K 1\{y^{(i)} = c\} \log P_{\vartheta}^{(c)}(y = c|x^{(i)}) \right] + \underbrace{\frac{\lambda}{2} \sum_{c=1}^K \sum_{j=1}^n (\vartheta_j^{(c)})^2}_{\text{Termine di regolarizzazione}}$$

Dove il termine di regolarizzazione serve a prevenire l'overfitting del modello.

4.9.4 Obiettivo di Learning

L'obiettivo di learning è quello di minimizzare la funzione di costo, trovando i parametri ottimali ϑ :

$$\vartheta = \arg \min_{\vartheta} J(\vartheta)$$

Ovvero , trovare i parametri che minimizzano la distanza tra la distribuzione di probabilità predetta dal modello e la distribuzione di probabilità reale (ground truth).

Per fare questo possiamo usare la discesa del gradiente, per cui abbiamo bisogno della derivata parziale della funzione di costo rispetto ai parametri $\vartheta_j^{(c)}$:

$$\frac{\partial J(\vartheta)}{\partial \vartheta_j^{(c)}} = -\frac{1}{m} \sum_{i=1}^m \left[\left(1\{y^{(i)} = c\} - P_{\vartheta}^{(c)}(y = c|x^{(i)}) \right) x_j^{(i)} \right] + \underbrace{\lambda \vartheta_j^{(c)}}_{\text{Termine di regolarizzazione}}$$

E da questa, aggiorniamo i pesi $\vartheta_j^{(c)}$ utilizzando la regola di aggiornamento della discesa del gradiente:

$$\vartheta_j^{(c)} \leftarrow \vartheta_j^{(c)} - \alpha \frac{\partial J(\vartheta)}{\partial \vartheta_j^{(c)}}, \quad \forall j \in [0, n], \quad \forall c \in [0, K - 1]$$

4.9.5 Proprietà del Softmax

Il modello softmax è **overparametrizzato**: per ogni ipotesi che potrebbe essere fatta sui dati, esistono infinite combinazioni di parametri che a partire dal vettore di feature $X \in \mathbb{R}^n$ producono la stessa predizione nel vettore di probabilità Y .

Per dimostrarlo, partiamo da:

$$\begin{aligned}
 h_{\vartheta}^{(c)}(x) &= \frac{e^{\vartheta^{(c)\top} x}}{\sum_{k=0}^{K-1} e^{\vartheta^{(k)\top} x}} && \text{Formula del Softmax} \\
 &= \frac{e^{(\vartheta^{(c)} - \psi)^{\top} x}}{\sum_{k=0}^{K-1} e^{(\vartheta^{(k)} - \psi)^{\top} x}} && \text{Sottraiamo lo stesso } \psi \\
 &= \frac{e^{\vartheta^{(c)\top} x} e^{-\psi^{\top} x}}{\sum_{k=0}^{K-1} e^{\vartheta^{(k)\top} x} e^{-\psi^{\top} x}} && \text{Separiamo gli esponenziali} \\
 &= \frac{e^{\vartheta^{(c)\top} x}}{\sum_{k=0}^{K-1} e^{\vartheta^{(k)\top} x}} && \text{Si cancella il fattore comune } e^{-\psi^{\top} x}
 \end{aligned}$$

Dove $\psi \in \mathbb{R}^n$ è un vettore arbitrario. Quindi, sottraendo lo stesso vettore ψ da tutti i vettori di parametri $\vartheta^{(k)}$, otteniamo gli stessi valori di probabilità predetti dal modello, perciò esistono infinite combinazioni di parametri che producono la stessa predizione.

Generalizzazione funzione di costo. È facile vedere che, per $K = 2$ (senza regolarizzazione) vale:

$$\begin{aligned}
 J(\vartheta) &= -\frac{1}{m} \sum_{i=1}^m \sum_{c=0}^1 \mathbf{1}\{y^{(i)} = c\} \log(h_{\vartheta}^{(c)}(x^{(i)})) \\
 &= -\frac{1}{m} \sum_{i=1}^m \left[\mathbf{1}\{y^{(i)} = 0\} \log h_{\vartheta}^{(0)}(x^{(i)}) + \mathbf{1}\{y^{(i)} = 1\} \log h_{\vartheta}^{(1)}(x^{(i)}) \right] \\
 &= -\frac{1}{m} \sum_{i=1}^m \left[(1 - y^{(i)}) \log h_{\vartheta}^{(0)}(x^{(i)}) + y^{(i)} \log h_{\vartheta}^{(1)}(x^{(i)}) \right] \\
 &= -\frac{1}{m} \sum_{i=1}^m \left[(1 - y^{(i)}) \log(1 - h_{\vartheta}^{(1)}(x^{(i)})) + y^{(i)} \log h_{\vartheta}^{(1)}(x^{(i)}) \right],
 \end{aligned}$$

dove si è usata la normalizzazione della softmax $h_{\vartheta}^{(0)}(x) + h_{\vartheta}^{(1)}(x) = 1$. Questa è esattamente la loss della regressione logistica binaria: dunque la regressione logistica è un caso particolare della softmax.

Si può ulteriormente dimostrare che il softmax è una generalizzazione della regressione logistica multiclasse usando una sua proprietà principale: l'*overparametrizzazione*. Per

$K = 2$:

$$h_{\vartheta}(x) = \begin{bmatrix} e^{\vartheta^{(0)\top} x} \\ e^{\vartheta^{(0)\top} x} + e^{\vartheta^{(1)\top} x} \\ e^{\vartheta^{(1)\top} x} \\ e^{\vartheta^{(0)\top} x} + e^{\vartheta^{(1)\top} x} \end{bmatrix} = \begin{bmatrix} e^{(\vartheta^{(0)} - \vartheta^{(1)})^\top x} \\ e^{(\vartheta^{(0)} - \vartheta^{(1)})^\top x} + e^{\mathbf{0}^\top x} \\ e^{(\vartheta^{(1)} - \vartheta^{(1)})^\top x} \\ e^{(\vartheta^{(0)} - \vartheta^{(1)})^\top x} + e^{\mathbf{0}^\top x} \end{bmatrix} = \begin{bmatrix} e^{\Delta^\top x} \\ 1 + e^{\Delta^\top x} \\ 1 \\ 1 + e^{\Delta^\top x} \end{bmatrix}, \quad \Delta = \vartheta^{(0)} - \vartheta^{(1)}.$$

Ponendo $\mathbf{w} = \vartheta^{(1)} - \vartheta^{(0)} = -\Delta$ e $\sigma(z) = \frac{1}{1 + e^{-z}}$ si ottiene

$$h_{\vartheta}(x) = \begin{bmatrix} 1 - \sigma(\mathbf{w}^\top x) \\ \sigma(\mathbf{w}^\top x) \end{bmatrix}.$$

Quindi, per $K = 2$, il softmax riduce alla regressione logistica binaria.

4.9.6 Rete neurale Softmax

Il softmax può essere visto come un particolare tipo di rete neurale:

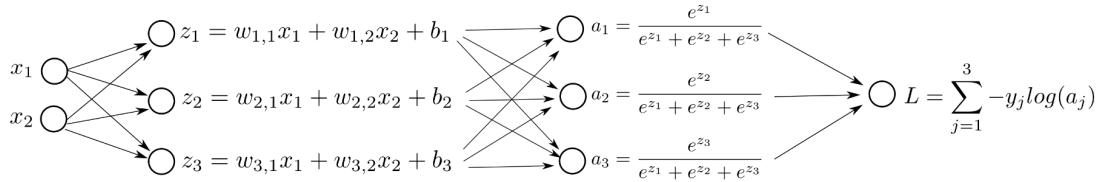


Figura 4.2: Flusso della regressione softmax con due ingressi e tre classi: dai logit $z_i = w_{i,1}x_1 + w_{i,2}x_2 + b_i$ alle probabilità $a_i = \frac{e^{z_i}}{\sum_{k=1}^3 e^{z_k}}$, fino alla loss a entropia incrociata $L = \sum_{j=1}^3 -y_j \log(a_j)$.

4.9.7 Funzione di Loss

La funzione di Loss usata nel softmax normalmente è una variante di **cross entropy loss** (molto simile alla funzione di costo). Si possono usare alternative, in particolare la **Hinge Loss** trasforma la classificazione multiclasse in un problema di massimizzazione del margine tra le classi, come nelle SVM. La Hinge Loss è definita come:

$$\text{Loss}(h_{\vartheta}(x), y) = \sum_{i \neq y} \max(0, h_{\vartheta}(x)_i - h_{\vartheta}(x)_y + \Delta)$$

Capitolo 5

Design, valutazione e scelta dei parametri di un algoritmo di ML

5.1 Momento e gradiente

Una variante dell'algoritmo di discesa del gradiente, ovvero **la discesa del gradiente con momento**, modifica la nostra formula del gradiente nel seguente modo:

$$\vartheta_j^{\text{new}} \leftarrow \vartheta_j^{\text{old}} - \alpha z^{\text{new}}$$

$$z^{\text{new}} \leftarrow \beta z^{\text{old}} + \frac{\partial J(\vartheta^{\text{old}})}{\partial \vartheta^{\text{old}}}$$

l'idea è quella di mantenere in z una **media dei gradienti recenti**, in modo da mantenere coerenza durante la convergenza: valori incoerenti del gradiente sono **rumore**, e questo **può causare zig-zag e rallentamenti nella convergenza**. Tenendo uno storico dei valori del gradiente (usando il **momentum** β come parametro che stabilisce l'influenza dei vecchi valori su quelli nuovi), diventa possibile **annullare parzialmente o totalmente l'effetto del rumore**. Accelera anche la convergenza stessa, **rinforzando i passi coerenti**. Si accorcia quindi il training, e si migliorano le performance a parità di epoche. Osservazioni statistiche consigliano l'utilizzo di $\beta = 0.90$ per ottenere risultati migliori.

5.2 Design di Algoritmi

Progettare bene un algoritmo di ML permette di ottenere migliori risultati in meno tempo e con meno risorse computazionali. Si pensi al cercare di inserire a casaccio i parametri di un modello, senza una logica precisa: si rischia di perdere tempo, se invece si pensa a come variare i parametri in maniera sistematica, per capire l'impatto di ciascuno di essi sul modello si potrebbe ottenere un modello migliore in meno tempo.

5.2.1 Strategie di miglioramento

Supponiamo di aver implementato la regressione lineare con regolarizzazione, ma gli errori su nuovi campioni sono comunque troppo alti. Potremmo pensare ad alcune strategie per migliorare il modello:

Aumentare i campioni di Training. Non è sempre una scelta utile, anche perché se il modello non funziona su nuovi campioni significa che il problema (molto probabilmente) è di overfitting. Aumentare i campioni di training potrebbe aiutare, ma non è detto che lo faccia.

Ridurre il set di Features. Si potrebbe pensare di ridurre il numero di features in input al modello, in modo da semplificare il problema e ridurre il rischio di overfitting. Questa tecnica può essere efficace se alcune features sono ridondanti o non rilevanti per il task. Tuttavia, è importante fare attenzione a non eliminare informazioni utili.

Cercare nuove features. L'opposto della riduzione delle features: il problema è di underfitting, in quanto il modello non ha abbastanza informazioni per generalizzare bene. Aggiungere nuove features può aiutare a migliorare le prestazioni del modello ma richiede un'attenta selezione delle stesse per evitare di introdurre rumore.

Utilizzare features polinomiali. Trasformare le features esistenti in polinomiali può aiutare a catturare relazioni non lineari tra le variabili. Questa tecnica può essere utile se si sospetta che il modello lineare non sia sufficiente per rappresentare i dati. Tuttavia, l'aggiunta di termini polinomiali aumenta la complessità del modello e il rischio di overfitting, quindi è importante bilanciare questa scelta con tecniche di regolarizzazione.

Far variare il parametro di regolarizzazione λ . Modificare il parametro di regolarizzazione può influenzare significativamente le prestazioni del modello. Un valore più alto di λ penalizza maggiormente i pesi del modello, riducendo il rischio di overfitting, mentre un valore più basso consente al modello di adattarsi meglio ai dati di training, ma aumenta il rischio di overfitting. È importante trovare un equilibrio ottimale attraverso tecniche come la validazione incrociata (che vedremo più avanti).

Solitamente si cerca di effettuare test di diagnostica applicati al modello, per permettere di ottenere informazioni dettagliate su cosa funziona o non funziona con l'obiettivo di ottenere indicazioni per migliorare le prestazioni.

5.3 Valutazione

La valutazione di un algoritmo di ML è fondamentale per capire se il modello è adatto al task.

5.3.1 Valutazione incrociata

Un modo di fare valutazione incrociata è una variante della k-Fold Cross Validation: non si suddivide più in training e test sets, ma piuttosto in training e validation sets. In questo modo si può utilizzare il validation set per scegliere i parametri del modello, mentre il test set viene utilizzato solo alla fine per valutare le prestazioni finali del modello.

La procedura è la seguente:

1. Si suddivide il dataset originale in training/validation set e un test set che non verrà usato fino alla fine della valutazione.
2. Si suddivide il dataset rimanente in t splits e in k folds.
3. Si seleziona uno split come validation set e gli altri $t - 1$ come training set.
4. Si allena il modello sui $t - 1$ training splits.
5. Si valuta il modello con una funzione J_{VAL} sul validation fold (non usato per il training).
6. Si ripete il processo per tutti i t splits, ottenendo t valori di J_{VAL} .
7. Si calcola la media dei t valori di J_{VAL} per ottenere una stima complessiva delle prestazioni del modello.
8. Si ripete l'intero processo per tutti i possibili set di iperparametri del modello.
9. Si selezionano gli iperparametri che hanno prodotto la migliore prestazione media sul validation set.
10. Infine, si valuta il modello finale con gli iperparametri selezionati sul test set per ottenere una stima finale delle prestazioni del modello.

In questo modo il modello è robusto perché è stato allenato su diversi training set per i dati, allenato sui validation set per gli iperparametri e testato su un test set **mai visto prima**.

Esempio. Ipotizziamo di avere un certo dataset, dividiamolo in un test set (20% dei dati) e in un training/validation set (80% dei dati). Definiamo le funzioni di costo:

- **Funzione di costo di training:**

$$J_{TR}(\vartheta) = \frac{1}{2m_{TR}} \sum_{i=1}^{m_{TR}} \left(h_{\vartheta}(x_{TR}^{(i)}) - y_{TR}^{(i)} \right)^2$$

Dove m_{TR} è il numero di campioni nel training set, $x_{TR}^{(i)}$ e $y_{TR}^{(i)}$ sono rispettivamente l'input e l'output del i -esimo campione del training set.

- **Funzione di costo di validation:**

$$J_{VAL}(\vartheta) = \frac{1}{2m_{VAL}} \sum_{i=1}^{m_{VAL}} \left(h_{\vartheta}(x_{VAL}^{(i)}) - y_{VAL}^{(i)} \right)^2$$

Dove m_{VAL} è il numero di campioni nel validation set, $x_{\text{VAL}}^{(i)}$ e $y_{\text{VAL}}^{(i)}$ sono rispettivamente l'input e l'output del i -esimo campione del validation set.

- **Funzione di costo di test:**

$$J_{\text{TEST}}(\vartheta) = \frac{1}{2m_{\text{TEST}}} \sum_{i=1}^{m_{\text{TEST}}} (h_{\vartheta}(x_{\text{TEST}}^{(i)}) - y_{\text{TEST}}^{(i)})^2$$

Dove m_{TEST} è il numero di campioni nel test set, $x_{\text{TEST}}^{(i)}$ e $y_{\text{TEST}}^{(i)}$ sono rispettivamente l'input e l'output del i -esimo campione del test set.

Supponiamo di voler utilizzare $t = 5$ splits e $k = 5$ folds. La suddivisione del training/validation set sarà la seguente:

	FOLD 1	FOLD 2	FOLD 3	FOLD 4	FOLD 5
SPLIT 1	█	█	█	█	█
SPLIT 2	█	█	█	█	█
SPLIT 3	█	█	█	█	█
SPLIT 4	█	█	█	█	█
SPLIT 5	█	█	█	█	█

Figura 5.1: Dataset diviso in split e folds per la valutazione incrociata.

Dopo aver suddiviso i dati, ognuno dei dati produrrà una stima di $J_{\text{VAL}}^H(\vartheta^{(i)})$ per un certo i -esimo set valutato sull'iperparametro H . Da queste andiamo a fare la media:

$$\bar{J}_{\text{VAL}}^H(\vartheta) = \frac{1}{t} \sum_{i=1}^t J_{\text{VAL}}^H(\vartheta^{(i)})$$

Ottenuta la media $\bar{J}_{\text{VAL}}^H(\vartheta)$ per ogni set di iperparametri H , si seleziona quello che minimizza la funzione di costo media:

$$H^* = \arg \min_H \bar{J}_{\text{VAL}}^H(\vartheta)$$

Una volta fissato il miglior iperparametro H^* , si allena il modello sul test set per ottenere la stima finale:

$$J_{\text{TEST}}^{H^*}(\vartheta)$$

5.3.2 Bias e Varianza

Per valutare le prestazioni di un modello possiamo usare i valori del bias e della varianza per fare delle stime su come il modello si comporta sui dati.

Bias. Ricordiamo che il bias è **l'errore sistematico** che il modello commette sui dati. Un modello con alto bias tende a sottostimare la complessità del problema, portando a errori

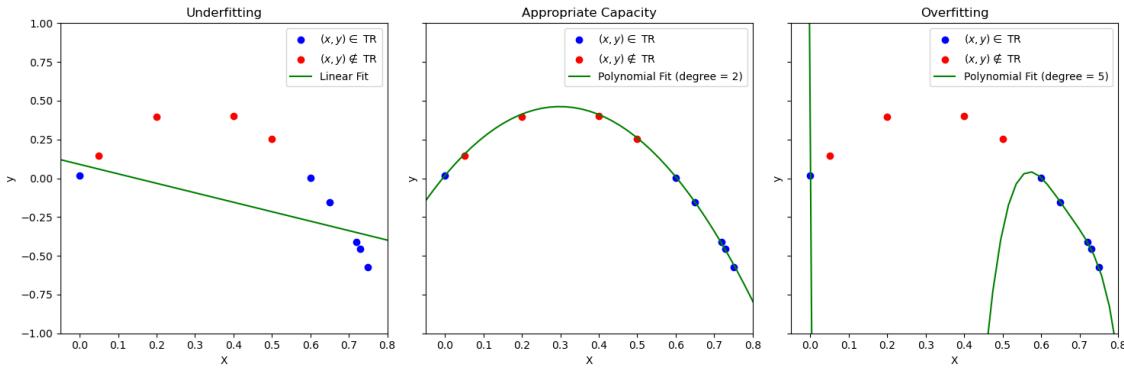


Figura 5.2: Underfitting—capacità adeguata—overfitting in regressione polinomiale: punti blu = dati di training (TR), punti rossi = fuori da TR; la linea verde mostra il fit del modello: lineare (sinistra), polinomio di grado 2 (centro) e polinomio di grado 5 (destra).

elevati sia sul training set che sul test set. Questo fenomeno è noto come **underfitting**. Nell'immagine 5.2, il grafico a sinistra mostra un esempio di underfitting, dove il modello lineare non riesce a catturare la relazione tra le variabili e commette un errore sistematico sia sul training set (punti blu) che sul test set (punti rossi).

Varianza. La varianza rappresenta la sensibilità del modello alle variazioni nei dati di training. Un modello con alta varianza tende a sovradattarsi ai dati di training, catturando il rumore invece della vera relazione tra le variabili. Questo porta a errori bassi sul training set ma elevati sul test set, fenomeno noto come **overfitting**.

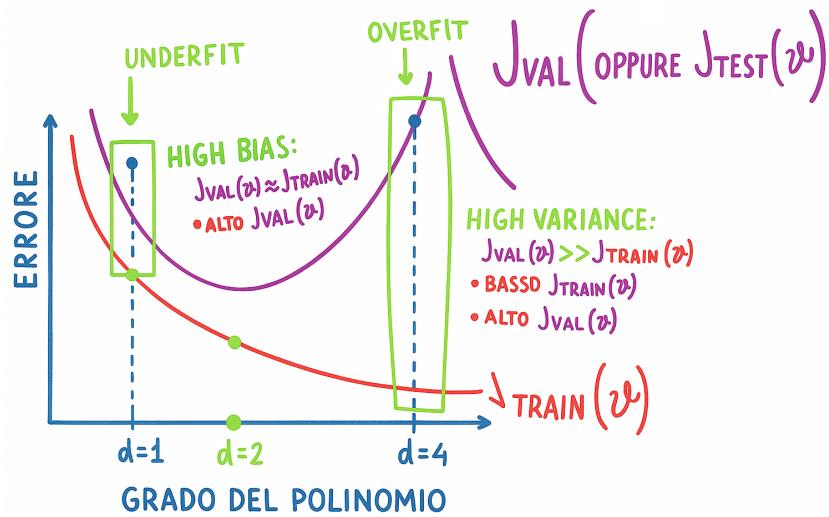


Figura 5.3: Trade-off tra bias e varianza in funzione della complessità del modello.

Nella figura 5.2, il grafico a destra mostra un esempio di overfitting, dove il modello polinomiale è troppo complesso e si adatta troppo strettamente ai dati di training, risultando

in prestazioni scadenti sui dati di test¹. Come si vede dall'immagine al centro della figura 5.2, un modello con capacità adeguata riesce a bilanciare bias e varianza, ottenendo buone prestazioni sia sul training set che sul test set. Per valutare e risolvere i problemi sul modello, possiamo andare a controllare come Bias e Varianza variano al variare dei parametri del modello² e notiamo che si può costruire una correlazione tra l'errore commesso dal modello (quindi o in J_{TEST} o in J_{VAL}) e la complessità del polinomio (il grado):

Parametro di regolarizzazione. Anche i parametri di regolarizzazione, esattamente come il modello (i suoi parametri), devono essere stabiliti sperimentalmente. Si parte da $\lambda^0 = 0$, testando valori di $\lambda^{(1)}, \lambda^{(2)} \dots \lambda^{(k)}$ sempre più grandi, verificando come cambia $J_{VAL}(\vartheta^i)$ in funzione di λ^i :

$$\begin{bmatrix} \lambda^{(0)} \\ \lambda^{(1)} \\ \lambda^{(2)} \\ \lambda^{(3)} \\ \vdots \\ \lambda^{(k)} \end{bmatrix} \Rightarrow \min_{\vartheta} J(\vartheta) \Rightarrow \begin{bmatrix} \vartheta^{(0)} \\ \vartheta^{(1)} \\ \vdots \\ \vartheta^{(k)} \end{bmatrix} \begin{bmatrix} J_{VAL}(\vartheta^{(0)}) \\ J_{VAL}(\vartheta^{(1)}) \\ \vdots \\ J_{VAL}(\vartheta^{(k)}) \end{bmatrix} \Rightarrow \underbrace{\min_{J_{VAL}}}_{\text{BEST}} \Rightarrow J_{TEST}(\vartheta^{\text{BEST}})$$

In questo modo si riesce a trovare il miglior parametro di regolarizzazione che minimizza l'errore sul validation set, e si può usare questo parametro per valutare il modello sul test set.

Altre curve. Un'altra curva potrebbe essere quella che mostra l'**errore** su *train set* e *validation set* in funzione del numero di campioni di training (figura 5.4). In questo modo si può capire se il modello soffre di overfitting o underfitting:

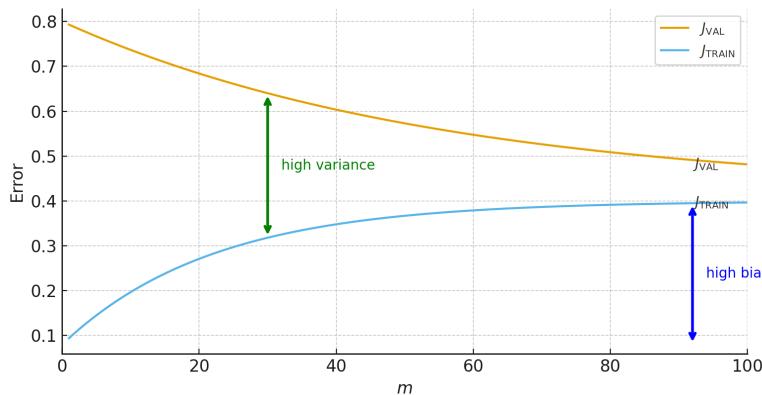


Figura 5.4: Learning curves: errore su training J_{TRAIN} e su validation J_{VAL} in funzione del numero di esempi m . J_{VAL} decresce, J_{TRAIN} cresce leggermente e il *generalization gap* (freccia) si riduce; le curve si avvicinano all'errore irreducibile.

¹Si noti che basterebbe rimuovere un punto del training set per far cambiare completamente il modello, in quanto altamente sensibile ai dati di input.

²In quella che molti chiamano **bias-variance trade-off**.

La figura conferma il fatto che esiste il trade-off bias-varianza, in quanto:

- Se siamo in presenza di *high bias*, quindi un alto gap tra la curva di train e l'asse delle ascisse, aumentare il numero di dati di training non aiuta molto.
- Se siamo in presenza di *high variance*, quindi un grande gap tra l'errore commesso tra il training set e il validation set, aumentare il numero di dati di training potrebbe aiutare a migliorare i risultati (non sempre).

Capitolo 6

Introduzione alle reti neurali

Capitolo 7

Decision Tree

Gli alberi decisionali (decision tree) sono modelli utilizzati per la classificazione e la regressione. Essi rappresentano un insieme di regole decisionali organizzate in una struttura ad albero, dove ogni nodo interno rappresenta una condizione su una caratteristica dei dati, ogni ramo rappresenta l'esito della condizione e ogni foglia rappresenta una classe o un'etichetta di output.

7.1 Alberi di classificazione

Un albero decisionale per la classificazione è una struttura gerarchica composta da nodi interni e foglie. Ogni nodo interno rappresenta un test su una caratteristica specifica del dataset, mentre ogni foglia rappresenta una classe di output. L'obiettivo principale di un albero decisionale è suddividere il dataset in modo tale che le tuple appartenenti alla stessa classe siano raggruppate insieme nelle foglie dell'albero.

Questo approccio, comunemente chiamato **divide and conquer**, suddivide iterativamente il dataset in sottoinsiemi più piccoli basati su condizioni specifiche, fino a quando non si raggiungono le foglie dell'albero che rappresentano le decisioni finali.

7.1.1 Divisione ricorsiva

La classe di una tupla q si ottiene seguendo il cammino radice → foglia guidato dai blocchi condizionali sui nodi interni. Ogni nodo interno applica un test su una caratteristica A e, in base al risultato del test, si procede lungo il ramo corrispondente fino a raggiungere una foglia che fornisce la classificazione finale. L'insieme di regole *deve* essere **esauritivo** e **mutuamente esclusivo**¹ (ogni tupla deve essere classificata da una e una sola regola).

La costruzione dell'albero decisionale è molto semplice:

1. Se tutte le tuple del nodo X hanno la *stessa* classe C , crea una foglia C .
2. Altrimenti scegli un attributo A (non ancora usato) e *ramifica* X (*splitting*) secondo i valori/soglia di A ; crea i figli.

¹Se così non fosse, alcune tuple potrebbero non essere classificate o potrebbero essere classificate da più regole contemporaneamente

3. Per ogni figlio X_i : se puro, fermati; se impuro, ripeti ricorsivamente.

Pruning. Se le tuple nel nodo sono poche o la profondità è elevata, si può fermare prima e rendere il nodo una foglia (vedere figura 7.1 con l'attributo "overcast").

Outlook	Temperature	Humidity	Windy	Class
sunny	hot	high	false	N
sunny	hot	high	true	N
overcast	hot	high	false	P
rain	mild	high	false	P
rain	cool	normal	false	P
rain	cool	normal	true	N
overcast	cool	normal	true	P
sunny	mild	high	false	N
sunny	cool	normal	false	P
rain	mild	normal	false	P
sunny	mild	normal	true	P
overcast	mild	high	true	P
overcast	hot	normal	false	P
rain	mild	high	true	N

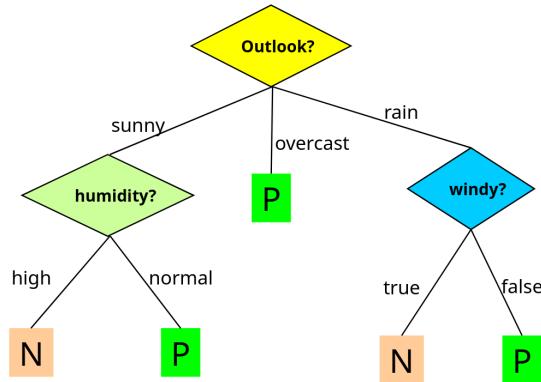


Figura 7.1: Dataset *weather* (a sinistra) e albero decisionale appreso (a destra). La tabella contiene 14 esempi con quattro attributi descrittivi (**Outlook**, **Temperature**, **Humidity**, **Windy**) e la classe binaria **P/N**. L'albero (stile ID3/C4.5) sceglie come radice **Outlook**; il ramo **overcast** porta direttamente alla classe **P**, mentre per **sunny** si testa **Humidity** e per **rain** si testa **Windy**. L'esempio illustra il passaggio da dati tabellari a regole interpretabili.

7.1.2 Migliore divisione dei nodi

Diciamo che per il nodo m , N_m è il numero di tuple nel training set che raggiungono il nodo m (per la radice, $N_{root} = N$, il numero totale di tuple). Sia N_m^i il numero di tuple di classe C_i che raggiungono il nodo m , con $\sum_i N_m^i = N_m$. Sapendo che una tupla x raggiunge il nodo m , la probabilità stimata che appartenga alla classe C_i è:

$$\hat{P}(C_i|x, m) \equiv p_m^i = \frac{N_m^i}{N_m}$$

Definiamo un **nodo puro** come un nodo in cui tutte le tuple appartengono alla stessa classe:

$$\exists i : N_m^i = N_m \implies p_m^i = 1 \quad \text{e} \quad \forall j \neq i : p_m^j = 0$$

ovvero la probabilità stimata di una classe è 1, mentre per tutte le altre è 0. Un nodo è **impuro** se contiene tuple di classi diverse. Un modo frequente per misurare l'impurità di un nodo è utilizzare l'entropia di Shannon:

$$H(m) = - \sum_i p_m^i \log_2(p_m^i)$$

L'entropia misura l'incertezza associata alla distribuzione delle classi nel nodo. Un nodo puro ha *entropia zero*, mentre un nodo con una distribuzione uniforme delle classi ha *entropia massima*.

Per trovare la miglior combinazione dei nodi eseguiamo la seguente procedura ricorsiva:

- Se il nodo è *puro*, creiamo una foglia con la classe corrispondente.
- Altrimenti, per ogni attributo A non ancora usato, si divide.

Per calcolare l'impurità dei nodi dopo lo splitting (un nodo impuro) bisogna diminuire il valore di impurità. Per calcolarlo:

$$\hat{P}(C_i|x, m, j) \equiv p_{mj}^i = \frac{N_{mj}^i}{N_{mj}}$$

dove N_{mj} è il numero di tuple che raggiungono il figlio j del nodo m e N_{mj}^i è il numero di tuple di classe C_i che raggiungono il figlio j del nodo m . Questo si traduce in una nuova entropia calcolata come:

$$H^j(m) = - \sum_{j=1}^n \frac{N_{mj}}{N_m} \sum_i^k p_{mj}^i \log_2(p_{mj}^i)$$

dove n è il numero di figli del nodo m e k è il numero di classi. L'entropia dopo lo splitting è una media pesata delle entropie dei figli, ponderata per la proporzione di tuple che raggiungono ciascun figlio.

7.2 Alberi di regressione

Un albero di regressione è un modello predittivo utilizzato per stimare valori continui basati su un insieme di caratteristiche. A differenza degli alberi decisionali per la classificazione, che suddividono i dati in classi discrete, gli alberi di regressione suddividono i dati in intervalli continui e forniscono una stima numerica come output.

7.2.1 Costruzione dell'albero di regressione

Ipotizziamo di avere un nodo m , con \mathcal{X}_m che rappresenta l'insieme \mathcal{X} delle tuple che raggiungono il nodo m . Quindi è l'insieme di tutti quei nodi x tali che $x \in \mathcal{X}$ e x raggiunge il nodo m . Possiamo definire:

$$b_m(x) = \begin{cases} 1 & \text{se } x \in \mathcal{X}_m \\ 0 & \text{altrimenti} \end{cases}$$

ovvero una funzione indicatrice che vale 1 se la tupla x raggiunge il nodo m e 0 altrimenti.

Nella regressione, la *goodness*² di una divisione viene misurata utilizzando la somma dei quadrati degli errori (SSE, Sum of Squared Errors). Per un nodo m , la SSE è definita

²Qualità

come:

$$E_m = \frac{1}{N_m} \sum_t (y^t - \hat{y}_m)^2 b_m(x^t)$$

dove y^t è il valore reale della tupla t , \hat{y}_m è la stima associata al nodo m , e N_m è il numero di tuple che raggiungono il nodo m . Si moltiplica per la funzione indicatrice $b_m(x^t)$ per considerare solo le tuple che effettivamente raggiungono il nodo m . La stima media \hat{y}_m per il nodo m è calcolata come:

$$\hat{y}_m = \frac{\sum_t b_m(x^t) y^t}{\sum_t b_m(x^t)} = \frac{1}{N_m} \sum_t b_m(x^t) y^t$$

dove la somma è calcolata su tutte le tuple t che raggiungono il nodo m . In pratica, \hat{y}_m è la media dei valori di output delle tuple che raggiungono il nodo m .

7.2.2 Divisione di un nodo

Per dividere un nodo m , si seleziona un attributo A e una soglia di divisione. La divisione crea più figli per il nodo m , ciascuno corrispondente a un intervallo di valori dell'attributo A . Dopo la divisione, si calcola l'errore E_m^j per ciascun figlio j del nodo m .

Se l'errore E_m viene ritenuto accettabile, ovvero $E_m < \theta_r$ per una certa soglia, si crea una foglia con la stima \hat{y}_m . Altrimenti, se l'errore non è accettabile, si procede viene diviso tante volte finché non si raggiunge una stima accettabile o si esauriscono gli attributi disponibili per la divisione. Si definisce \mathcal{X}_{mj} come l'insieme delle tuple che raggiungono il figlio j del nodo m sottoinsieme di \mathcal{X}_m . Definiamo ulteriormente:

$$b_{mj}(x) = \begin{cases} 1 & \text{se } x \in \mathcal{X}_{mj} \\ 0 & \text{altrimenti} \end{cases}$$

La stima media per il figlio j del nodo m è calcolata come:

$$\hat{y}_{mj} = \frac{\sum_t b_{mj}(x^t) y^t}{\sum_t b_{mj}(x^t)} = \frac{1}{N_{mj}} \sum_t b_{mj}(x^t) y^t$$

dove N_{mj} è il numero di tuple che raggiungono il figlio j del nodo m . L'errore dopo lo split è calcolato come:

$$E_m^j = \frac{1}{N_m} \sum_t \sum_j (y^t - \hat{y}_{mj})^2 b_{mj}(x^t)$$

dove la somma esterna è calcolata su tutte le tuple t che raggiungono il nodo m e la somma interna è calcolata su tutti i figli j del nodo m . Ipotizziamo di avere un nodo m , con X_m che rappresenta l'insieme X delle tuple che raggiungono il nodo m . Quindi è l'insieme di tutti quei nodi x tali che $x \in X$ e x raggiunge il nodo m .

7.3 Pruning: riduzione dell'albero

Il pruning è una tecnica utilizzata per ridurre la complessità di un albero decisionale (di classificazione) o di regressione, al fine di migliorare la sua capacità di generalizzazione sui dati non visti. Durante la costruzione dell'albero, è possibile che si creino nodi che si adattano troppo strettamente ai dati di addestramento, portando ad *overfitting*. Il pruning mira a rimuovere questi nodi superflui, semplificando l'albero e migliorando le prestazioni sui dati di test.

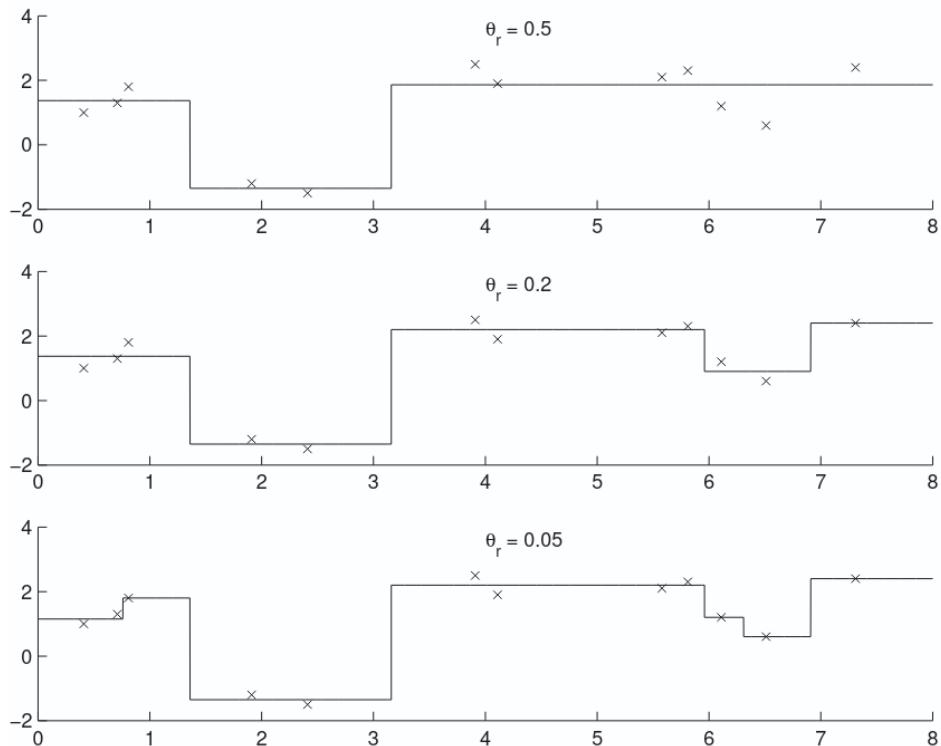


Figura 7.2: Esempio di albero di regressione in cui si mostra l'effetto del parametro di complessità θ_r sulla funzione stimata: al diminuire di θ_r l'albero produce più suddivisioni e una funzione a gradini più irregolare e aderente ai dati.

7.3.1 Pre-pruning

Il pre-pruning consiste nell'interrompere la crescita dell'albero prima che raggiunga la sua massima profondità. Durante la costruzione dell'albero, si valuta la bontà di ogni divisione e si decide di fermarsi se la divisione non migliora significativamente le prestazioni del modello. Questo può essere fatto utilizzando criteri come la riduzione dell'impurità o la diminuzione dell'errore di regressione.

7.3.2 Post-pruning

Il post-pruning, invece, avviene dopo che l'albero è stato completamente costruito. In questa fase, si esaminano i nodi dell'albero e si valutano le prestazioni del modello su un set di validazione. I nodi che non contribuiscono significativamente alla precisione del modello vengono rimossi, in quanto contribuiscono all'*overfitting*, e i loro figli vengono sostituiti con foglie che rappresentano la stima media o la classe più frequente delle tuple che raggiungevano quel nodo.

7.4 Regole di learning

Gli alberi decisionali