

# **Riassunti e Appunti di Introduzione al Data Mining**

Emanuele Galiano  
Studente di Informatica  
Università di Catania

Anno Accademico 2025/2026



# Indice

<b>I Concetti Preliminari</b>	<b>1</b>
<b>1 Prerequisiti Matematici Essenziali [WIP]</b>	<b>3</b>
<b>2 Grafi</b>	<b>5</b>
2.1 Definizione formale . . . . .	5
2.1.1 Network science . . . . .	6
2.2 Grafi diretti e indiretti . . . . .	6
2.3 Grafi pesati e grafi etichettati . . . . .	7
2.4 Gradi dei vertici . . . . .	7
2.4.1 Distribuzione dei gradi . . . . .	8
2.5 Grafi bipartiti . . . . .	8
2.6 Grafo completo vs Grafo regolare . . . . .	9
2.7 Cammini tra due nodi . . . . .	9
2.7.1 Cammino minimo . . . . .	9
2.7.2 Diametro . . . . .	10
2.7.3 Ciclo . . . . .	10
2.8 Connettività . . . . .	10
2.8.1 Connnettività forte e debole . . . . .	11
2.9 Coefficiente di Clustering . . . . .	11
2.10 Misure di centralità . . . . .	12
2.10.1 Centralità di grado . . . . .	12
2.10.2 Centralità di vicinanza . . . . .	12
2.10.3 Centralità di prossimità . . . . .	14
2.10.4 Centralità di PageRank . . . . .	14
<b>II Tecniche di Data Mining</b>	<b>17</b>
<b>3 Introduzione al Data Mining</b>	<b>19</b>
3.1 Definizione e finalità . . . . .	19
3.2 Caratteristiche dei pattern . . . . .	19
3.3 Metodi di data mining . . . . .	19
3.4 Perché fare data mining . . . . .	20
3.4.1 Big Data . . . . .	20

3.4.2	Dai dati alla conoscenza e alle comunità coinvolte . . . . .	20
3.5	Limiti e insidie del data mining . . . . .	20
3.5.1	Caso di studio: Total Information Awareness (TIA) . . . . .	20
3.5.2	Esempio: co-presenza in hotel come criterio di sospetto . . . . .	21
3.6	Principio di Bonferroni e test multipli . . . . .	22
<b>4</b>	<b>Preprocessing</b>	<b>23</b>
4.1	Estrazione di feature . . . . .	23
4.1.1	Tecniche di estrazione di feature . . . . .	24
4.2	Portabilità dei dati . . . . .	24
4.2.1	Da dati numerici a categorici . . . . .	24
4.2.2	Da dati categorici a numerici . . . . .	25
4.2.3	Da testo a dati numerici . . . . .	25
4.3	Cleaning dei dati . . . . .	25
4.3.1	Gestione dei valori mancanti . . . . .	25
4.3.2	Gestione dei valori errati . . . . .	26
4.3.3	Scala dei dati . . . . .	26
4.4	Riduzione dei dati . . . . .	27
4.4.1	Sampling dei dati . . . . .	27
4.4.2	Selezione di feature . . . . .	28
4.4.3	Riduzione della dimensionalità . . . . .	28
4.4.4	PCA: Principal Component Analysis . . . . .	28
4.4.5	SVD: Singular Value Decomposition . . . . .	30
4.4.6	LSA: Latent Semantic Analysis . . . . .	32
4.4.7	Riduzione di dimensionalità con trasformazione dei dati . . . . .	32
<b>5</b>	<b>Insiemi Frequenti e Regole d'Associazione</b>	<b>33</b>
5.1	Market-basket model e definizioni . . . . .	33
5.1.1	Supporto . . . . .	33
5.2	Regole d'associazione . . . . .	33
5.2.1	Qualità di una regola . . . . .	34
5.3	Insiemi frequenti chiusi e massimali . . . . .	34
5.4	Anti-monotonia e Principio di Apriori . . . . .	35
5.4.1	Principio di Apriori . . . . .	35
5.5	Algoritmo Apriori . . . . .	35
5.5.1	Esempio Apriori ( $\text{minsup} = 2$ ) . . . . .	36
5.5.2	Generazione dei candidati . . . . .	36
5.6	Ottimizzazioni di Apriori . . . . .	36
5.6.1	Hashing in bucket: PCY . . . . .	36
5.6.2	Partizionamento del DB: SON . . . . .	37
5.6.3	Campionamento e frontiera negativa: Toivonen . . . . .	37
5.7	Perché andare oltre Apriori . . . . .	37
5.8	FP-Growth: idea di base . . . . .	37

5.8.1	Costruzione dell'FP-tree . . . . .	37
5.8.2	Esempio di FP-Growth . . . . .	38
5.9	Confronto: FP-Growth vs Apriori . . . . .	39
<b>6</b>	<b>Clustering</b>	<b>41</b>
6.1	Concetti generali . . . . .	41
6.1.1	Spazi metrici e funzioni distanza . . . . .	41
6.1.2	Tassonomia degli algoritmi . . . . .	42
6.1.3	Alta dimensionalità: equidistanza e ortogonalità . . . . .	42
6.2	Clustering gerarchico . . . . .	43
6.2.1	Distanza tra cluster ( <i>linkage</i> ) . . . . .	43
6.2.2	Dendrogramma e criteri di stop . . . . .	43
6.2.3	Altri criteri di combinazione . . . . .	43
6.2.4	Versioni divisive . . . . .	44
6.2.5	Complessità e ottimizzazioni . . . . .	44
6.3	Clustering partizionale: k-means . . . . .	45
6.3.1	Algoritmo base . . . . .	45
6.3.2	Inizializzazione . . . . .	45
6.3.3	Funzione obiettivo e arresto . . . . .	45
6.3.4	Scelta del numero di cluster <b>k</b> . . . . .	46
6.3.5	Complessità computazionale . . . . .	47
6.4	Clustering per densità . . . . .	47
6.4.1	DBSCAN . . . . .	47
6.4.2	OPTICS . . . . .	48
6.4.3	HDBSCAN . . . . .	51
<b>7</b>	<b>Classificazione</b>	<b>55</b>
7.1	Introduzione . . . . .	55
7.1.1	Schema generale di un classificatore . . . . .	55
7.1.2	Requisiti desiderabili . . . . .	55
7.2	Alberi decisionali . . . . .	56
7.2.1	Classificazione tramite albero . . . . .	56
7.2.2	Costruzione top-down . . . . .	56
7.2.3	Splitting degli attributi . . . . .	57
7.2.4	Scelta dell'attributo e strategia greedy . . . . .	57
7.3	Misure di goodness . . . . .	57
7.3.1	Information Gain (ID3) . . . . .	57
7.3.2	Gain Ratio (C4.5) . . . . .	58
7.3.3	Gini Index (CART) . . . . .	59
7.3.4	Pruning degli alberi . . . . .	59
7.4	Classificatori generativi . . . . .	61
7.4.1	Teorema di Bayes e regola di decisione . . . . .	62
7.4.2	Naive Bayes . . . . .	62

7.4.3	Reti Bayesiane . . . . .	63
7.5	Classificatori discriminativi . . . . .	63
7.5.1	Classificazione lineare e non lineare . . . . .	64
7.5.2	Perceptron . . . . .	64
7.5.3	Support Vector Machines (SVM) . . . . .	66
7.6	Apprendimento Lazy . . . . .	70
7.6.1	K-Nearest Neighbor . . . . .	70
7.7	Ensemble Learning . . . . .	71
7.7.1	Bagging . . . . .	71
7.7.2	Boosting . . . . .	72
7.7.3	Adaboost . . . . .	73
7.8	Validazione di un classificatore . . . . .	75
7.8.1	Matrice di confusione . . . . .	75
7.8.2	Soglia discriminativa in un classificatore binario . . . . .	76
7.8.3	Curva ROC . . . . .	76
7.8.4	Curva di Precision-Recall . . . . .	76
7.8.5	Validazione di un classificatore . . . . .	77
<b>8</b>	<b>Cenni di Regressione</b>	<b>79</b>
8.1	Regressione lineare semplice . . . . .	79
8.1.1	Formulazione del modello . . . . .	79
8.1.2	Stima dei parametri . . . . .	80
8.1.3	Interpretazione geometrica . . . . .	80
8.2	Regressione lineare multipla . . . . .	80
8.2.1	Formulazione del modello . . . . .	80
8.2.2	Stima dei parametri . . . . .	81
8.2.3	Interpretazione geometrica . . . . .	81
8.3	Regressione non lineare . . . . .	81
8.4	Regressione logistica . . . . .	81
8.4.1	Regressione logistica binaria semplice . . . . .	82
<b>9</b>	<b>Subgraph Matching</b>	<b>83</b>
9.1	Isomorfismo di Grafi . . . . .	83
9.1.1	Automorfismo . . . . .	83
9.2	Operazione di subgraph matching . . . . .	84
9.3	Complessità computazionale . . . . .	85
9.4	Algoritmi di subgraph matching . . . . .	85
9.4.1	Soluzione Bruteforce . . . . .	85
9.4.2	Algoritmo di Ullmann . . . . .	86
9.4.3	Algoritmo VF . . . . .	87
9.4.4	Algoritmo VF2 . . . . .	90
9.4.5	Algoritmo RI . . . . .	91
9.4.6	RI-DS . . . . .	93

9.5	Subgraph Matching in Database di Grafi . . . . .	93
9.5.1	Indicizzazione . . . . .	94
9.6	Features dei grafi . . . . .	94
9.6.1	Schema di subgraph matching in database di grafi . . . . .	95
9.6.2	Indicizzazione inversa . . . . .	96
9.6.3	Algoritmo SING . . . . .	96
<b>10</b>	<b>Subgraph Matching di Grafi Frequenti</b>	<b>99</b>
10.1	Algoritmo FSG . . . . .	100
10.1.1	Regola Apriori per sotto-grafi . . . . .	100
10.1.2	Join tra sotto-grafi . . . . .	101
10.1.3	Procedura dell'algoritmo . . . . .	104
10.1.4	Generazione dei candidati . . . . .	104
10.1.5	Stringa di adiacenza . . . . .	105
10.1.6	Forma canonica . . . . .	105
10.1.7	Verifica della regola Apriori . . . . .	106
10.2	Algoritmo gSpan . . . . .	107
10.2.1	Visita DFS . . . . .	107
10.2.2	Codifica DFS . . . . .	107
10.2.3	Generazione dei candidati . . . . .	110
<b>11</b>	<b>Elementi di Reti neurali</b>	<b>113</b>
11.1	Strati . . . . .	114
11.1.1	Connessioni tra layer . . . . .	115
11.2	Progettare una rete neurale . . . . .	115
11.3	Funzioni di attivazione . . . . .	115
11.3.1	Funzione step . . . . .	116
11.3.2	Funzione logistica . . . . .	117
11.3.3	Tangente iperbolica . . . . .	118
11.3.4	Funzione softmax . . . . .	118
11.3.5	ReLU: Rectified Linear Unit . . . . .	119
11.4	Funzioni Loss . . . . .	120
11.4.1	Regression Loss . . . . .	120
11.4.2	Classification Loss . . . . .	122
11.5	Training di una rete neurale . . . . .	123
11.5.1	Ottimizzazione dei pesi . . . . .	123
11.5.2	Metodo di discesa del gradiente . . . . .	123
11.5.3	Esempio di computazione . . . . .	125
11.5.4	Backpropagation . . . . .	126
11.6	Tecniche di Regolarizzazione . . . . .	129
11.6.1	Regolarizzazione L1 e L2 . . . . .	129
11.6.2	Dropout . . . . .	130
11.6.3	Early Stopping . . . . .	130

11.6.4 Aumento del training set . . . . .	130
11.7 Tipi di reti neurali . . . . .	131
11.7.1 Feed-Forward Networks . . . . .	131
11.7.2 Reti Neurali Convoluzionali . . . . .	131
11.7.3 Rete neurali di grafi . . . . .	133
11.7.4 Reti Neurali Ricorrenti . . . . .	134
11.7.5 LSTM: Long Short-Term Memory . . . . .	135
11.7.6 Autoencoder . . . . .	138
11.7.7 Variational Autoencoder . . . . .	138
<b>12 Introduzione a Transformer e LLM</b>	<b>141</b>
<b>III Approfondimenti</b>	<b>143</b>

# **Parte I**

## **Concetti Preliminari**



## Capitolo 1

# Prerequisiti Matematici Essenziali [WIP]



# Capitolo 2

## Grafi

I grafi risolvono uno dei problemi più comuni in informatica: la rappresentazione e l'analisi delle relazioni tra oggetti. Molto spesso, infatti, i dati non sono semplicemente una lista di elementi ma possono essere visti come un insieme di connessioni tra degli elementi: basti pensare a una rete sociale, dove gli utenti sono collegati tra loro da amicizie, oppure a una mappa stradale, dove le città sono collegate da strade. In questi casi, i grafi forniscono un modo efficace per rappresentare e analizzare queste relazioni.

### 2.1 Definizione formale

Un graffo è una coppia ordinata  $G = (V, E)$ , dove:

- $V$  è un insieme non vuoto di vertici (o nodi).
- $E$  è un insieme di archi (o spigoli), che sono coppie ordinate o non ordinate di vertici.

Gli archi possono essere diretti (nel caso di grafi orientati) o non diretti (nel caso di grafi non orientati).

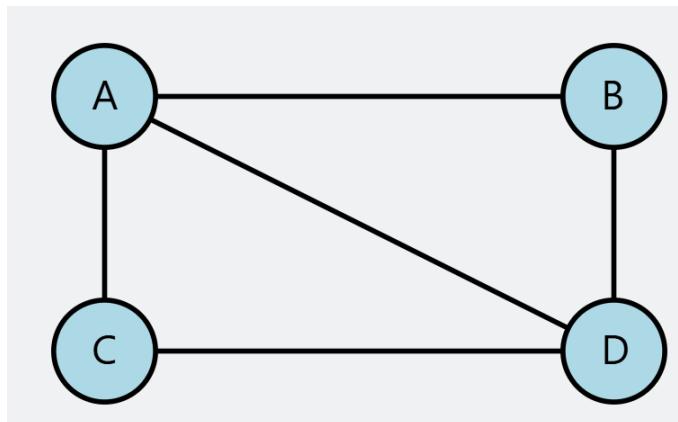


Figura 2.1: Grafo non orientato con quattro vertici  $V = \{A, B, C, D\}$  e archi  $E = \{(A, B), (A, C), (A, D), (B, D), (C, D)\}$ . In figura: A è collegato a B (in alto), a C (sinistra) e a D (diagonale); B è collegato a D (destra); C è collegato a D (in basso).

### 2.1.1 Network science

La scienza che studia i grafi, o reti in questo caso, è la **network science**. È una disciplina interdisciplinare che combina elementi di matematica, informatica, fisica e sociologia per analizzare e comprendere le strutture complesse delle reti.

Un esempio recente di Network science è il *Covid*: Gli epidemiologi hanno utilizzato modelli basati su grafi per tracciare la diffusione del virus, identificare i nodi critici (come le persone più connesse) e prevedere l'impatto delle misure di contenimento.

## 2.2 Grafi diretti e indiretti

I grafi possono essere classificati in due categorie principali: grafi diretti (o orientati) e grafi indiretti (o non orientati):

**Grafo diretto.** In un grafo diretto, gli archi hanno una direzione specifica, rappresentata da una freccia. Questo significa che la relazione tra due vertici è unidirezionale. Ad esempio, in un grafo che rappresenta le relazioni di follower su un social media, se l'utente A segue l'utente B, c'è un arco diretto da A a B, ma non necessariamente da B ad A.

**Grafo indiretto.** In un grafo indiretto, gli archi non hanno una direzione specifica. Questo significa che la relazione tra due vertici è bidirezionale, ovvero per ogni arco  $(a, b)$  esiste anche l'arco  $(b, a)$ . Ad esempio, in un grafo che rappresenta le amicizie in un social media, se l'utente A è amico dell'utente B, c'è un arco non diretto tra A e B.

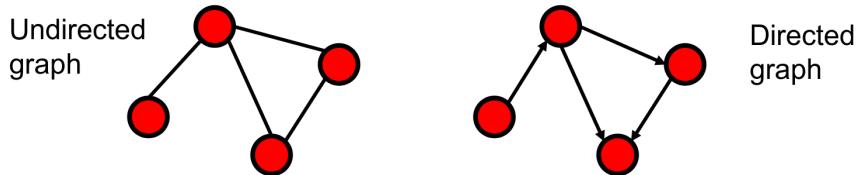


Figura 2.2: Confronto tra grafo non orientato (a sinistra) e grafo orientato (a destra). Nel grafo orientato, gli archi hanno una direzione indicata da frecce, mentre nel grafo non orientato rappresentano relazioni bidirezionali.

**Come capire che tipologia usare.** La scelta tra grafo diretto e indiretto dipende dalla natura delle relazioni che si desidera rappresentare. Se le relazioni sono unidirezionali, come nel caso dei follower sui social media, è appropriato utilizzare un grafo diretto. Se le relazioni sono bidirezionali, come nel caso delle amicizie, è più adatto utilizzare un grafo indiretto.

## 2.3 Grafi pesati e grafi etichettati

Oltre alla distinzione tra grafi diretti e indiretti, i grafi possono essere ulteriormente classificati in grafi pesati e grafi etichettati:

**Grafo pesato.** In un grafo pesato, ogni arco ha un peso associato, che rappresenta il costo, la distanza o qualsiasi altra misura quantitativa tra i vertici collegati. Ad esempio, in un grafo che rappresenta una rete stradale, il peso degli archi può rappresentare la distanza tra le città.

**Grafo etichettato.** In un grafo etichettato, i vertici e/o gli archi hanno etichette o nomi associati, che forniscono informazioni aggiuntive sui nodi o sulle relazioni. Ad esempio, in un grafo che rappresenta una rete sociale, i vertici possono essere etichettati con i nomi degli utenti.

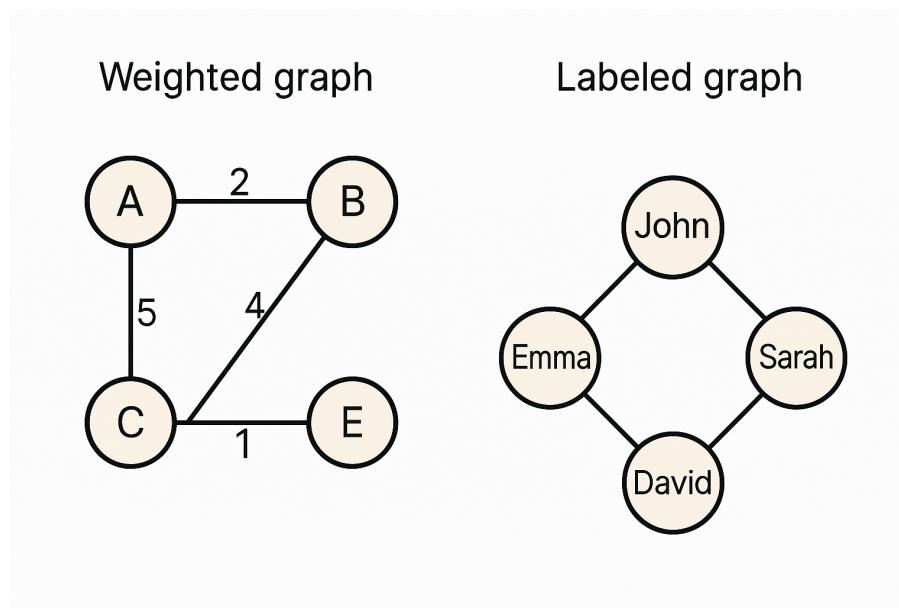


Figura 2.3: Esempio di grafo pesato (a sinistra) ed etichettato (a destra). Nel grafo pesato, i numeri sugli archi rappresentano i pesi associati a ciascun arco. Nel grafo etichettato, i vertici sono etichettati con nomi specifici.

## 2.4 Gradi dei vertici

Il grado di un vertice in un grafo rappresenta il numero di archi che sono collegati a quel vertice. In un grafo indiretto, il grado di un vertice  $v$  è semplicemente il numero di archi incidenti su  $v$ .

Nel caso particolare di un **grafo diretto** si fa distinzione tra:

- **Grado entrante** (in-degree): il numero di archi che arrivano al vertice  $v$ .
- **Grado uscente** (out-degree): il numero di archi che partono dal vertice  $v$ .

Poi si parla di **grado totale** di un vertice  $v$  come la somma del grado entrante e del grado uscente.

Si può parlare anche di **grado medio** di un grafo, che rappresenta la media dei gradi di tutti i vertici nel grafo. In un grafo indiretto, il grado medio  $\bar{k}$  può essere calcolato utilizzando la formula:

$$\bar{k} = \frac{|E|}{|V|}$$

dove  $|E|$  è il numero totale di archi e  $|V|$  è il numero totale di vertici nel grafo.

#### 2.4.1 Distribuzione dei gradi

La distribuzione dei gradi di un grafo descrive come i gradi dei vertici sono distribuiti all'interno del grafo.

Questa è una distribuzione di probabilità  $P$  dove  $p_k$  è la probabilità che un vertice scelto a caso abbia grado  $k$ .

In una rete reale,  $p_k$  si ottiene dividendo il numero  $N_k$  di nodi con grado  $k$  per il numero totale  $N$  di nodi nella rete:

$$p_k = \frac{N_k}{N}$$

### 2.5 Grafi bipartiti

Un grafo bipartito è un tipo speciale di grafo in cui i vertici possono essere divisi in due insiemi disgiunti  $U$  e  $V$  tali che ogni arco collega un vertice in  $U$  a un vertice in  $V$ . Non ci sono archi che collegano vertici all'interno dello stesso insieme.

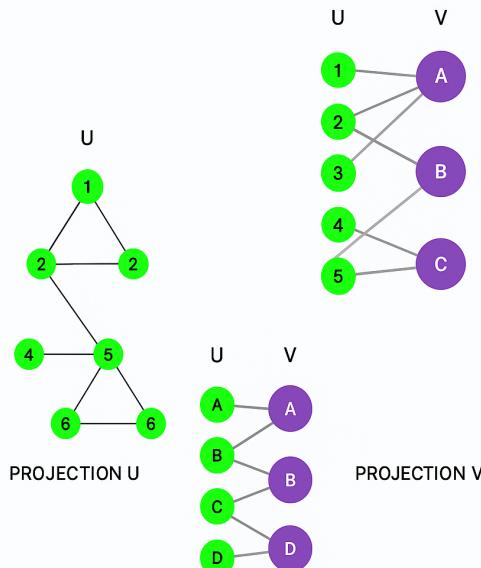


Figura 2.4: Esempio di grafo bipartito con partizioni  $U$  (nodi verdi) e  $V$  (nodi viola): gli archi collegano solo vertici appartenenti a insiemi diversi. In basso sono mostrati i grafi proiettati: *projection U* (a sinistra), dove due vertici di  $U$  sono adiacenti se condividono almeno un vicino in  $V$ , e *projection V* (a destra), definita simmetricamente.

Da un *grafo bipartito* si possono derivare due *grafi proiettati*, che sono grafi non bipartiti ottenuti collegando i vertici di uno degli insiemi  $U$  o  $V$  se condividono un vicino nell'altro insieme.

**Generalizzazione.** I grafi bipartiti possono essere generalizzati a *grafi multipartiti*, dove i vertici sono divisi in più di due insiemi disgiunti, e gli archi collegano solo vertici appartenenti a insiemi diversi.

## 2.6 Grafo completo vs Grafo regolare

Un grafo completo è un grafo in cui ogni coppia di vertici distinti è collegata da un arco. In altre parole, in un grafo completo con  $n$  vertici, ogni vertice ha un arco che lo collega a tutti gli altri  $n - 1$  vertici. Un grafo completo con  $n$  vertici è denotato come  $K_n$ .

Un grafo regolare è un grafo in cui ogni vertice ha lo stesso grado. In altre parole, in un grafo regolare con  $n$  vertici, ogni vertice ha esattamente  $k$  archi collegati ad esso, dove  $k$  è un numero fisso. Un grafo regolare con  $n$  vertici e grado  $k$  è denotato come  $R(n, k)$ .

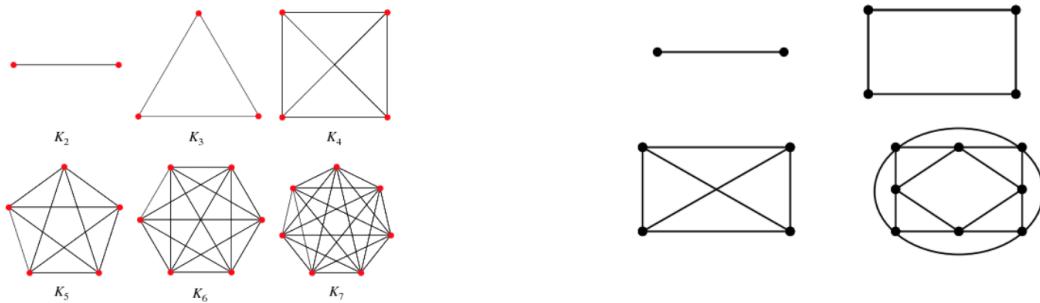


Figura 2.5: A sinistra: esempi di grafi completi  $K_n$  per  $n = 2, \dots, 7$ , in cui ogni coppia di vertici è adiacente. A destra: esempi di grafi  $k$ -regolari (da  $k = 1$  a  $k = 4$ ), in cui ogni vertice ha lo stesso grado  $k$ .

## 2.7 Cammini tra due nodi

Si definisce **cammino** tra due nodi  $u$  e  $v$  in un grafo come una sequenza di vertici e archi che collega  $u$  a  $v$ . Un cammino può essere rappresentato come una sequenza di vertici ( $u = v_0, v_1, v_2, \dots, v_k = v$ ) tale che ogni coppia di vertici consecutivi  $(v_i, v_{i+1})$  è collegata da un arco nel grafo.

### 2.7.1 Cammino minimo

Si definisce **cammino minimo** tra due nodi  $u$  e  $v$  come il cammino che collega  $u$  a  $v$  con il minor numero di archi possibile. In un grafo pesato, il cammino minimo può essere definito come il cammino che minimizza la somma dei pesi degli archi attraversati. Nella figura

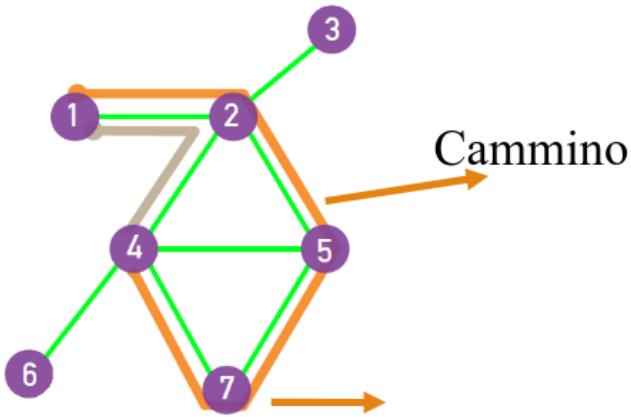


Figura 2.6: Esempio di cammino in un grafo: gli archi verdi definiscono la struttura, mentre in arancione è evidenziato un *cammino* che attraversa i vertici  $1 \rightarrow 2 \rightarrow 5 \rightarrow 7$ .

2.6, il cammino minimo tra i nodi 1 e 7 è  $1 \rightarrow 2 \rightarrow 5 \rightarrow 7$ , che attraversa tre archi, quindi  $d(1, 7) = 3$ .

### 2.7.2 Diametro

Il **diametro** di un grafo è la massima distanza tra tutte le coppie di vertici nel grafo. In altre parole, è la lunghezza del cammino minimo più lungo tra qualsiasi coppia di vertici nel grafo. Il diametro fornisce una misura della "grandezza" del grafo in termini di distanza tra i suoi vertici. Nel caso della figura 2.6, il diametro del grafo è 4, che corrisponde alla distanza massima tra le coppie di nodi (1, 6) e (3, 6).

### 2.7.3 Ciclo

Un particolare tipo di cammino è il **ciclo**, che inizia e termina nello stesso vertice senza ripetere alcun altro vertice lungo il percorso. Un ciclo può essere rappresentato come una sequenza di vertici  $(v_0, v_1, v_2, \dots, v_k, v_0)$  tale che ogni coppia di vertici consecutivi  $(v_i, v_{i+1})$  è collegata da un arco nel grafo, e  $v_0 = v_k$ . Nell'immagine 2.6, un esempio di ciclo è  $2 \rightarrow 5 \rightarrow 4 \rightarrow 2$ .

**Cappio.** Un particolare tipo di ciclo è il **cappio**, che è un arco che collega un vertice a se stesso. In altre parole, un cappio è un ciclo di lunghezza 1.

## 2.8 Connettività

Due nodi  $i, j$  di un grafo si dicono **connessi** se esiste almeno un cammino che li collega. Un grafo si dice **connesso** se ogni coppia di nodi del grafo è connessa, al contrario si dice **disconnesso**.

Un grafo disconnesso  $G$  risulta formato dall'unione di più sottografi connessi, detti **componenti connesse** di  $G$ .

### 2.8.1 Connattività forte e debole

In un grafo diretto, si distinguono due tipi di connattività:

**Connattività forte** Due nodi  $u$  e  $v$  sono fortemente connessi se esiste un cammino diretto da  $u$  a  $v$  e un cammino diretto da  $v$  a  $u$ . Un grafo diretto è fortemente connesso se ogni coppia di nodi è fortemente connessa.

**Connattività debole** Due nodi  $u$  e  $v$  sono debolmente connessi se esiste un cammino diretto da  $u$  a  $v$  o un cammino diretto da  $v$  a  $u$ . Un grafo diretto è debolmente connesso se ogni coppia di nodi è debolmente connessa.

## 2.9 Coefficiente di Clustering

Il **coefficiente di clustering**  $C_n$  di un nodo  $n$  è una misura di quanto gli adiacenti di  $n$  siano **connessi tra loro**. Misura la *densità locale* della rete attorno a  $n$ : più il vicinato di  $n$  è densamente connesso, più alto è il coefficiente.

Formalmente, se  $k_n$  è il grado di  $n$  (numero di vicini) e  $L_n$  è il numero di archi effettivamente presenti tra i  $k_n$  vicini, il coefficiente di clustering locale si definisce come:

$$C_n = \frac{2L_n}{k_n(k_n - 1)}$$

La formula normalizza il numero di archi esistenti rispetto al numero massimo possibile di archi tra i  $k_n$  vicini, così che  $C_n \in [0, 1]$ .

Esempi visuali: il primo caso mostra un vicinato completamente connesso ( $C_i = 1$ ), il secondo un vicinato parzialmente connesso ( $C_i = 1/2$ ) e il terzo un vicinato senza archi tra vicini ( $C_i = 0$ ).

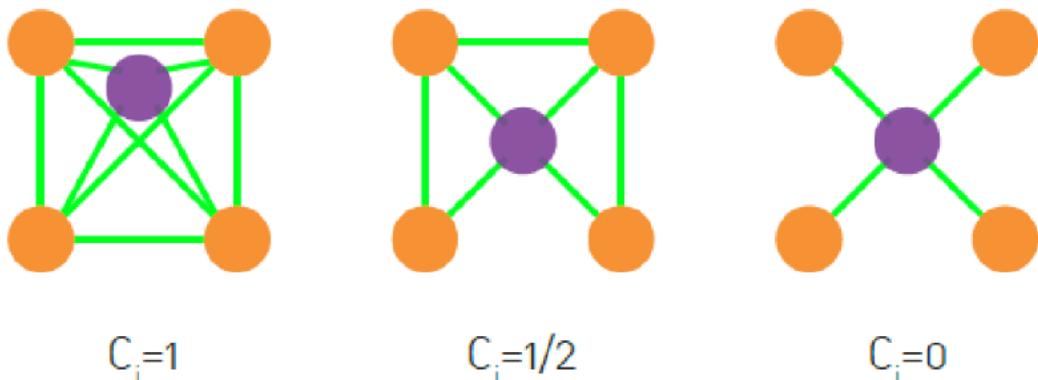


Figura 2.7: Esempio del coefficiente di clustering locale per un nodo centrale (viola) con quattro vicini (arancioni). A sinistra il vicinato è completamente connesso, quindi  $C_i = 1$ ; al centro solo metà delle possibili connessioni tra i vicini è presente ( $C_i = \frac{1}{2}$ ); a destra non ci sono archi tra i vicini e il coefficiente è nullo ( $C_i = 0$ ).

**Clustering medio.** Il **clustering medio**  $\langle C \rangle$  di un grafo si ottiene calcolando la media aritmetica dei coefficienti di clustering locali di tutti i nodi del grafo:

$$\langle C \rangle = \frac{1}{N} \sum_{n=1}^N C_n$$

dove  $N$  è il numero totale di nodi nel grafo. Il clustering medio fornisce una misura globale della tendenza dei nodi a formare gruppi o comunità all'interno del grafo.

**Coefficiente di clustering globale.** Un'altra misura del clustering in un grafo è il *coefficiente di clustering globale*  $\phi$ , che si basa sul conteggio dei triangoli e delle triplette nel grafo. Un triangolo è una tripla di nodi tutti connessi tra loro, mentre una tripletta è una sequenza di tre nodi collegati da due archi.

Un triangolo però, si può esprimere in 3 modi diversi in base all'orientamento: ad esempio, i nodi  $A, B$  e  $C$  formano un triangolo, ma si possono contare le triplette  $(A, B, C)$ ,  $(B, C, A)$  e  $(C, A, B)$ .

## 2.10 Misure di centralità

La **centralità** di un nodo in un grafo è una misura dell'importanza di un nodo nella rete. Esistono diverse misure di centralità a seconda del criterio di misura.

### 2.10.1 Centralità di grado

La **centralità di grado**  $C_G(v)$  di un nodo  $v$  è definita come il numero di archi incidenti su  $v$ . In un grafo indiretto, la centralità di grado è semplicemente il grado del nodo:

$$C_G(v) = \text{grado}(v)$$

In un grafo diretto, si può distinguere tra centralità di grado entrante e centrale di grado uscente:

$$C_G^{in}(v) = \text{grado entrante}(v)$$

$$C_G^{out}(v) = \text{grado uscente}(v)$$

### 2.10.2 Centralità di vicinanza

La **centralità di vicinanza**  $C_C(v)$  di un nodo  $v$  è definita come il numero di cammini minimi che passano per  $v$  tra tutte le coppie di nodi nel grafo:

$$C_C(v) = \sum_{s \neq v \neq t} \frac{\delta_{ij}(v)}{\delta_{ij}}$$

dove  $\delta_{ij}$  è il numero di cammini minimi tra i nodi  $i$  e  $j$ , e  $\delta_{ij}(v)$  è il numero di tali cammini che passano per  $v$ . Un nodo con alta centralità di vicinanza agisce come un ponte critico nella rete, facilitando la comunicazione tra altri nodi.

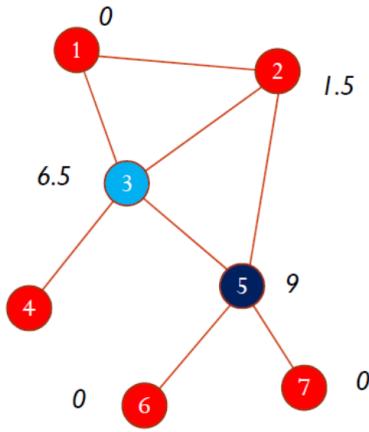


Figura 2.8: Esempio di calcolo della centralità di vicinanza per i nodi in un grafo. I numeri accanto ai nodi indicano le distanze minime da ciascun nodo agli altri nodi nel grafo.

**Esempio (nodo 3).** Consideriamo ora il calcolo della **centralità di vicinanza** per il nodo 3. Questa misura quantifica quante volte il nodo 3 si trova sui cammini minimi tra coppie di altri nodi del grafo.

Nella tabella seguente riportiamo tutte le coppie di nodi  $(i, j)$  che non includono il nodo 3, insieme ai valori di  $\delta_{ij}(3)$ ,  $\delta_{ij}$  e del loro rapporto.

Coppia $(i, j)$	$\delta_{ij}(3)$	$\delta_{ij}$	$\frac{\delta_{ij}(3)}{\delta_{ij}}$
(1, 2)	0	1	0
(1, 4)	1	1	1
(1, 5)	1	2	0.5
(1, 6)	1	2	0.5
(1, 7)	1	2	0.5
(2, 4)	1	1	1
(2, 5)	0	1	0
(2, 6)	0	1	0
(2, 7)	0	1	0
(4, 5)	1	1	1
(4, 6)	1	1	1
(4, 7)	1	1	1
(5, 6)	0	1	0
(5, 7)	0	1	0
(6, 7)	0	1	0
Total	—	—	6.5

La somma dei rapporti fornisce il valore complessivo della centralità di intermediazione per il nodo 3:

$$C_B(3) = 6.5$$

### 2.10.3 Centralità di prossimità

La **centralità di prossimità**  $C_P(v)$  di un nodo  $v$  è definita come l'inverso della somma delle distanze minime da  $v$  a tutti gli altri nodi nel grafo:

$$C_P(v) = \frac{1}{\sum_{u \neq v} d(v, u)}$$

dove  $d(v, u)$  è la distanza minima tra i nodi  $v$  e  $u$ . Un nodo con alta centralità di prossimità è in grado di raggiungere rapidamente tutti gli altri nodi nel grafo.

Il grafo in figura 2.8 può essere riscritto con i valori di centralità di prossimità, come mostrato in figura 2.9.

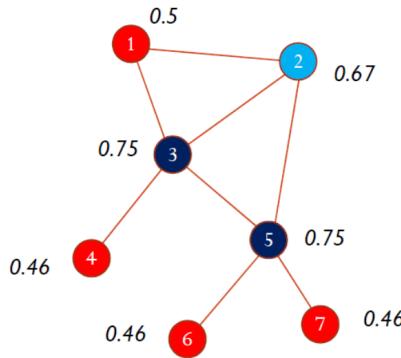


Figura 2.9: Esempio di calcolo della centralità di prossimità per i nodi in un grafo. I numeri accanto ai nodi indicano i valori di centralità di prossimità calcolati in base alle distanze minime da ciascun nodo agli altri nodi nel grafo.

### 2.10.4 Centralità di PageRank

La **centralità di PageRank**  $C_{PR}(v)$  di un nodo  $v$  è una misura dell'importanza di un nodo basata sulla struttura del grafo e sulle connessioni tra i nodi. PageRank è stato originariamente sviluppato da Google per valutare l'importanza delle pagine web, ma può essere applicato a qualsiasi grafo.

La formula di PageRank è data da:

$$C_{PR}(v) = \frac{1-d}{N} + d \sum_{u \in \text{In}(v)} \frac{C_{PR}(u)}{\text{OutDegree}(u)}$$

dove:

- $d$  è il fattore di damping, solitamente impostato a 0.85.
- $N$  è il numero totale di nodi nel grafo.
- $\text{In}(v)$  è l'insieme dei nodi che puntano a  $v$ .
- $\text{OutDegree}(u)$  è il grado uscente del nodo  $u$ .

**Esempio.** Consideriamo una **rete diretta** costituita da quattro nodi  $A, B, C, D$ . Ogni nodo parte con uno *score iniziale* uguale per tutti, pari a 0.25, poiché lo score complessivo della rete è 1. La figura 2.10 mostra la configurazione iniziale del grafo, in cui ogni nodo ha lo stesso valore di PageRank.

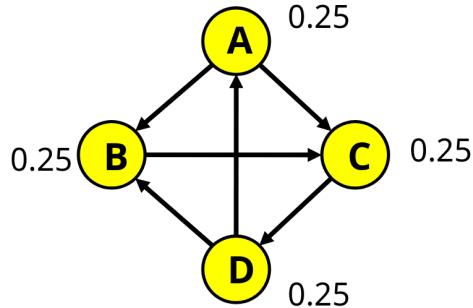


Figura 2.10: Configurazione iniziale della rete diretta con quattro nodi  $A, B, C, D$ , ciascuno con uno score iniziale di 0.25.

In ogni iterazione, ciascun nodo cede il proprio score in **parti uguali** tra i nodi verso i quali ha archi uscenti. Nel primo round le distribuzioni sono le seguenti:

- $A$  cede metà score a  $B$  (0.125) e metà a  $C$  (0.125);
- $B$  cede tutto il proprio score a  $C$  (0.25);
- $C$  cede tutto il proprio score a  $D$  (0.25);
- $D$  cede metà score a  $A$  (0.125) e metà a  $B$  (0.125).

Dopo il primo round si ottengono i seguenti valori di PageRank:  $A = 0.125$ ,  $B = 0.25$ ,  $C = 0.375$ ,  $D = 0.25$

Iterando questo processo di redistribuzione, i valori dei nodi si aggiornano progressivamente fino a raggiungere una situazione di **equilibrio**, in cui i punteggi non variano più in modo significativo:

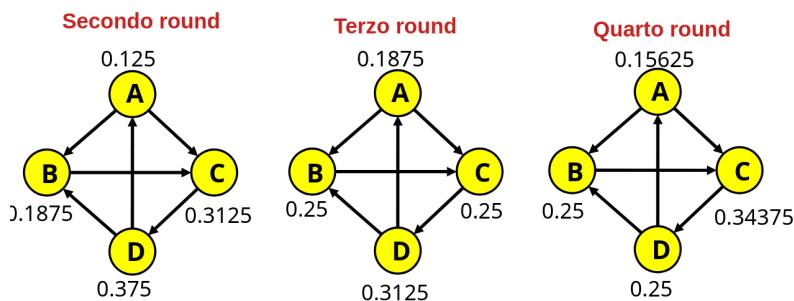


Figura 2.11: Evoluzione dei valori di PageRank per i nodi  $A, B, C, D$  nei round successivi.

Al termine delle iterazioni, i valori di equilibrio rappresentano la **centralità di PageRank** dei nodi, indicando la loro importanza relativa all'interno della rete.

## Riferimenti

I riferimenti di questo capitolo possono essere trovati al capitolo 1 e 2 del libro [1].

## **Parte II**

# **Tecniche di Data Mining**



## Capitolo 3

# Introduzione al Data Mining

### 3.1 Definizione e finalità

Il *Data Mining* consiste nello scoprire *pattern* (modelli e regolarità) interessanti e possibilmente inattesi all'interno di un insieme di dati. Le conoscenze estratte possono essere impiegate per supportare decisioni, formulare previsioni o fungere da base per ulteriori attività (ad es. profilazione di utenti).

- **Data cleaning (pre-processing)**: individuare e rimuovere artefatti e dati fittizi o rumorosi, armonizzare formati, gestire valori mancanti.
- **Visualizzazione**: comunicare in modo efficace i risultati del processo di data mining.

### 3.2 Caratteristiche dei pattern

I pattern da estrarre dovrebbero essere:

- **Validi**: veri (con alta probabilità) anche su dati nuovi non visti.
- **Utili**: capaci di suggerire o guidare azioni concrete.
- **Inattesi**: non banali, non ovvi.
- **Comprensibili**: interpretabili da esseri umani.

### 3.3 Metodi di data mining

Gli algoritmi di data mining si distinguono in:

- **Descrittivi**: mirano a rappresentare e *descrivere* la struttura dei dati (es. clustering, regole di associazione, analisi di similarità).
- **Predittivi**: usano alcune variabili per *predire* valori sconosciuti o futuri (es. classificazione, regressione, sistemi di raccomandazione).

## 3.4 Perché fare data mining

Negli ultimi anni la quantità di dati disponibili è esplosa. Le principali sorgenti includono:

- **Business:** web, e-commerce, transazioni, mercati finanziari, log applicativi.
- **Multimedia:** video, audio, testo, immagini.
- **Scienza:** telerilevamento, medicina, bioinformatica.
- **Società:** news, e-mail, social network, forum.

### 3.4.1 Big Data

I dati moderni sono spesso:

- **Grandi** (*volume*);
- **Ad elevata dimensionalità** (*varietà* di attributi);
- **Complessi** (relazionali, temporali, eterogenei).

La sola disponibilità di molti dati non si traduce automaticamente in conoscenza: servono metodi e strumenti per analizzarli in modo efficace.

### 3.4.2 Dai dati alla conoscenza e alle comunità coinvolte

L'enorme quantità di dati non diventa automaticamente conoscenza: occorre trasformarla con tecniche appropriate. Il data mining è al crocevia di più comunità scientifiche (apprendimento automatico, pattern recognition, visualizzazione, algoritmi, ...).

## 3.5 Limiti e insidie del data mining

Un'idea intuitiva (ma pericolosa) è: “raccogliamo quanti più dati possibile e troveremo pattern affidabili”. In realtà, al crescere della dimensione aumenta anche la probabilità di osservare regolarità *spurie*, cioè non realmente significative. Consideriamo un caso di studio.

### 3.5.1 Caso di studio: Total Information Awareness (TIA)

Dopo gli attentati dell'11 settembre 2001, il Dipartimento della Difesa degli Stati Uniti propose il programma *Total Information Awareness* (TIA), volto a raccogliere in modo massivo informazioni su persone (ricevute di pagamento, spostamenti, ecc.) per prevenire attacchi terroristici. Al di là delle criticità etiche e di privacy, un rischio metodologico è la generazione di moltissimi *falsi positivi*: attività apparentemente anomale ma statisticamente spiegabili.

### 3.5.2 Esempio: co-presenza in hotel come criterio di sospetto

Vogliamo identificare potenziali coppie di malfattori assumendo che essi si riuniscano periodicamente nello stesso hotel. Sui dati osservati cerchiamo tutte le coppie di persone che, in *due* giorni distinti, risultano nello *stesso* hotel.

#### Dati di partenza.

- Numero di persone tracciate:  $N = 10^9$ .
- Orizzonte temporale:  $D = 1000$  giorni.
- Numero di hotel:  $H = 10^5$ .
- Capacità per hotel e per giorno:  $C = 100$  persone.

**Ipotesi nulla (random).** Ogni persona, in ciascun giorno, sceglie in modo casuale e indipendente se (e dove) soggiornare; in particolare, per un dato hotel in un dato giorno la probabilità che una persona vi alloggi è

$$P(\text{persona in un hotel specifico in un giorno}) = \frac{C}{N} = \frac{100}{10^9} = 10^{-7}.$$

#### Calcoli numerici.

1. **Stesso hotel in un giorno fissato.** Per due persone specifiche  $p, q$ , la probabilità di trovarle nello stesso hotel in un *giorno specifico* è

$$P_1 = H \cdot \left( \frac{C}{N} \right)^2 = 10^5 \cdot (10^{-7})^2 = 10^{-9}.$$

2. **Due giorni distinti non specificati.** Le coppie di giorni distinti sono  $\binom{D}{2} = \frac{D(D-1)}{2} \approx \frac{1000 \cdot 999}{2} \approx 5 \cdot 10^5$ . Supponendo indipendenza tra i giorni, la probabilità che  $p, q$  si trovino nello stesso hotel in *entrambi* i due giorni è

$$P_2 = \binom{D}{2} \cdot P_1^2 = \left( 5 \cdot 10^5 \right) \cdot (10^{-9})^2 = 5 \cdot 10^{-13}.$$

3. **Numero atteso di coppie sospette.** Le coppie distinte di persone sono  $\binom{N}{2} \approx \frac{10^9(10^9-1)}{2} \approx 5 \cdot 10^{17}$ . Il numero atteso di coppie candidate è dunque

$$\mathbb{E}[\#\text{coppie}] = P_2 \cdot \binom{N}{2} \approx \left( 5 \cdot 10^{-13} \right) \cdot \left( 5 \cdot 10^{17} \right) = 2,5 \cdot 10^5.$$

**Considerazioni.** Verificare manualmente  $\sim 250,000$  coppie è impraticabile, specie a fronte di un numero reale di coppie colpevoli verosimilmente molto più basso. L'esempio mostra come, su dati enormi, criteri apparentemente sensati possano generare moltissimi falsi positivi *anche in assenza di segnale*.

### 3.6 Principio di Bonferroni e test multipli

**Principio di Bonferroni.** Se il numero atteso di occorrenze dell’evento cercato (sotto l’ipotesi di casualità dei dati) è significativamente *maggior*e del numero di istanze che ci si aspetta di trovare nella realtà, allora qualsiasi “pattern” osservato è più verosimilmente un *artefatto* che non un’evidenza.

**Interpretazione operativa.** Quando formuliamo molte ipotesi/ricerche sui dati (*multiple testing*), è necessario *correggere* il livello di significatività per tenere sotto controllo i falsi positivi. Una correzione conservativa è la *correzione di Bonferroni*: se eseguiamo  $m$  test, imponiamo che il  $p$ -value di ciascun test sia  $< \alpha/m$  per mantenere il Family-Wise Error Rate al di sotto di  $\alpha$ .

**Quando applicarlo.** In scenari esplorativi su grandi basi dati (come nel caso sopra), prima di agire sui “pattern” trovati occorre verificare che il loro numero sia compatibile con quanto ci si aspetterebbe per puro caso. In caso contrario, i pattern vanno trattati con sospetto e sottoposti a verifica indipendente (es. dati di conferma, A/B test, validazione su hold-out).

## Capitolo 4

# Preprocessing

Il preprocessing dei dati è una fase cruciale nel flusso di lavoro di qualsiasi progetto che sfrutta un dataset. Questa fase include una serie di operazioni volte a migliorare la qualità dei dati, rendendoli più adatti per l'analisi e la modellazione. In questo capitolo, esploreremo le principali tecniche di preprocessing utilizzate nel contesto del dataset in esame.

Generalmente una fase di preprocessing dei dati dipende dalle caratteristiche specifiche del dataset e dagli obiettivi dell'analisi. Tuttavia, alcune operazioni comuni includono:

**Estrazione di feature** - Questa parte del preprocessing coinvolge l'identificazione e l'estrazione delle caratteristiche più rilevanti dai dati grezzi. Queste feature possono essere utilizzate come input per modelli di machine learning o altre analisi statistiche.

**Portabilità dei dati** - Assicurarsi che i dati siano in un formato compatibile con gli strumenti e le librerie utilizzate per l'analisi. Questo può includere la conversione di formati di file, la normalizzazione delle strutture dei dati e l'adeguamento delle codifiche.

**Cleaning dei dati** - Questa fase include la gestione dei valori mancanti, la rimozione di outlier e la correzione di errori nei dati. Un dataset pulito è essenziale per garantire risultati affidabili nelle fasi successive dell'analisi.

**Riduzione dei dati** - In alcuni casi, può essere utile ridurre la dimensionalità del dataset o selezionare un sottoinsieme di dati per migliorare l'efficienza computazionale e ridurre il rumore nei dati.

### 4.1 Estrazione di feature

L'estrazione di feature consiste nel creare un **set di feature** più adatte al problema da risolvere rispetto ai dati grezzi. Ipotizziamo di avere un dataset di immagini, le feature grezze generalmente sono i pixel dell'immagine, ma sono poco utili per un'analisi più approfondita. In questo caso, potremmo estrarre feature come bordi, texture o forme presenti nell'immagine, che possono essere più informative per un modello di machine learning.

### 4.1.1 Tecniche di estrazione di feature

Esistono diverse tecniche per l'estrazione di feature, tra cui:

- **Feature basate su statistiche** - Calcolo di statistiche descrittive come media, varianza, skewness e kurtosis.
- **Feature basate su trasformazioni** - Utilizzo di trasformazioni matematiche come la Trasformata di Fourier o la Trasformata Wavelet per estrarre informazioni frequenziali.
- **Feature basate su modelli** - Applicazione di modelli predefiniti per estrarre feature, come l'uso di reti neurali convoluzionali per l'estrazione di feature da immagini.

## 4.2 Portabilità dei dati

In questo caso il problema diventa quello di convertire i dati in un formato che sia facilmente utilizzabile dagli strumenti di analisi. Ad esempio, se i dati sono in un formato proprietario, potrebbe essere necessario convertirli in un formato standard come CSV o JSON. Inoltre i dati sono generalmente salvati in modo **eterogeneo** (valori numerici oppure categorici) e quindi è necessario uniformarli per poterli utilizzare in modo efficace.

### 4.2.1 Da dati numerici a categorici

La conversione da dati *numerici* a *categorici* è detta **discretizzazione**. Questa tecnica consiste nel suddividere l'intervallo dei valori numerici in un numero finito di intervalli, assegnando a ciascun intervallo una categoria specifica. Ad esempio, i valori di età possono essere suddivisi in categorie come "giovane", "adulto" e "anziano".

**Equi-width ranges.** In questo caso, l'intervallo dei valori numerici viene suddiviso in intervalli di larghezza uguale. Ad esempio, se i valori variano da 0 a 100 e si desidera creare 5 categorie, ogni intervallo avrà una larghezza di 20 (0 – 19, 20 – 39, 40 – 59, 60 – 79, 80 – 100).

**Equi-log ranges.** Questa tecnica invece, si costruisce ogni intervallo  $[a, b]$  in modo tale che la scala logaritmica sia uniforme, ovvero  $\log_x a - \log_x b$  è costante e ha sempre lo stesso valore per ogni intervallo. Questa tecnica è particolarmente utile quando i dati coprono un ampio intervallo di valori e si desidera preservare le proporzioni relative tra i valori. Ad esempio, se i valori variano da 1 a 1000 e si desidera creare 3 categorie, gli intervalli potrebbero essere  $[1, 10]$ ,  $[11, 100]$  e  $[101, 1000]$  (per un logaritmo in base 10).

**Equi-depth ranges.** In questo caso, gli intervalli vengono creati in modo tale che ogni intervallo contenga lo stesso numero di istanze. Ad esempio, se si hanno 100 istanze e si desidera creare 5 categorie, ogni intervallo conterrà 20 istanze. Questa tecnica è utile quando si desidera bilanciare la distribuzione delle categorie.

### 4.2.2 Da dati categorici a numerici

La conversione da dati *categorici* a *numerici* è detta **codifica**. Questa tecnica consiste nell'assegnare un valore numerico a ciascuna categoria. Ad esempio, le categorie "rosso", "verde" e "blu" possono essere codificate come 1, 2 e 3 rispettivamente. Questo si fa perché alcuni modelli di machine learning o algoritmi di analisi richiedono input numerici.

**One-hot encoding.** Questo schema di codifica consiste nel creare una nuova variabile binaria per ogni categoria. Quindi si ottiene un vettore di lunghezza pari al numero di categorie, in cui solo la posizione corrispondente alla categoria attuale è impostata a 1, mentre tutte le altre sono impostate a 0. Ad esempio, per le categorie "rosso", "verde" e "blu", la codifica one-hot sarebbe:

- Rosso: [1, 0, 0]
- Verde: [0, 1, 0]
- Blu: [0, 0, 1]

### 4.2.3 Da testo a dati numerici

La conversione da *testo* a *dati numerici* è detta **vettorizzazione**. Questa tecnica consiste nel rappresentare il testo come un vettore di numeri, in modo che possa essere utilizzato come input per modelli di machine learning o altre analisi statistiche. Ad esempio, una frase come "Il gatto è sul tappeto" può essere rappresentata come un vettore di numeri che rappresentano la frequenza delle parole nella frase.

## 4.3 Cleaning dei dati

Il cleaning dei dati è una fase essenziale del preprocessing che mira a migliorare la qualità dei dati rimuovendo errori, gestendo valori mancanti e trattando outlier. Un dataset pulito è fondamentale per garantire risultati affidabili nelle fasi successive dell'analisi.

### 4.3.1 Gestione dei valori mancanti

I valori mancanti possono verificarsi per vari motivi, come errori di raccolta dati o problemi di trasmissione. Ci sono diverse strategie per gestire i valori mancanti:

**Rimozione delle istanze** - Eliminare le righe o le colonne che contengono valori mancanti.

Questa strategia è semplice ma può portare alla perdita di informazioni importanti, tuttavia funziona bene quando la quantità di dati mancanti è minima.

**Stima dei valori mancanti** - Utilizzare tecniche di imputazione<sup>1</sup> per stimare i valori mancanti basandosi sui dati disponibili. Alcuni metodi comuni includono la media, la mediana o la moda per variabili numeriche, e il valore più frequente per variabili

---

<sup>1</sup>Per imputazione si intende la sostituzione dei valori mancanti con valori stimati basati su altre informazioni presenti nel dataset.

categoriche. Metodi più avanzati includono l'uso di modelli di regressione o algoritmi di machine learning per prevedere i valori mancanti.

### 4.3.2 Gestione dei valori errati

I valori errati possono derivare da errori di inserimento dati, misurazioni imprecise o problemi di trasmissione. Per gestire i valori errati, è possibile:

**Rilevamento di inconsistenze** - Identificare valori che non rientrano nell'intervallo previsto o che sono logicamente incoerenti con altri dati. Ad esempio, un'età negativa o una data di nascita futura.

**Correzione dei valori errati** - Sostituire i valori errati con valori stimati o medi basati su altre informazioni nel dataset. In alcuni casi, potrebbe essere necessario consultare esperti del dominio per determinare il valore corretto. Per esempio, se abbiamo dati di ristoranti negli Stati Uniti e troviamo un ristorante nella città "Roma", è probabile che si tratti di un errore.

Quando si parla di valori errati si parla anche di **outlier**, ovvero valori che si discostano significativamente dalla maggior parte dei dati. Gli outlier possono essere il risultato di errori di misurazione o possono rappresentare fenomeni rari ma validi. Per esempio, in un dataset di altezze umane, un valore di 250 cm potrebbe essere considerato un outlier. Esistono diversi metodi per eliminare gli outlier, come l'uso di tecniche statistiche.

**Quantili.** Un quantile è un valore che divide un insieme di dati ordinati in intervalli con una certa percentuale di dati in ciascun intervallo. Ad esempio, il primo quartile (Q1) è il valore che separa il 25% inferiore dei dati dal resto del dataset, mentre il terzo quartile (Q3) separa il 75% inferiore dal 25% superiore. Un valore  $x$  potrebbe essere definito come *outlier* se non appartenente all'intervallo:

$$[Q1 - 1.5 \cdot IQR, Q3 + 1.5 \cdot IQR]$$

dove  $IQR$  (Interquartile Range) è la differenza tra il terzo e il primo quartile ( $IQR = Q3 - Q1$ ).

Per esempio, se in un dataset il primo quartile è 10 e il terzo quartile è 20, l'intervallo per identificare gli outlier sarebbe:

$$[10 - 1.5 \cdot (20 - 10), 20 + 1.5 \cdot (20 - 10)] = [-5, 35]$$

Quindi, qualsiasi valore al di fuori di questo intervallo sarebbe considerato un outlier.

### 4.3.3 Scala dei dati

La scala dei dati si riferisce all'intervallo di valori che una variabile può assumere. In alcuni casi, le variabili possono avere scale molto diverse, il che può influenzare negativamente le prestazioni di alcuni algoritmi. Si può pensare ad esempio a due variabili: altezza (in

centimetri) e peso (in chilogrammi). L'altezza può variare da 150 a 200 cm, mentre il peso può variare da 50 a 150 kg. Se si utilizzano queste variabili senza alcuna normalizzazione, l'algoritmo potrebbe dare più importanza alla variabile con la scala più ampia (in questo caso, l'altezza).

**Standardizzazione.** La standardizzazione è una tecnica che trasforma i dati in modo che abbiano una media di 0 e una deviazione standard di 1. La formula per standardizzare un valore  $x$  è:

$$z = \frac{x - \mu}{\sigma}$$

dove  $\mu$  è la media dei dati e  $\sigma$  è la deviazione standard:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i, \quad \sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

**Min-Max scaling.** Il Min-Max scaling è una tecnica che trasforma i dati in modo che rientrino in un intervallo  $[0, 1]$ . Viene considerato un metodo meno robusto della standardizzazione, perché è sensibile agli outlier. Dato un valore  $x$ , la formula per il Min-Max scaling è:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

dove  $x_{min}$  e  $x_{max}$  sono rispettivamente il valore minimo e massimo dei dati.

## 4.4 Riduzione dei dati

La riduzione dei dati consiste nel *rappresentare* i dati in maniera più compatta, in modo da facilitare l'uso di algoritmi di analisi e modellazione.

### 4.4.1 Sampling dei dati

Un modo semplice per ridurre la quantità di dati è il **sampling**, ovvero la selezione di un sottoinsieme rappresentativo del dataset originale. Esistono diverse tecniche di sampling, tra cui:

**Biased** - Selezione di istanze in base a criteri specifici, come la frequenza di una classe o la rilevanza per un particolare obiettivo di analisi.

**Random** - Selezione casuale di istanze dal dataset originale, garantendo che ogni istanza abbia la stessa probabilità di essere selezionata.

**Stratificato** - Suddivisione del dataset in sottogruppi (strati) basati su una o più caratteristiche, e successiva selezione casuale di istanze da ciascuno strato per garantire una rappresentazione equilibrata delle diverse categorie nel sottoinsieme.

#### 4.4.2 Selezione di feature

La selezione di feature consiste nel scartare dai dati attributi che sono irrilevanti per l'analisi. La rilevanza delle feature **dipende** dal dominio del problema. Ad esempio, in un dataset di immagini, le feature relative al colore potrebbero essere irrilevanti per un'analisi che si concentra sulla forma degli oggetti presenti nell'immagine. Le tecniche possono essere **supervised**, ovvero basate su etichette di classe, oppure **unsupervised**, ovvero basate solo sulle caratteristiche intrinseche dei dati.

#### 4.4.3 Riduzione della dimensionalità

I dati reali spesso contengono molte feature, alcune delle quali possono essere ridondanti o irrilevanti, spesso difficili da notare oppure *implicite*. La riduzione della dimensionalità mira a ridurre il numero di feature mantenendo quante più informazioni possibili.

Una possibile tecnica di riduzione è quella di individuare una rotazione degli assi, ovvero una nuova base, in cui i dati possano essere rappresentati in modo più compatto. Ricordando dall'algebra lineare, una rotazione degli assi è un cambiamento di sistema di coordinate che preserva le distanze e gli angoli tra i punti. In altre parole, i dati vengono proiettati su un nuovo insieme di assi che sono combinazioni lineari delle feature originali.

#### 4.4.4 PCA: Principal Component Analysis

La PCA è una tecnica di riduzione della dimensionalità che identifica le direzioni principali (componenti principali). Per capire quali sono, le componenti principali, dobbiamo partire dalla matrice di **covarianza** dei dati: dati due vettori  $X, Y$  la covarianza misura la *varianza* di  $X$  rispetto a  $Y$ , ovvero quanto variano insieme. Se a più alti valori di  $X$  corrispondono più alti valori di  $Y$ , la covarianza sarà positiva, mentre se a più alti valori di  $X$  corrispondono più bassi valori di  $Y$ , la covarianza sarà negativa. La covarianza tra due variabili  $X$  e  $Y$  è calcolata come:

$$\begin{aligned} C(X, Y) &= \frac{1}{N} \sum_{k=1}^N (X_k - \mu_X)(Y_k - \mu_Y) \\ \Rightarrow c_{ij} &= \frac{x_i \cdot x_j}{N} - \mu_i \mu_j \end{aligned}$$

dove  $\mu_X$  e  $\mu_Y$  sono le medie di  $X$  e  $Y$ , rispettivamente, e  $N$  è il numero di osservazioni.

La matrice di covarianza  $C$  gode di diverse proprietà:

- La covarianza di un vettore con se stesso equivale alla **varianza** del vettore:  $C(X, X) = \sigma_X^2$ .
- La matrice di covarianza è **simmetrica**, ovvero  $C = C^T \Rightarrow C(X, Y) = C(Y, X)$ .
- Il segno della covarianza indica la direzione della relazione tra le variabili: una covarianza positiva indica che le variabili tendono a variare nella stessa direzione, mentre una covarianza negativa indica che variano in direzioni opposte.

Le componenti principali sono le direzioni lungo le quali i dati variano maggiormente. Queste direzioni sono rappresentate dagli **autovettori** della matrice di covarianza, mentre la quantità di varianza spiegata da ciascuna componente principale è rappresentata dagli **autovalori** corrispondenti.

Si consideri un dataset rappresentato da una matrice  $D$  con  $M$  osservazioni (righe) e  $N$  feature (colonne). A partire da  $D$  si costruisce la matrice di covarianza, da cui si ricavano le componenti principali, ovvero nuovi attributi ottenuti come combinazioni lineari degli attributi originari. I nuovi attributi presentano due proprietà fondamentali:

- Sono tra di loro **non correlate**, ovvero la covarianza tra due componenti principali è zero.
- Concentrano la maggior parte della **varianza** dei dati originali in poche dimensioni.

In presenza di forte correlazione tra attributi, poche componenti principali riescono a spiegare gran parte della varianza totale del dataset. Questo permette di ridurre la dimensionalità del dataset mantenendo la maggior parte delle informazioni rilevanti.

I passi principali della PCA sono:

**Standardizzazione dei dati.** I dati vengono standardizzati per avere media zero e deviazione standard uno. Ciò è necessario perché la PCA è sensibile alla varianza iniziale dei dati nelle varie dimensioni, infatti dimensioni con varianze molto diverse possono dominare la direzione delle componenti principali.

**Calcolo delle componenti principali.** Grazie alla PCA si può decomporre la matrice di covarianza nel prodotto di 3 matrici:

$$C = P \Lambda P^T$$

dove  $P$  è la matrice degli autovettori (le componenti principali) e  $\Lambda$  è la matrice diagonale degli autovalori (la varianza spiegata da ciascuna componente principale).

**Selezione delle componenti principali.** Le componenti principali determinate sono uguali al numero di feature originali. La prima componente principale, però, **cattura** la più alta varianza nei dati e da un punto di vista geometrico corrisponde nel trovare la retta dove le proiezioni dei punti hanno la massima varianza. La seconda componente principale è ortogonale alla prima e cattura la seconda più alta varianza, e così via. Si selezionano le prime  $k$  componenti principali che spiegano una percentuale significativa della varianza totale (ad esempio, il 95%).

**Trasformazione dei dati.** Infine, utilizzando il dataset  $D : M \times N$  la matrice  $P : N \times k$  ottenuta concatenando per colonna i primi  $k$  autovettori selezionati, si ottiene il dataset ridotto  $D' : M \times k$  tramite la moltiplicazione:

$$D' = D \cdot P$$

#### 4.4.5 SVD: Singular Value Decomposition

La Singular Value Decomposition (SVD) è un'altra tecnica di riduzione della dimensionalità che decomponete una matrice  $M$  di dimensione  $n \times d$  nel prodotto:

$$M = U\Sigma V^T$$

dove:

- $U$  è una matrice ortogonale di dimensione  $n \times n$  le cui colonne sono chiamate *left singular vectors*.
- $\Sigma$  è una matrice diagonale di dimensione  $n \times d$  i cui elementi diagonali sono chiamati *singular values*, ordinati in ordine decrescente.
- $V$  è una matrice ortogonale di dimensione  $d \times d$  le cui colonne sono chiamate *right singular vectors*.

**Interpretazione geometrica.** La SVD può essere interpretata come una rotazione e una scalatura dello spazio dei dati. I *right singular vectors* (colonne di  $V$ ) rappresentano le direzioni principali nello spazio delle feature, mentre i *left singular vectors* (colonne di  $U$ ) rappresentano le direzioni principali nello spazio delle osservazioni. I *singular values* nella matrice  $\Sigma$  indicano l'importanza di ciascuna direzione.

Consideriamo il caso in cui  $M$  sia una matrice  $2 \times 2$  e agisca quindi sul piano  $\mathbb{R}^2$ . Partiamo da un cerchio unitario con i due vettori canonici. La SVD

$$M = U\Sigma V^T$$

può essere vista come una successione di tre operazioni semplici. La prima trasformazione, data da  $V^T$ , ruota il cerchio e riallinea gli assi secondo le direzioni individuate dai *right singular vectors*. La matrice diagonale  $\Sigma$  applica poi una scalatura lungo tali direzioni, trasformando il cerchio in un'ellisse i cui semiassi hanno lunghezze pari ai valori singolari non nulli. Infine, la matrice  $U$  effettua un'ulteriore rotazione, orientando l'ellisse nelle direzioni dei *left singular vectors*. In sintesi, nella SVD la matrice  $M$  agisce come una combinazione di rotazioni e scalature che mappa la sfera unitaria in un'ellissoide.

**Varianti ridotte della SVD.** Esistono varianti della SVD che permettono di ridurre il tempo computazionale e lo spazio di memoria necessari per calcolare la decomposizione, specialmente quando la matrice  $M$  è di grandi dimensioni o sparsa. La variante principale si chiama **Full-SVD** ed è la versione completa della SVD descritta sopra, ne esistono tuttavia altre versioni:

**Think SVD** - In questo approccio si rimuovono le colonne di  $U$  e le righe di  $\Sigma$  in eccesso rispetto alle colonne della matrice  $V$ , in modo da assicurare una decomposizione più compatta. La matrice  $U$  diventa quindi di dimensione  $n \times r$ , dove  $r$  è il rango della matrice  $M$ , mentre  $\Sigma$  diventa una matrice diagonale di dimensione  $r \times r$ .

**Compact SVD** - Questa versione rimuove le righe di  $\Sigma$  che contengono valori singolari nulli e di conseguenza anche le colonne di  $U$  e le righe di  $V^T$  in eccesso rispetto al nuovo numero di righe di  $\Sigma$ . La matrice  $U$  diventa quindi di dimensione  $n \times k$ , dove  $k$  è il numero di valori singolari non nulli, mentre  $\Sigma$  diventa una matrice diagonale di dimensione  $k \times k$ .

**Truncated SVD** - In questa variante si selezionano solo i primi  $k$  valori singolari più grandi e le corrispondenti colonne di  $U$  e righe di  $V^T$ . Questo approccio è particolarmente utile quando si desidera ridurre la dimensionalità dei dati mantenendo solo le componenti più significative. La matrice  $U$  diventa quindi di dimensione  $n \times k$ , mentre  $\Sigma$  diventa una matrice diagonale di dimensione  $k \times k$ .

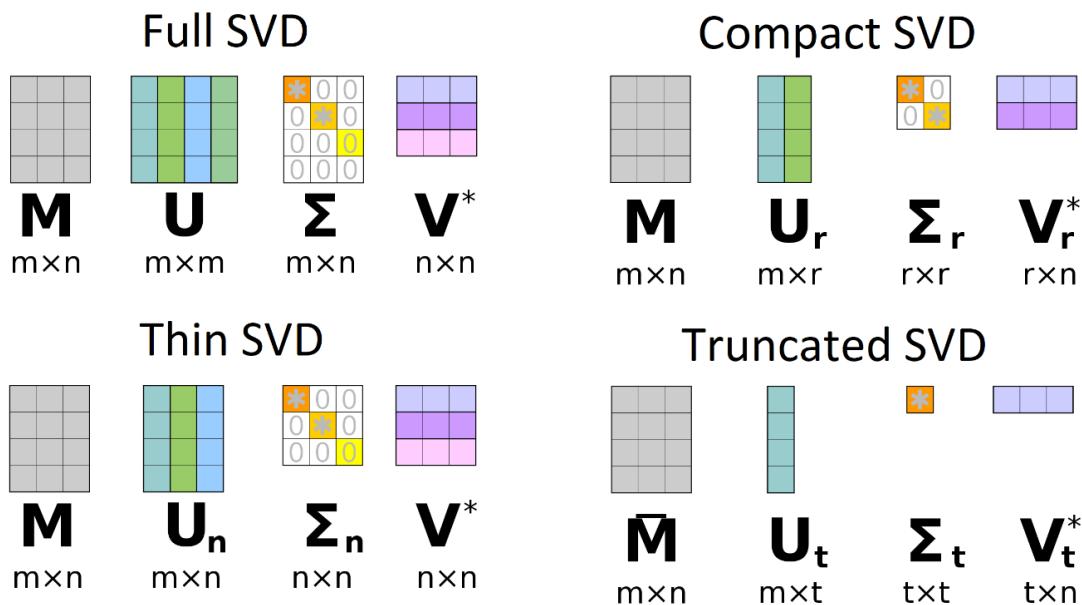


Figura 4.1: Confronto tra le principali varianti della Decomposizione ai Valori Singolari (SVD): Full SVD, Compact SVD, Thin SVD e Truncated SVD, con rappresentazione delle dimensioni delle matrici coinvolte.

**SVD vs PCA.** La SVD e la PCA sono entrambe tecniche di riduzione della dimensionalità, ma presentano alcune differenze chiave:

- SVD è più generale della PCA, poiché produce due set di autovettori anziché uno solo. La PCA può essere vista come un caso speciale della SVD applicata alla matrice di covarianza.
- SVD corrisponde alla PCA nel caso in cui i dati sono centrati attorno allo zero, ovvero quando la media dei valori di ogni attributo è zero.
- La PCA cattura quanto più varianza possibile nei dati, la SVD cattura quanta più distanza euclidea al quadrato rispetto all'origine possibile.

#### 4.4.6 LSA: Latent Semantic Analysis

La Latent Semantic Analysis (LSA) è una tecnica di riduzione della dimensionalità utilizzata principalmente nell'elaborazione del linguaggio naturale e nel recupero delle informazioni. LSA mira a identificare le relazioni semantiche tra parole e documenti, riducendo la dimensionalità dello spazio delle caratteristiche.

La matrice di partenza è una matrice  $n \times d$  di  $n$  documenti e  $d$  termini, contenente le frequenze normalizzate delle parole in ciascun documento. La LSA utilizza la SVD per decomporre questa matrice nei suoi componenti principali, identificando le direzioni principali nello spazio delle caratteristiche che catturano le relazioni semantiche tra parole e documenti.

#### 4.4.7 Riduzione di dimensionalità con trasformazione dei dati

Oltre alle tecniche basate su decomposizioni matriciali, esistono metodi di riduzione della dimensionalità che si basano sulla trasformazione non lineare dei dati. Questi metodi cercano di mappare i dati originali in uno spazio a bassa dimensionalità preservando le relazioni strutturali tra i punti dati.

**Esempio: serie temporali.** Un esempio di riduzione della dimensionalità basata sulla trasformazione dei dati è l'analisi delle serie temporali. Le serie temporali sono sequenze di dati raccolti nel tempo, e spesso presentano una struttura complessa che può essere difficile da analizzare direttamente. Tecniche come la trasformata di Fourier o la trasformata wavelet possono essere utilizzate per rappresentare le serie temporali in uno spazio a bassa dimensionalità, catturando le caratteristiche principali delle variazioni temporali.

## Capitolo 5

# Insiemi Frequenti e Regole d'Associazione

### 5.1 Market-basket model e definizioni

Nel *market-basket model* ogni transazione (*basket*) è un insieme di oggetti (*item*). L'obiettivo è individuare **itemset frequenti**, cioè insiemi di item che compaiono assieme in molte transazioni, e derivarne **regole d'associazione** utili per descrivere co-occorrenze interessanti. Per farlo, abbiamo bisogno di definire delle nozioni di base:

#### 5.1.1 Supporto

Sia  $\mathcal{D}$  l'insieme dei basket ( $|\mathcal{D}|=N$ ) e sia  $I \subseteq \mathcal{I}$  un itemset. Il **supporto** assoluto di  $I$  è

$$\text{supp}(I) = |\{T \in \mathcal{D} : I \subseteq T\}|, \quad \text{supp}_{\text{rel}}(I) = \frac{\text{supp}(I)}{N}.$$

Dato un valore soglia  $\sigma$  (*min-sup*),  $I$  è **frequente** se  $\text{supp}(I) \geq \sigma$ .

Per fare un esempio, con  $N=5$  e transazioni  $\{a, b, c\}, \{a, c\}, \{b, c\}, \{a, b\}, \{a, b, c\}$ , l'itemset  $\{a, b\}$  ha supporto 3 (frequente se  $\sigma \leq 3$ ), mentre  $\{b, c\}$  ha supporto 3 (frequente se  $\sigma \leq 3$ ) e  $\{a, b, c\}$  ha supporto 2 (frequente se  $\sigma \leq 2$ ).

**Soglia di supporto: trade-off.** Soglie alte eliminano pattern rari ma potenzialmente informativi; soglie basse provocano un'esplosione di candidati, con costi elevati nel conteggio e nella validazione.

### 5.2 Regole d'associazione

Una **regola d'associazione** è un'implicazione  $X \rightarrow j$  con  $X$  itemset e  $j$  un singolo item,  $j \notin X$ . Si estende a  $X \rightarrow Y$  con  $X \cap Y = \emptyset$ . La regola afferma che la presenza di  $X$  in una transazione implica (con una certa probabilità) la presenza di  $j$ . Le regole si estraggono dagli insiemi frequenti: se  $I$  è frequente e  $j \in I$ , allora  $X = I \setminus \{j\}$  produce la regola  $X \rightarrow j$ .

### 5.2.1 Qualità di una regola

**Confidenza.** La confidenza di  $X \rightarrow j$  è

$$\text{conf}(X \rightarrow j) = \frac{\text{supp}(X \cup \{j\})}{\text{supp}(X)} = P(j | X).$$

**Coverage.**  $\text{supp}(X)$  è anche *coverage*: misura la copertura/applicabilità della regola.

**Interesse.** Scostamento rispetto alla prevalenza marginale di  $j$ :

$$\text{int}(X \rightarrow j) = \text{conf}(X \rightarrow j) - \frac{\text{supp}(\{j\})}{N}.$$

**Lift.** Rapporto tra co-occorrenza osservata e quella attesa in indipendenza:

$$\text{lift}(X \rightarrow j) = \frac{N \cdot \text{supp}(X \cup \{j\})}{\text{supp}(X) \text{supp}(\{j\})} = \frac{\text{conf}(X \rightarrow j)}{\text{supp}(\{j\})/N}.$$

Valori  $> 1$  indicano associazione positiva;  $< 1$  negativa.

Supporto e confidenza alti possono produrre regole ovvie o ridondanti perché guidate da item molto frequenti. Altre metriche come lift (o *leverage*:  $\text{supp}(XY) - \text{supp}(X)\text{supp}(Y)/N$ ) aiutano a individuare co-occorrenze non banali.

**Mini-esempio (toy dataset).** Sia  $N=8$  con item  $\{b, c, j, m, p\}$ . Supponiamo:

$$\text{supp}(\{b\})=6, \text{supp}(\{c\})=5, \text{supp}(\{j\})=4, \text{supp}(\{m\})=5, \text{supp}(\{p\})=2,$$

e, tra le coppie,  $\text{supp}(\{b, c\})=4, \text{supp}(\{c, j\})=3, \text{supp}(\{c, m\})=2, \text{supp}(\{m, p\})=2$ , ecc. Per la regola  $\{c, m\} \rightarrow b$ :

$$\text{conf} = \frac{\text{supp}(\{b, c, m\})}{\text{supp}(\{c, m\})} = \frac{2}{2} = 1.0, \quad \text{lift} = \frac{8 \cdot 2}{2 \cdot 6} = 1.33.$$

## 5.3 Insiemi frequenti chiusi e massimali

Sia  $I$  frequente.

- $I$  è **chiuso** se nessun suo superinsieme ha lo *stesso* supporto di  $I$ .
- $I$  è **massimale** se nessun suo superinsieme è frequente.

Gli insiemi massimali sono chiusi; gli insiemi chiusi forniscono una rappresentazione più compatta (mantengono il supporto dei soli chiusi).

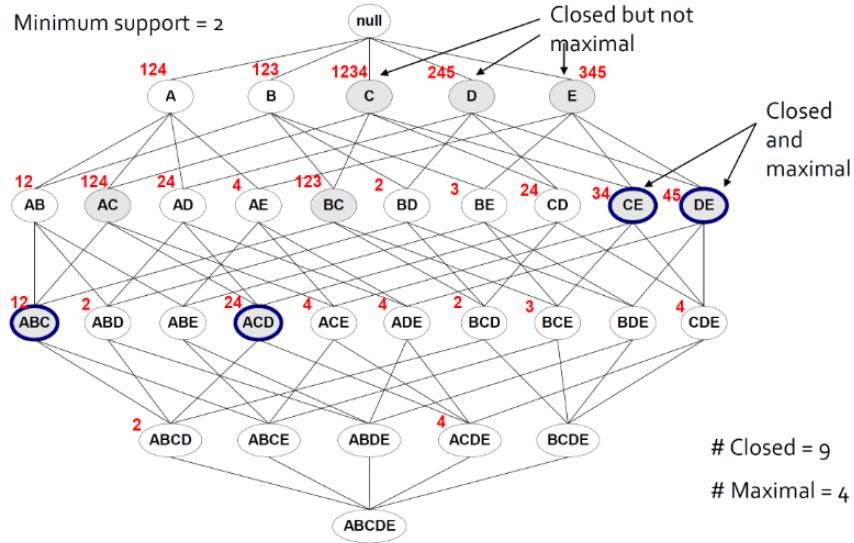


Figura 5.1: Grafo degli itemset con  $\text{minsup} = 2$ . *Frequente*: supporto  $\geq 2$ ; *chiuso*: nessun superinsieme con lo stesso supporto; *massimale*: nessun superinsieme frequente. Esempio con chiusi (es. CE) e chiusi-massimali (CE, DE).

## 5.4 Anti-monotonie e Principio di Apriori

Per  $S \subseteq I$  vale l'**anti-monotonicità del supporto**:

$$\text{supp}(I) \leq \text{supp}(S).$$

Da questo principio, nasce il **Principio di Apriori**.

### 5.4.1 Principio di Apriori

Se  $I$  è frequente, ogni suo sottoinsieme è frequente; equivalentemente, se  $I$  non è frequente, nessun suo superinsieme può esserlo.

Questo principio consente di ridurre lo spazio di ricerca degli insiemi frequenti: se un candidato  $C$  contiene un sottoinsieme non frequente,  $C$  può essere scartato senza calcolarne il supporto.

## 5.5 Algoritmo Apriori

Ricerca *bottom-up* per cardinalità crescente.

1. Calcola  $L_1$  (item singoli frequenti).
2. Per  $k = 1, 2, \dots$ :
  - (a) **Join**: genera  $C_{k+1}$  (candidati di taglia  $k+1$ ) con self-join di  $L_k$ .

- (b) **Prune**: elimina da  $C_{k+1}$  i candidati che contengono un sottoinsieme di taglia  $k$  non frequente (§5.4).
- (c) **Conteggio**: scansiona il DB e costruisci  $L_{k+1} = \{c \in C_{k+1} : \text{supp}(c) \geq \sigma\}$ .
3. Arresta quando  $C_{k+1} = \emptyset$ .

### 5.5.1 Esempio Apriori (minsup = 2)

Item	<i>b</i>	<i>c</i>	<i>j</i>	<i>m</i>	<i>p</i>
supp( $\cdot$ )	6	5	4	5	2

Con minsup = 2,  $L_1 = \{b, c, j, m, p\}$ .

$k = 1 \rightarrow 2$ : generazione  $C_2$  e conteggi.

$$\{b, c\}, \{b, j\}, \{b, m\}, \{b, p\}, \{c, j\}, \{c, m\}, \{c, p\}, \{j, m\}, \{j, p\}, \{m, p\}.$$

Itemset	$\{b, c\}$	$\{b, j\}$	$\{b, m\}$	$\{b, p\}$	$\{c, j\}$	$\{c, m\}$	$\{c, p\}$	$\{j, m\}$	$\{j, p\}$	$\{m, p\}$
supp( $\cdot$ )	4	2	4	1	3	2	0	2	1	2

Quindi  $L_2 = \{\{b, c\}, \{b, j\}, \{b, m\}, \{c, j\}, \{c, m\}, \{j, m\}, \{m, p\}\}$ .

$k = 2 \rightarrow 3$ : self-join e prune.

$$C_3 = \{\{b, c, j\}, \{b, c, m\}, \{b, j, m\}, \{c, j, m\}\}.$$

Itemset	$\{b, c, j\}$	$\{b, c, m\}$	$\{b, j, m\}$	$\{c, j, m\}$
supp( $\cdot$ )	2	2	1	1

Quindi  $L_3 = \{\{b, c, j\}, \{b, c, m\}\}$ ;  $L_4 = \emptyset$ .

### 5.5.2 Generazione dei candidati

Se gli item sono ordinati, due insiemi  $A = (a_1, \dots, a_{k-1}, x)$  e  $B = (a_1, \dots, a_{k-1}, y)$  in  $L_k$  con  $x < y$  producono  $(a_1, \dots, a_{k-1}, x, y)$ . Il *prune* scarta i candidati che hanno almeno un sottoinsieme di taglia  $k$  non in  $L_k$ .

## 5.6 Ottimizzazioni di Apriori

### 5.6.1 Hashing in bucket: PCY

Alla prima passata si contano i singoli item e, in parallelo, si proiettano tutte le coppie in bucket tramite hash. I bucket sotto soglia sono marcati come non frequenti: alla seconda passata, una coppia  $(i, j)$  è candidata solo se *entrambi* gli item sono frequenti e il bucket di  $(i, j)$  è frequente. Riduce notevolmente  $|C_2|$ .

### Varianti multistadio e multihash.

- **Multistage PCY:** più passate di hashing con funzioni diverse; un candidato è ammesso solo se sopravvive a tutti i filtri (meno falsi positivi rispetto a PCY base).
- **Multihash PCY:** nella stessa passata si usano più funzioni hash indipendenti; una coppia è candidata se tutti i bucket corrispondenti sono frequenti (filtro più severo).

#### 5.6.2 Partizionamento del DB: SON

Divide il dataset in partizioni; su ciascuna esegue Apriori con min-sup scalato. L'unione dei frequenti locali fornisce i candidati globali, poi verificati su tutto il DB. Adatto a MapReduce e ambienti distribuiti.

#### 5.6.3 Campionamento e frontiera negativa: Toivonen

Esegue Apriori su un campione  $S$  con soglia più bassa  $\sigma'$ ; produce un insieme di candidati e la *frontiera negativa* (insiemi non frequenti in  $S$  con tutti i sottoinsiemi immediati frequenti). Se nessun elemento della frontiera negativa risulta frequente a livello globale, l'output è corretto; altrimenti si ripete con nuovo campione (evitando falsi negativi).

## 5.7 Perché andare oltre Apriori

Apriori richiede (i) generare esplicitamente  $C_k$  a ogni livello e (ii) molte passate sul DB per i conteggi. Con soglie basse o molti pattern, i candidati esplodono e le scansioni sono costose. **FP-Growth** evita entrambi: rappresenta il DB in modo compatto (FP-tree) e *fa crescere* i pattern senza generare  $C_k$ .

## 5.8 FP-Growth: idea di base

1. **Costruzione FP-tree:** prima passata per  $\text{supp}(i)$ ; scarta item sotto soglia, ordina gli item per supporto decrescente, reinserirsi le transazioni condividendo i prefissi (header table + node-link).
2. **Pattern-growth:** per ogni item  $x$  (dal meno al più frequente) estrai la *pattern base condizionale* dai cammini che portano a  $x$ , costruisci l'*FP-tree condizionale* e ripeti ricorsivamente. I pattern trovati si concatenano con  $x$ .

In genere bastano **due passate** sul DB; poi si lavora sull'FP-tree in memoria.

### 5.8.1 Costruzione dell'FP-tree

1. **Prima passata:** calcola  $\text{supp}(i)$  e rimuovi gli item con  $\text{supp}(i) < \sigma$ .
2. **Ordina** gli item per supporto decrescente; **riordina** ogni transazione seguendo lo stesso ordine.

3. **Inserisci** ogni transazione nell'albero dalla radice, condividendo prefissi e aggiornando i *node-link*.

*Proprietà:* l'FP-tree conserva i conteggi necessari ed è molto compatto quando molte transazioni condividono prefissi.

### 5.8.2 Esempio di FP-Growth

Soglia  $\sigma=3$ . Prima passata:

$$f:4, \quad c:4, \quad a:3, \quad b:3, \quad m:3, \quad p:3.$$

Ordine decrescente  $f > c > a > b > m > p$ .

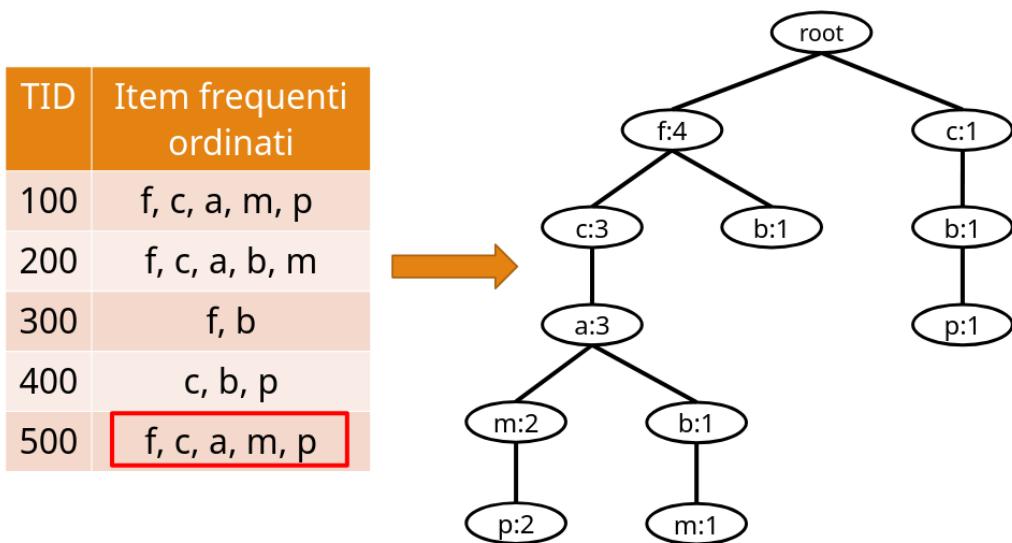


Figura 5.2: Costruzione dell'FP-tree: a sinistra transazioni riordinate; a destra l'albero con prefissi condivisi e contatori aggiornati.

**Visita per pattern-growth.** Processa gli item *dal meno al più frequente*:

$$p \rightarrow m \rightarrow b \rightarrow a \rightarrow c \rightarrow f.$$

**Espansione di un item  $x$ .**

1. **Pattern base condizionale:** segui i node-link di  $x$ ; per ogni occorrenza, prendi il cammino radice → genitore di  $x$  con *peso* uguale al contatore di  $x$ .
2. **FP-tree condizionale  $T_x$ :** somma i pesi per item, rimuovi quelli sotto soglia, ordina e inserisci i cammini pesati.
3. **Ricorsione e output:** concatena  $x$  a ogni pattern frequente trovato in  $T_x$ . *Caso path unica:* tutte le combinazioni dei nodi sono frequenti con supporto pari al *minimo* contatore lungo la combinazione.

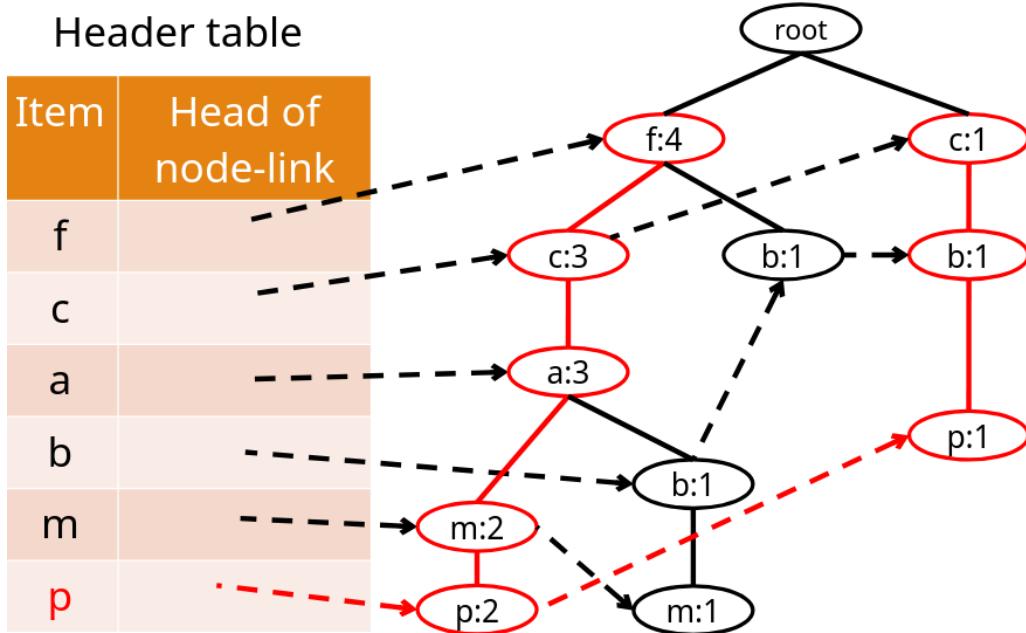


Figura 5.3: Header table e node-link per l'item  $p$ : la base condizionale di  $p$  si ottiene seguendo i link e risalendo verso la radice.

**Esempio 1: item  $p$ .** Cammini verso radice:  $\langle f, c, a, m \rangle : 2$  e  $\langle c, b \rangle : 1$ . Con  $\sigma=3$  nessuna combinazione con  $p$  è frequente.

**Esempio 2: item  $m$ .** Cammini:  $\langle f, c, a \rangle : 2$ ,  $\langle f, c \rangle : 1$ . Con  $\sigma=3$  risultano frequenti  $\{m, f\}$ ,  $\{m, c\}$  e  $\{m, f, c\}$  (supporto 3).

**Esempio 3: item  $b$ .** Cammini:  $\langle f, c, a \rangle : 2$ ,  $\langle c \rangle : 1$ . Con  $\sigma=3$  si ottiene  $\{b, c\}$  frequente; combinazioni con  $f$  o  $a$  non superano la soglia.

## 5.9 Confronto: FP-Growth vs Apriori

Aspetto	Apriori	FP-Growth
Generazione candidati	Sì: crea $C_k$ a ogni livello (rischio di esplosione)	No: crescita diretta dei pattern da FP-tree
Accessi al DB	Molte passate (una per $k$ )	Tipicamente 2, poi in memoria
Strutture dati	Liste di candidati e conteggi	FP-tree + header table
Quando preferirlo	DB piccoli/sparsi, soglie alte, ambienti distribuiti semplici	DB densi, soglie basse, prefissi condivisi (compressione efficace)
Note pratiche	Pruning via Apriori; implementazione semplice	Evita candidati; veloce se l'FP-tree è compatto

Tabella 5.1: Confronto sintetico tra Apriori e FP-Growth.



# Capitolo 6

## Clustering

### 6.1 Concetti generali

Il **clustering** raggruppa oggetti in *cluster* tali che i punti nello stesso cluster siano tra loro simili, mentre punti in cluster diversi siano dissimili. È un compito *unsupervised*: non si conoscono etichette a priori. La *classificazione* è invece *supervised* e richiede classi note per l'addestramento.

#### 6.1.1 Spazi metrici e funzioni distanza

Si assume uno **spazio metrico**  $(S, D)$ , con  $D$  che soddisfa:

$$\text{Non negatività} \quad D(x, y) \geq 0 \quad \forall x, y \in S,$$

$$\text{Simmetria} \quad D(x, y) = D(y, x) \quad \forall x, y \in S,$$

$$\text{Disuguaglianza triangolare} \quad D(x, y) + D(y, z) \geq D(x, z) \quad \forall x, y, z \in S.$$

**Distanze in spazi euclidei.** Siano  $\mathbf{x} = (x_1, \dots, x_n)$  e  $\mathbf{y} = (y_1, \dots, y_n) \in \mathbb{R}^n$ .

$$\text{Distanza euclidea: } D_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

$$\text{Distanza di Manhattan: } D_1(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^n |x_i - y_i|$$

$$\text{Norma } L_r : \quad D_r(\mathbf{x}, \mathbf{y}) = \left( \sum_{i=1}^n |x_i - y_i|^r \right)^{1/r}, \quad r \geq 1$$

$$\text{Norma } L_\infty : \quad D_\infty(\mathbf{x}, \mathbf{y}) = \max_{1 \leq i \leq n} |x_i - y_i|$$

$$\text{Distanza del coseno (angolare): } D_\angle(\mathbf{x}, \mathbf{y}) = \arccos \left( \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \right), \quad \mathbf{x} \neq \mathbf{0}, \mathbf{y} \neq \mathbf{0}.$$

**Spazi non euclidei.** Per oggetti-insiemi o stringhe il centroide può non avere senso: si usa il **medoide** (elemento del dataset che minimizza la somma delle distanze agli altri). Esempi di metriche:

$$D_{\text{Jac}}(S, T) = 1 - \frac{|S \cap T|}{|S \cup T|} \quad (\text{Jaccard}),$$

Altri esempi di distanze sono:

1. **Distanza di Edit:** il minimo numero di operazioni di cancellazione o inserzioni di caratteri da effettuare partendo da una stringa A per ottenere la stringa B. (es. A = abcde, B = acfdeg  $\Rightarrow D(A, B) = 3$ ).
2. **Distanza di Hamming:** dati A, B vettori, il numero di componenti in corrispondenza delle quali differiscono (es. A = (1, 0, 1, 0, 1), B = (1, 1, 1, 1, 0)  $\Rightarrow D(A, B) = 3$ ).

### 6.1.2 Tassonomia degli algoritmi

Tre famiglie principali:

1. **gerarchici** (agglomerativi/divisivi);
2. **partizionali** (es. k-means);
3. **a densità** (DBSCAN/OPTICS/HDBSCAN).

**Bontà di un algoritmo.** Dipende da: *scalabilità*, supporto a attributi eterogenei, capacità di cogliere *forme diverse* di cluster, *robustezza* a outlier/rumore e dati mancanti, *stabilità* all'aggiunta di nuovi dati, e *interpretabilità* dei risultati. La scelta pratica è un compromesso tra qualità e costi computazionali.

### 6.1.3 Alta dimensionalità: equidistanza e ortogonalità

Spazi euclidei ad elevata dimensionalità soffrono del *problema della dimensionalità*:

- quasi tutte le coppie di punti risultano **equidistanti** e lontane tra loro;
- quasi tutte le coppie di vettori sono quasi **ortogonali**.

**Equidistanza dei punti.** Sia  $D(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2$  con  $\mathbf{x}, \mathbf{y} \in [0, 1]^n$  indipendenti. Quando  $n$  è grande, con alta probabilità:

$$\underbrace{\frac{1}{\sqrt{n}}}_{\text{limite inferiore}} \lesssim D(\mathbf{x}, \mathbf{y}) \lesssim \underbrace{\sqrt{n}}_{\text{limite superiore}}$$

e solo una *frazione trascurabile* di coppie è vicina ai due limiti. La **maggior parte** delle coppie ha una distanza vicina alla media, circa  $\sqrt{n}/3$  (concentrazione della distanza). Inoltre i prodotti scalari tendono a 0, così gli angoli sono prossimi a  $90^\circ$  (quasi ortogonalità).

**Conseguenze pratiche.** Distinguere “vicini” da “lontani” diventa difficile; conviene standardizzare le feature, ridurre la dimensionalità (es. PCA) o usare metriche più adatte (cosine/angolare), soprattutto in presenza di dati sparsi.

## 6.2 Clustering gerarchico

**Schema agglomerativo.** (a) inizializza: ogni punto è un cluster; (b) ripeti: fonde i due cluster più vicini secondo una *distanza tra cluster*; (c) termina con un criterio (numero desiderato di cluster o qualità).

### 6.2.1 Distanza tra cluster (*linkage*)

- **Single-link:**  $\min\{D(x, y) : x \in C_i, y \in C_j\}$  (tende a catene).
- **Complete-link:**  $\max\{D(x, y) : x \in C_i, y \in C_j\}$  (favoreisce cluster compatti).
- **Average-link:** media delle distanze su tutte le coppie  $x \in C_i, y \in C_j$  (compromesso).
- **Centroid/medoid:** distanza tra centroidi (euclideo) o tra medoidi (generale).

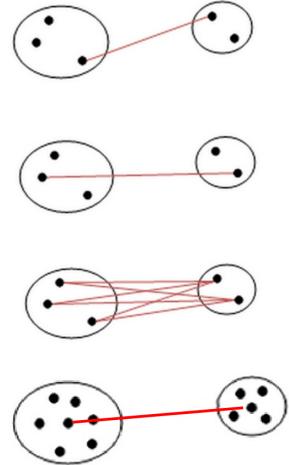


Figura 6.1: Esempi grafici delle diverse nozioni di distanza tra cluster.

### 6.2.2 Dendrogramma e criteri di stop

Il **dendrogramma** registra le fusioni; tagliandolo a una certa altezza si ottiene la partizione. Criteri di terminazione: (i) fermarsi a  $k$  cluster prefissati; (ii) fermarsi quando l’unione successiva degrada troppo la qualità (es. aumento del diametro o della distanza media intra-cluster).

### 6.2.3 Altri criteri di combinazione

Si può fondere la coppia che massimizza la *qualità* del cluster risultante. Definizioni utili:  $raggio = \max_{x \in C} D(x, \text{centroide}(C))$ ;  $diametro = \max_{x, y \in C} D(x, y)$ .

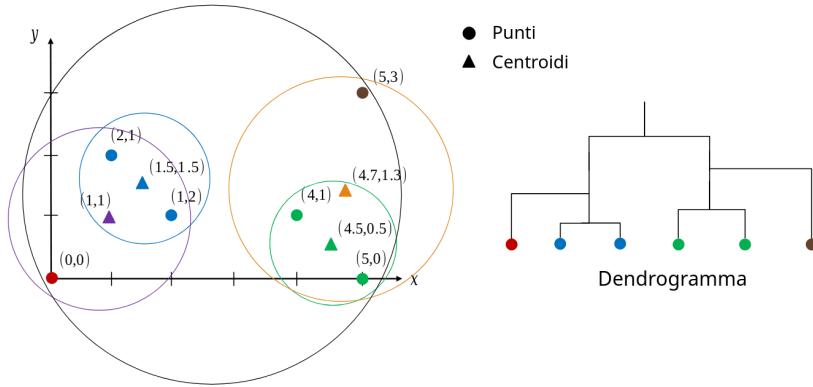


Figura 6.2: A sinistra: punti nel piano con centroidi (triangoli) e cerchi che schematizzano la coesione dei gruppi; i colori indicano i cluster. A destra: dendrogramma agglomerativo che mostra l’ordine di fusione e l’altezza (distanza di linkage). Un taglio orizzontale del dendrogramma determina il numero di cluster.

#### 6.2.4 Versioni divisive

Approccio **top-down**: un’altra versione dove si parte da un unico cluster e lo si *divide* iterativamente scegliendo il miglior *taglio*. Le stesse metriche di distanza/qualità si applicano in modo duale.

#### 6.2.5 Complessità e ottimizzazioni

**Analisi naive.** Al primo passo si valuta la distanza per ogni coppia di cluster e si sceglie la migliore: costo  $\Theta(n^2)$ . Dopo ogni fusione i cluster diminuiscono di uno, quindi i passi successivi costano, nell’ordine,  $(n-1)^2, (n-2)^2, \dots, 2^2$ .

$$T_{\text{naive}} = \sum_{k=2}^n k^2 = \frac{n(n+1)(2n+1)}{6} - 1 = \Theta(n^3).$$

(Spazio tipico: matrice delle distanze  $O(n^2)$ .)

**Ottimizzazione con coda di priorità.** Usando una coda di priorità (min-heap) sulle distanze tra cluster:

- accesso al minimo (*peek*) in  $O(1)$ ; inserimenti e cancellazioni in  $O(\log n)$ ;
- ad ogni fusione si *rimuovono* al più  $2(n-1)$  distanze (quelle dai due cluster che si fondono):  $O(n \log n)$ ;
- si *calcolano* e *inseriscono* le distanze tra il nuovo cluster e gli altri (al più  $n-2$ ):  $O(n \log n)$ .

Su  $n-1$  fusioni:

$$T_{\text{heap}} = O(n \cdot (n \log n)) = O(n^2 \log n).$$

Risultato: la complessità scende da  $O(n^3)$  a circa  $O(n^2 \log n)$  mantenendo la matrice (o la coda) aggiornata a ogni iterazione.

## 6.3 Clustering partizionale: k-means

Metodi per spazi euclidei che partizionano i dati in  $k$  cluster minimizzando la somma delle distanze al quadrato dai centroidi.

### 6.3.1 Algoritmo base

1. **Inizializza**  $k$  centroidi (idealmente separati).
2. **Assegna** ogni punto al centroide più vicino (distanza euclidea).
3. **Aggiorna** ogni centroide come media dei punti assegnati.
4. **Ripeti** finché i centroidi si stabilizzano o il miglioramento è sotto soglia.

Converge in pochi round, ma solo a un ottimo *locale*.

### 6.3.2 Inizializzazione

Scelta *greedy*:

1. Si sceglie il primo punto in maniera casuale o lo si aggiunge all'insieme  $S$  dei punti già selezionati, inizialmente vuoto.
2. Si calcola la massima distanza minima dai centroidi scelti.
3. Si ripete il passo 2. finché  $|S| < k$ .

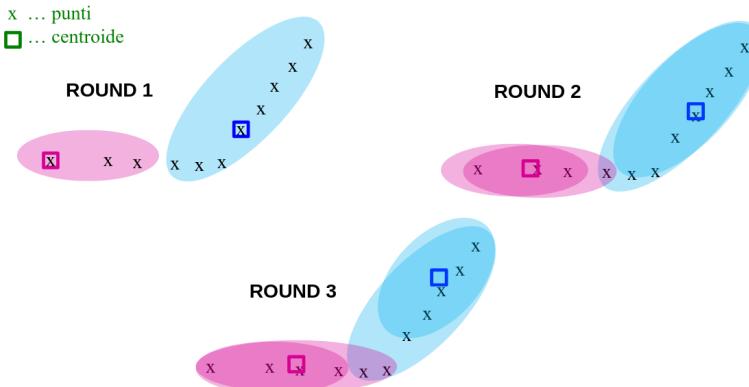


Figura 6.3: K-means: evoluzione in tre round. Round 1: inizializzazione e prime assegnazioni ai centroidi (quadrati). Round 2: ricalcolo dei centroidi e riassegnazione dei punti (X). Round 3: i centroidi si stabilizzano e i cluster (ellissi colorate) convergono.

### 6.3.3 Funzione obiettivo e arresto

Con partizione  $C_1, \dots, C_k$  e centroidi  $\mu_r$ :

$$J = \sum_{r=1}^k \sum_{x \in C_r} \|x - \mu_r\|_2^2.$$

Arresto quando  $\Delta J$  tra iterazioni consecutive è sotto soglia o quando non cambia l'assegnazione. Con metriche diverse da euclidea il centroide non è il minimizzatore naturale.

#### 6.3.4 Scelta del numero di cluster k

Poiché  $k$  non è noto a priori, si esegue il metodo per più valori e si seleziona quello che ottimizza una metrica di qualità interna.

**Funzione obiettivo.** Per  $k$  cluster  $C_1, \dots, C_k$  con centroidi  $\mathbf{c}_1, \dots, \mathbf{c}_k$ , la funzione standard è

$$W(k) = \sum_{i=1}^k \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \mathbf{c}_i\|_2^2, \quad \bar{W}(k) = \frac{W(k)}{n} \text{ (distanza media al centroide).}$$

$W(k)$  è decrescente in  $k$ .

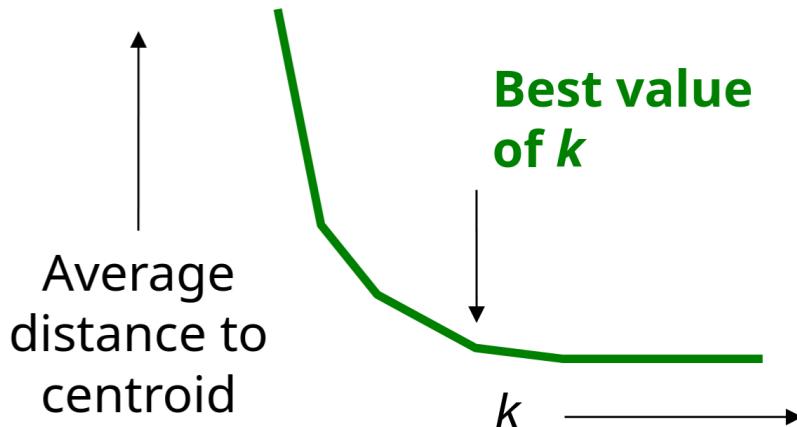


Figura 6.4: Metodo (*elbow*). Si traccia la distanza media dal centroide (o WCSS/n) al variare di  $k$ ; il valore “ottimo” è nel punto di flesso, dove l'aumento di  $k$  porta benefici marginali trascurabili.

**Metodo *elbow*.** Si calcola  $\bar{W}(k)$  per  $k = k_{\min}, \dots, k_{\max}$  e si sceglie il  $k$  per cui il calo di  $\bar{W}$  passa da “ripido” a “lento” (punto di flesso).

- *Procedura pratica:* si valuta  $\bar{W}(k)$  su una griglia di valori e si ispeziona il grafico  $\bar{W}$  vs  $k$ .
- *Variante a ricerca binaria:* fissati due estremi  $x < y$ , si prende  $z = \lfloor (x+y)/2 \rfloor$ , si calcola  $\bar{W}(z)$  e si sostituisce l'estremo *più vicino* a  $\bar{W}(z)$  con  $z$ ; si ripete finché l'intervallo è piccolo. Il  $k$  finale approssima il gomito.

*Nota:* se la curva non mostra un gomito netto, l'*elbow* diventa ambiguo e conviene affiancarlo a silhouette/stabilità.

**Metodo *silhouette*.** Per ogni punto  $\mathbf{x}$  assegnato al cluster  $C_i$ :

$$a(\mathbf{x}) = \frac{1}{|C_i| - 1} \sum_{\mathbf{y} \in C_i, \mathbf{y} \neq \mathbf{x}} \|\mathbf{x} - \mathbf{y}\|, \quad b(\mathbf{x}) = \min_{j \neq i} \frac{1}{|C_j|} \sum_{\mathbf{y} \in C_j} \|\mathbf{x} - \mathbf{y}\|.$$

Lo *score di silhouette* del punto è

$$s(\mathbf{x}) = \frac{b(\mathbf{x}) - a(\mathbf{x})}{\max\{a(\mathbf{x}), b(\mathbf{x})\}} \in [-1, 1].$$

Valori vicini a 1 indicano assegnazioni “pulite”, vicini a 0 punti al confine, negativi assegnazioni sbagliate. Si sceglie

$$k^* = \arg \max_k \frac{1}{n} \sum_{r=1}^n s(\mathbf{x}_r).$$

*Regole d’uso.* Calcolare gli indici su più esecuzioni (inizializzazioni diverse) e riportare media/deviazione; standardizzare le feature prima del confronto; evitare  $k$  troppo grandi che trivialiscono  $\bar{W}$  ma peggiorano la silhouette.

### 6.3.5 Complessità computazionale

Ogni iterazione del  $k$ -means ha due passi:

1. **Assegnamento** (nearest–centroid): per ciascun punto si valuta la distanza verso i  $k$  centroidi. Costo  $O(nkd)$  in  $\mathbb{R}^d$  (spesso si sottintende  $d$ , scrivendo  $O(nk)$ ).
2. **Aggiornamento dei centroidi:** si ricalcolano le medie dei cluster. Costo  $O(nd)$ .

Con  $t$  iterazioni totali:

$$T(n, k, d, t) = O(t n k d) \quad (\text{nelle slide: } O(tkn)).$$

## 6.4 Clustering per densità

### 6.4.1 DBSCAN

DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) definisce i cluster come regioni con densità elevata separate da regioni a bassa densità. Richiede due parametri:

$$\varepsilon > 0 \quad (\text{raggio dell’intorno}) \quad \text{e} \quad \text{MinPts} \in \mathbb{N} \quad (\text{soglia di densità}).$$

**Definizioni.** Dato un punto  $p$  e una metrica  $D(\cdot, \cdot)$ :

- **$\varepsilon$ -intorno:**  $N_\varepsilon(p) = \{x : D(x, p) \leq \varepsilon\}$ .
- **Core point:**  $p$  è *core* se  $|N_\varepsilon(p)| \geq \text{MinPts}$ .
- **Directly density-reachable:**  $q$  è *direttamente raggiungibile per densità* da  $p$  se  $q \in N_\varepsilon(p)$  e  $p$  è core.

- **Density-reachable:**  $q$  è raggiungibile per densità da  $p$  se esiste una catena  $p = x_0, x_1, \dots, x_m = q$  in cui ogni  $x_{i+1}$  è direttamente raggiungibile da  $x_i$ .
- **Density-connected:**  $p$  e  $q$  sono connessi per densità se esiste  $o$  tale che  $p$  e  $q$  sono entrambi raggiungibili per densità da  $o$ .

Cluster  $C$  = insieme massimale di punti *density-connected*; i punti non assegnati sono *rumore* (outlier). Punti non-core inclusi in un cluster sono detti *border*.

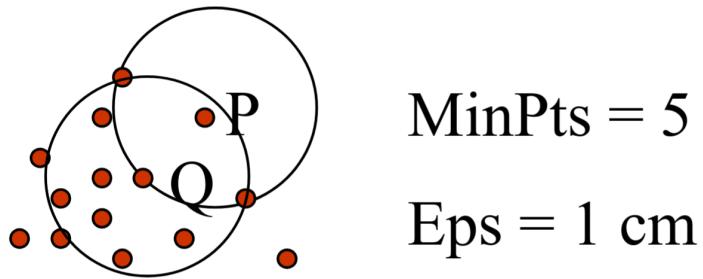


Figura 6.5: DBSCAN: esempi di *core*, *border* e *noise* con parametri  $\varepsilon$  e MinPts.

### Algoritmo.

1. Visita un punto non ancora etichettato  $p$  e calcola  $N_\varepsilon(p)$ .
2. Se  $p$  è *core*, crea un nuovo cluster e *espandilo*: aggiungi ricorsivamente tutti i punti direttamente raggiungibili, iterando finché possibile (tutti i raggiungibili per densità da  $p$ ).
3. Se  $p$  non è core e non è stato assegnato come *border*, etichettalo come *rumore*.
4. Ripeti finché tutti i punti sono visitati.

**Scelta dei parametri.** Usare  $\text{MinPts} \approx d+1$  come minimo (con  $d$  dimensioni) e valori più alti con dataset grandi o rumorosi. Per  $\varepsilon$  si usa il *k-distance plot*: per ogni punto si considera la distanza dal  $k$ -esimo vicino ( $k = \text{MinPts}$ ), si ordina in senso decrescente e si cerca il *gomito* della curva.

**Complessità.** Il costo è dominato dalle ricerche di vicinato. Con appropriate strutture (R-tree) il costo è  $O(n \log n)$ . Senza questo tipo di strutture, il costo è  $O(n^2)$ .

**Pro e contro.** *Pro:* non richiede  $k$ , trova cluster di forma arbitraria, gestisce il rumore, poco sensibile all'ordine di scansione. *Contro:* scelta di  $(\varepsilon, \text{MinPts})$  non banale; difficile con densità molto diverse o in alta dimensione.

### 6.4.2 OPTICS

OPTICS (*Ordering Points To Identify the Clustering Structure*) estende DBSCAN per gestire **densità variabili**. Invece di una singola partizione, produce un *ordinamento* dei

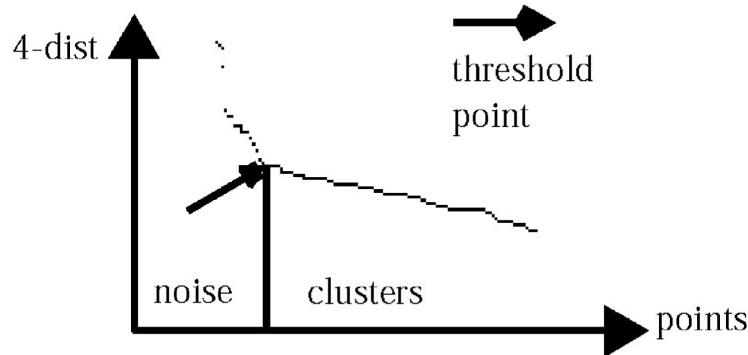


Figura 6.6:  $k$ -distance plot: il gomito suggerisce il valore di  $\varepsilon$ .

punti con associata una misura di “raggiungibilità” che riassume la struttura di densità a più scale.

**Core-distance e reachability (OPTICS).** Fissiamo  $\text{MinPts} = k$  e una metrica  $D$ .

**Core-distance di  $p$ :** è il raggio minimo che rende  $p$  un punto *core*. In pratica è la distanza dal  $k$ -esimo vicino di  $p$ :

$$\text{core\_dist}_k(p) = d_k(p).$$

Se  $p$  ha meno di  $k$  vicini, non è core e si pone  $\text{core\_dist}_k(p) = +\infty$  (non definita).

**Reachability-distance di  $o$  da  $p$ :** misura quanto è “raggiungibile”  $o$  partendo da  $p$ :

$$\text{reach\_dist}_k(o | p) = \max\{\text{core\_dist}_k(p), D(p, o)\}.$$

#### Lettura immediata.

- Se  $p$  è in una regione densa ( $\text{core\_dist}_k(p)$  piccola) e  $o$  è *dentro* quel raggio, allora  $\text{reach\_dist}_k(o | p) = \text{core\_dist}_k(p)$  (tutti questi  $o$  “valgon lo stesso” nel plot).
- Se  $o$  è *più lontano* del raggio denso di  $p$ ,  $\text{reach\_dist}_k(o | p) = D(p, o)$  (serve “uscire” dalla zona densa).

**Mini-esempio.** Con  $k = 5$  e  $\text{core\_dist}_5(p) = 0.8$ : un vicino a distanza 0.6 ha  $\text{reach\_dist} = 0.8$ ; un punto a distanza 1.2 ha  $\text{reach\_dist} = 1.2$ .

**Risultato: ordering e reachability plot.** OPTICS visita iterativamente i punti scegliendo, tramite una coda di priorità, quello con  $\text{reach\_dist}$  minima; registra per ciascun punto l’ordine di visita e la sua *reachability*. Plotando le *reachability* nell’ordine si ottiene il **reachability plot**: le *valli* corrispondono a cluster, le creste a separazioni.

**Estrazione dei cluster.** Si possono ottenere partizioni:

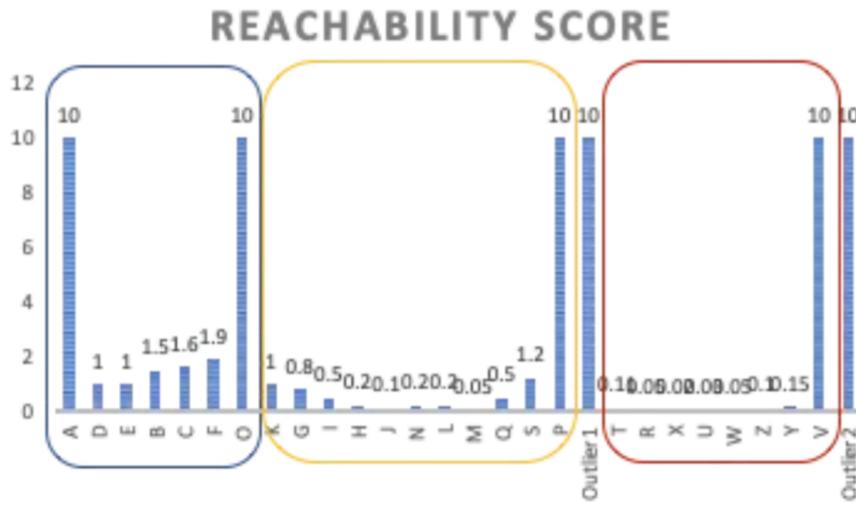


Figura 6.7: OPTICS: esempio di *reachability plot*. Le zone basse (valli) indicano cluster densi; le zone alte (creste) indicano separazioni. (immagine da libro/slide)

- applicando un *cut* orizzontale sul plot (equivalente a DBSCAN a un dato  $\varepsilon$ );
- individuando automaticamente valli significative (metodi di *valley picking* o soglie relative).

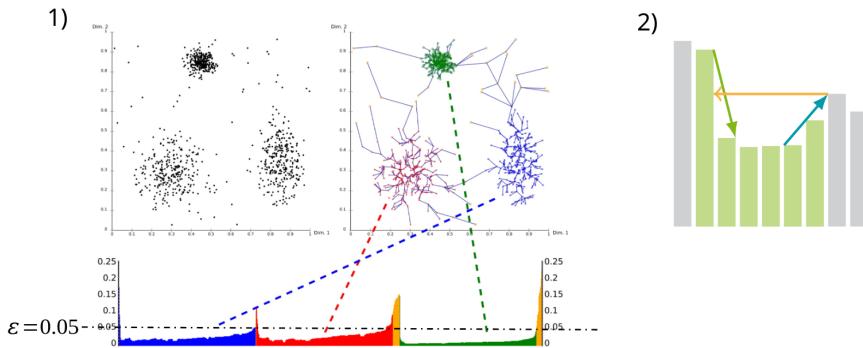


Figura 6.8: OPTICS. (1) A sinistra: punti nel piano e, sotto, *reachability plot*; le valli (segmenti colorati) corrispondono a regioni dense/cluster, mentre la linea orizzontale tratteggiata indica una soglia  $\varepsilon$  che produce un taglio in stile DBSCAN. Le linee tratteggiate collegano ogni gruppo nel piano al suo intervallo nel plot. (2) A destra: regola *steep down/up* per l'estrazione automatica dei cluster dal *reachability plot* (si entra quando la *reachability* scende bruscamente e si esce quando risale).

**Costo.** Con le adeguate strutture:  $O(n \log n)$ ; altrimenti  $O(n^2)$ .

### 6.4.3 HDBSCAN

**HDBSCAN** (Hierarchical DBSCAN) estende DBSCAN costruendo una *gerarchia di cluster per densità* e selezionando automaticamente i cluster più *stabili*. L'unico parametro concettuale è `MinPts` (soglia minima di densità); nelle librerie può comparire anche `min_cluster_size` (qui lo assimiliamo a `MinPts` come nelle slide).

**Idea.**

1. Definisco una distanza che “appiattisce” le regioni rade e rende confrontabili densità diverse.
2. Costruisco l’MST su tale distanza e, facendo crescere la densità minima  $\lambda = 1/\varepsilon$ , ottengo un *cluster tree*.
3. Condroso l’albero tenendo solo rami sostenuti da almeno `MinPts` punti e scelgo i cluster più *stabili*.

**Core distance di X.**  $\text{core\_dist}(p) = \text{distanza dal } \text{MinPts} \text{ più vicino.}$

**Distanza di mutual reachability.** Sia  $\text{core\_dist}(p)$  la distanza dal `MinPts` punto più vicino di  $p$ . La distanza

$$d_{\text{mreach}}(p, q) = \max \{ \text{core\_dist}(p), \text{core\_dist}(q), D(p, q) \}$$

“dilata” le regioni poco dense (aumentando le loro distanze interne) e *comprime* quelle dense: in questo modo cluster a densità diverse risultano separabili con un unico parametro.

**Mutual Reachability graph  $G_{\text{MinPts}}$ .** grafo pesato in cui i nodi sono gli oggetti dello spazio e, per ogni coppia di oggetti  $X, Y$ , viene inserito un arco il cui peso coincide con la Mutual Reachability distance tra  $X$  e  $Y$ .

#### Costruzione del dendogramma

Per la costruzione del dendogramma si procede così:

**Costruzione del MST** Si costruisce il Minimum Spanning Tree (MST) di  $G_{\text{MinPts}}$ , ovvero il sottografo aciclico di  $G$  il cui peso totale sugli archi è minimo e che garantisce la connessione di  $G$ .

**Rimozione degli archi** Si ordinano gli archi del MST in ordine decrescente di peso e si rimuovono progressivamente, partendo da quello più pesante. Ad ogni rimozione si ottiene una partizione dei punti in cluster (componenti connesse del grafo rimanente). Questa struttura, la chiamiamo *cluster tree*.

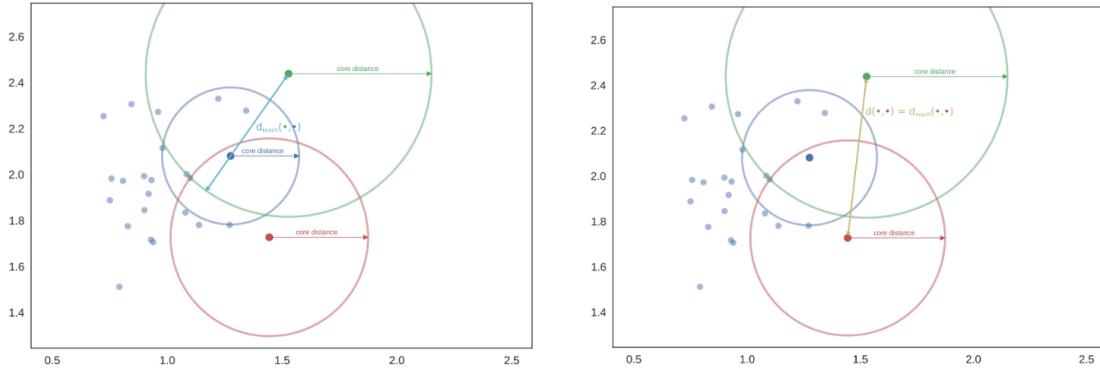


Figura 6.9: HDBSCAN: illustrazione di *core\_dist*, *reach\_dist* e *mutual reachability*. Ogni cerchio centrato in un punto ha raggio pari alla sua  $\text{core\_dist}(\cdot)$  (distanza dal MinPts-esimo vicino). **Sinistra:** per i punti  $p$  (verde) e  $o$  (blu) la distanza di raggiungibilità da  $p$  a  $o$  è  $\text{reach\_dist}(o \mid p) = \max\{\text{core\_dist}(p), D(p, o)\}$ ; poiché il raggio di  $p$  domina, vale  $\text{reach\_dist}(o \mid p) = \text{core\_dist}(p)$ . **Destra:** tra  $p$  (verde) e  $q$  (rosso) prevale la distanza euclidea, quindi la *mutual reachability* risulta  $d_{\text{mreach}}(p, q) = \max\{\text{core\_dist}(p), \text{core\_dist}(q), D(p, q)\} = D(p, q)$ . L'uso di  $d_{\text{mreach}}$  “gonfia” le regioni rade e rende più separabili cluster con densità diverse.

### Condensed tree e selezione dei cluster significativi

**Condensed tree.** A partire dal cluster tree si *condensa* mantenendo solo i rami che, per qualche intervallo di densità  $\lambda$ , hanno almeno `MinPts` punti (gli altri rami vengono collassati sul padre). Ogni nodo  $C$  del condensed tree porta:

$$\begin{aligned}\lambda_{\min}(C) &\quad (\text{densità alla nascita}) \\ \lambda_{\max}(C) &\quad (\text{densità alla scomparsa/split}) \\ |C|(\lambda) &\quad (\text{numero di punti in } C \text{ alla densità } \lambda)\end{aligned}$$

Qui  $|C|(\lambda)$  indica quanti punti appartengono al cluster  $C$  per una data densità  $\lambda$ ; questo valore viene usato per decidere se mantenere o collassare un ramo nel *condensed tree*.

**Stabilità (persistenza).** La stabilità misura “quanto a lungo” un cluster esiste al crescere di  $\lambda$ . Per un nodo  $C$  del condensed tree:

$$\text{stab}(C) = \sum_{p \in C} (\lambda_p^{\max}(C) - \lambda_p^{\min}(C)),$$

dove  $\lambda_p^{\max}(C)$  è la densità alla quale il punto  $p$  lascia  $C$  (per split o perché  $C$  muore). Equivalente: è l’“area sotto la curva” del numero di punti di  $C$  lungo  $\lambda$  nel condensed tree.

**Estrazione dei cluster significativi (dal condensed tree).** *Input:* albero condensato in cui ogni nodo  $C$  porta  $\lambda_{\text{birth}}(C)$ ,  $\lambda_{\text{death}}(C)$  e la funzione  $|C|(\lambda)$  (# punti vivi in  $C$  alla densità  $\lambda$ ).

- 1. Calcolo della stabilità (bottom-up).** Per ogni nodo  $C$  si valuta la *persistenza* del cluster lungo l'intervallo in cui esiste come:

$$\text{stab}(C) = \sum_{p \in C} (\lambda_p^{\max}(C) - \lambda_p^{\min}(C)),$$

dove  $\lambda^{\min}(C)$  è la densità alla *nascita* di  $C$  e  $\lambda_p^{\max}(C)$  è la densità alla quale il punto  $p$  *esce* da  $C$  (per split o scomparsa). Si procede dal basso verso l'alto in modo da avere già disponibili le stabilità dei figli quando si valuta il padre.

- 2. Regola di selezione (top-down, senza sovrapposizioni).** Visitando un nodo  $C$  con figli  $C_1, \dots, C_m$ , si confronta la stabilità del padre con la somma di quelle dei figli:

$$S_{\text{figli}} = \sum_{i=1}^m \text{stab}(C_i).$$

- Se  $\text{stab}(C) \geq S_{\text{figli}}$ : **seleziona**  $C$  come cluster e *non* scendere oltre in quel ramo.
- Altrimenti: **non** selezionare  $C$  e **scendere** ricorsivamente, applicando la stessa regola ai figli e selezionando quelli con stabilità  $> 0$ .

- 3. Output.** I nodi selezionati sono disgiunti e costituiscono i *cluster significativi* massimizzando la stabilità complessiva; in caso di parità si preferisce il padre.

**Assegnazione dei border (opzionale).** I punti non-core vicini a più cluster possono essere assegnati al cluster selezionato con  $\lambda_{\text{birth}}$  più basso o tramite un punteggio di *membership* proporzionale al tempo di permanenza del punto nel cluster nel condensed tree.

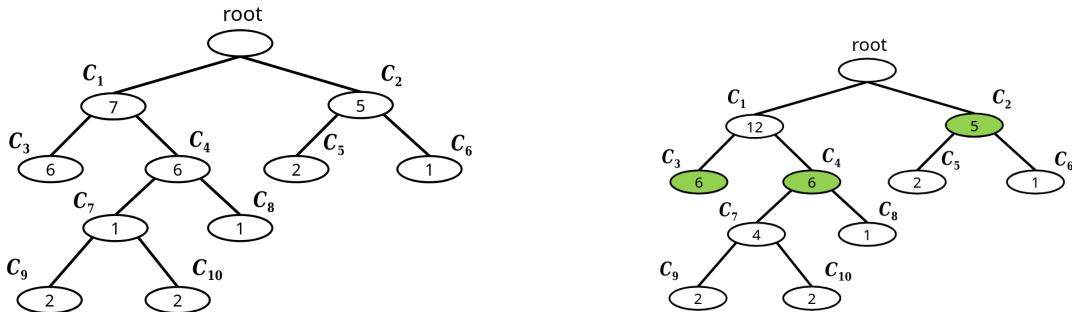


Figura 6.10: HDBSCAN. Sinistra: MST costruito sulla *mutual reachability distance* (archi più pesanti vengono tagliati al crescere di  $\lambda$ ). Destra: *condensed tree*; in evidenza i cluster scelti massimizzando la stabilità.

**Costo.** Dominato dalla costruzione del MST: con strutture adeguate  $O(n \log n)$ ; altrimenti  $O(n^2)$ .



# Capitolo 7

## Classificazione

### 7.1 Introduzione

La **classificazione** suddivide un insieme di dati in *classi* note a priori (etichette), apprendendo da esempi etichettati come assegnare la classe a nuove tuple. È quindi *apprendimento supervisionato*. Al contrario, il *clustering* non parte da etichette (*unsupervised*) e scopre gruppi per similarità.

**Predizione (regressione).** Quando il target è *numerico continuo*, il compito è di *predire* un valore reale (apprendimento supervisionato *continuo*), cercando una funzione che approssimi il target, non un confine tra classi.

#### 7.1.1 Schema generale di un classificatore

1. **Costruzione del modello** (training): si apprende da un *training set* etichettato.
2. **Validazione/valutazione** (test): si misura la bontà su un *test set* etichettato.
3. **Uso** (deploy): si applica il modello a nuove tuple per predirne la classe.

**Overfitting.** L'overfitting si verifica quando un modello “impara a memoria” il training, compreso il rumore: va molto bene sui dati visti ma generalizza male su dati nuovi. In pratica è un segnale che il modello è troppo complesso rispetto alle informazioni disponibili. Per ridurlo, si separano chiaramente i dati per la verifica e si preferiscono soluzioni più semplici quando offrono prestazioni simili.

#### 7.1.2 Requisiti desiderabili

- **Accuratezza:** corretta predizione delle classi (o del valore, per i predittori).
- **Velocità:** tempi di training e di classificazione contenuti.
- **Robustezza:** tolleranza a rumore e dati mancanti.
- **Scalabilità:** efficienza su dataset di grandi dimensioni.

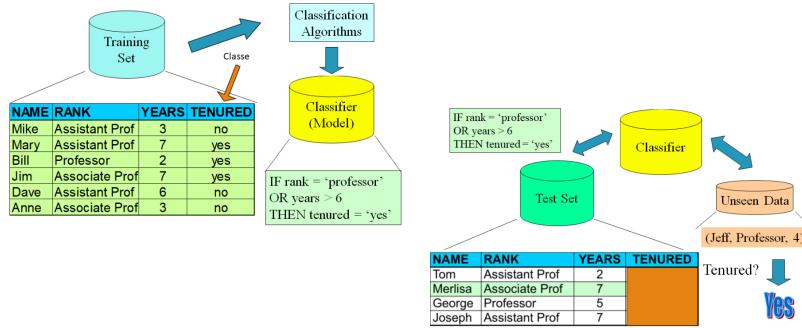


Figura 7.1: Schema a blocchi di un classificatore: addestramento, validazione e uso.

## 7.2 Alberi decisionali

Gli **alberi decisionali** classificano applicando test su attributi lungo i nodi interni; le *foglie* portano le etichette di classe.

### 7.2.1 Classificazione tramite albero

La classe di una tupla  $q$  si ottiene seguendo il cammino radice→foglia guidato dai test. Ogni cammino implementa una regola IF-THEN (le condizioni interne sono congiunte in AND). L'insieme di regole è *esaustivo* e *mutuamente esclusivo* (ogni tupla è coperta da una sola regola).

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

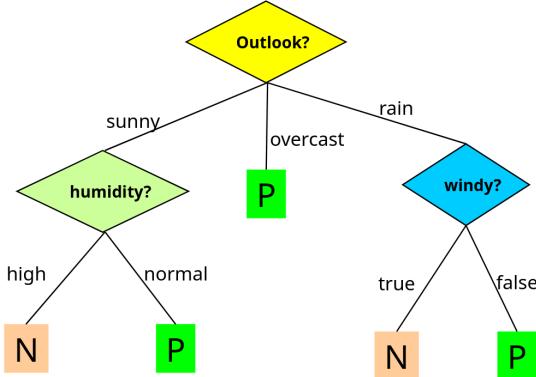


Figura 7.2: Dataset *weather* (a sinistra) e albero decisionale appreso (a destra). La tabella contiene 14 esempi con quattro attributi descrittivi (Outlook, Temperature, Humidity, Windy) e la classe binaria P/N. L'albero (stile ID3/C4.5) sceglie come radice Outlook; il ramo overcast porta direttamente 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.

### 7.2.2 Costruzione top-down

Costruzione ricorsiva dalla radice:

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.
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 7.2 con l'attributo "overcast").

### 7.2.3 Splitting degli attributi

- **Booleani/numerici:** split *binario* su soglia  $t$  (" $\leq t$ " a sinistra, " $> t$ " a destra).
- **Categoriali:** split *binario* definendo un sottoinsieme non vuoto di valori (a sinistra se il valore *non* appartiene al sottoinsieme, a destra altrimenti).

### 7.2.4 Scelta dell'attributo e strategia greedy

L'albero minimale è un problema *NP-hard*; si usa una strategia *greedy* che, ad ogni passo, seleziona l'attributo con massima *goodness* (partizioni più pure), costruendo l'albero "più compatto" possibile.

## 7.3 Misure di goodness

La scelta dell'attributo si basa su misure di *goodness*, che variano da algoritmo ad algoritmo.

### 7.3.1 Information Gain (ID3)

**Idea.** L'**Information gain** è un algoritmo che si basa sull'idea di selezionare l'attributo che massimizza la riduzione dell'entropia riguardo alla classe delle tuple dopo lo split. Questo perché:

- **Entropia massima:** si ha quando le classi sono equamente distribuite (massima incertezza).
- **Entropia minima:** si ha quando tutte le tuple appartengono alla stessa classe (certezza completa).

Sia  $S_X$  l'insieme di tuple al nodo  $X$ , con due classi  $P$  e  $N$ ; si indichino con  $p$  e  $n$  le rispettive numerosità. L'**entropia** di  $S_X$  è

$$H(S_X) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}.$$

Sia  $A$  un attributo con  $k$  valori distinti, che induce la partizione  $S_X \rightarrow S_1, \dots, S_k$ . Se  $S_i$  contiene  $p_i$  e  $n_i$  elementi, allora

$$H(S_i) = -\frac{p_i}{|S_i|} \log_2 \frac{p_i}{|S_i|} - \frac{n_i}{|S_i|} \log_2 \frac{n_i}{|S_i|}$$

Possiamo anche calcolare l'**entropia media** dopo lo split su  $A$ :

$$\overline{H}_A(S_X) = \sum_{i=1}^k \frac{|S_i|}{|S_X|} H(S_i).$$

L'**information gain** è definito come la riduzione di entropia ottenuta dal partizionamento  $S_x$  scegliendo l'attributo  $A$ :

$$\text{Gain}(S_X, A) = H(S_X) - \overline{H}_A(S_X).$$

Si sceglie l'attributo con gain massimo.

### Esempio e limitazioni

Sul dataset “weather” (Fig. 7.2) si ottengono:

$$\text{Gain}(outlook) = 0.246, \quad \text{Gain}(temperature) = 0.029, \quad \text{Gain}(humidity) = 0.151, \quad \text{Gain}(windy) = 0.048.$$

**Limite noto.** L'Information Gain è *sbilanciato* verso attributi con molti valori: un attributo quasi univoco (es. ID) produce molte partizioni piccole (foglie pure), abbattendo l'entropia media e gonfiando artificialmente , pur senza reale capacità predittiva.

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

L'attributo *outlook* è scelto come radice:

$$\text{gain(outlook)} = 0.246$$

$$\text{gain(temperature)} = 0.029$$

$$\text{gain(humidity)} = 0.151$$

$$\text{gain(windy)} = 0.048$$

Figura 7.3: Esempio di scelta della radice con Information Gain sul dataset “weather”.

### 7.3.2 Gain Ratio (C4.5)

Il **Gain Ratio** corregge il bias dell'Information Gain verso attributi con molti valori introducendo la *split information*, che misura quanta informazione è generata dal solo atto di partizionare i dati secondo l'attributo (indipendentemente dalla classe).

Sia  $S$  l'insieme di tuple nel nodo corrente e  $A$  un attributo che induce la partizione  $S = S_1 \cup \dots \cup S_k$ . Definiamo

$$\text{SplitInfo}(A, S) = - \sum_{i=1}^k \frac{|S_i|}{|S|} \log_2 \left( \frac{|S_i|}{|S|} \right), \quad \text{Gain}(A, S) = H(S) - \sum_{i=1}^k \frac{|S_i|}{|S|} H(S_i).$$

Il **Gain Ratio** è

$$\text{GR}(A, S) = \frac{\text{Gain}(A, S)}{\text{SplitInfo}(A, S)}.$$

**Selezione in C4.5.** Per evitare divisioni spurie quando SplitInfo è piccola, C4.5 sceglie l'attributo con GR massimo tra quelli con *Gain* non inferiore (ad es.) al gain medio del nodo. In pratica:

1. calcola  $\text{Gain}(A, S)$  e scarta attributi con  $\text{gain} \leq 0$ ;
2. tra i rimanenti, seleziona l'attributo con GR più alto.

**Nota pratica (attributi continui).** Per un attributo numerico  $A$  si ordinano i valori e si valutano soglie candidate  $t$  nelle posizioni fra due valori consecutivi:  $A \leq t$  vs  $A > t$ . Per ogni soglia si calcolano gain e gain ratio; si sceglie la soglia che massimizza la metrica.

### 7.3.3 Gini Index (CART)

Sia  $i$  una classe e  $T$  una tupla di classe  $i$  scelta a caso da  $S_x$ . Per ricavare il **Gini Index** si calcola la probabilità che  $T$  venga classificata erroneamente, ovvero che appartenga a una classe diversa da  $i$ : e quindi occorre considerare:

- **Probabilità che  $T$  sia di classe  $i$ :**  $P(i | S_X)$ .
- **Probabilità che  $T$  sia di una classe diversa da  $i$ :**  $1 - P(i | S_X)$ .

Dato che il ragionamento fatto vale per ogni classe, si sommano le probabilità di errore su tutte le classi:

$$\text{Gini}(S_X) = \sum_{i=1}^n p_i(1 - p_i) = \sum_{i=1}^n (p_i - p_i^2) = \sum_{i=1}^n p_i - \sum_{i=1}^n p_i^2 = 1 - \sum_{i=1}^n p_i^2$$

Con la supposizione che  $S_x$  contenga  $k$  classi e che  $p_i$  sia la probabilità che una tupla scelta a caso da  $S_X$  appartenga alla classe  $i$ , si ha che il **Gini Index** dello split è definito come:

$$\text{Gini}_{\text{split}}(S_X) = 1 - \sum_{i=1}^k \frac{|S_i|}{|S_x|} \text{Gini}(S_i)$$

CART seleziona l'attributo/soglia che *minimizza* GiniSplit. Su attributi categoriali si cercano partizioni in due sottoinsiemi di valori; su continui, soglie come in C4.5.

### 7.3.4 Pruning degli alberi

Alberi molto profondi generalizzano male (generano *overfitting*). Per evitare questo, si effettua un **pruning**, ovvero si riduce la dimensione dell'albero sostituendo un sottoalbero con una foglia etichettato con la classe maggioritaria delle tuple nel sottoalbero, il pruning inserisce però un tasso di errore, si fa solo se necessario. Si usano due strategie principali:

**Pre-pruning** in fase di costruzione dell'albero si interrompe la crescita quando la goodness dello split è al di sopra di una *soglia*.

**Post-pruning** si costruisce l'albero completo e poi lo si riduce valutando l'errore su validation set o tramite stima incrociata. Generalmente è più dispensioso ma più efficace.

### Pruning pessimistico (C4.5)

Confronta l'errore stimato del *sottoalbero*  $T$  radicato in  $X$  con l'errore stimato della *foglia* che sostituisce  $T$  (classe maggioritaria in  $X$ ).

Sia  $X$  un nodo dell'albero con insieme di esempi  $S_x$  ( $N = |S_x|$ ) e classe di maggioranza  $C$ . Sia  $T$  il sottoalbero radicato in  $X$  e siano  $x_1, \dots, x_k$  i figli immediati di  $X$ , con  $S_{x_i}$  gli esempi nel figlio  $x_i$  e  $C_i$  la sua classe di maggioranza. Le due quantità

$$E_p(T) = \frac{|\{t \in S_x \mid \text{class}(t) \neq C\}| + \epsilon}{|S_x|}$$

$$E'_p(T) = \frac{\sum_{i=1}^k |\{t \in S_{x_i} \mid \text{class}(t) \neq C_i\}| + k\epsilon}{|S_x|}$$

sono le **stime del tasso di errore** usate per decidere se fare pruning.

#### Che cosa misurano.

- $E_p(T)$  è l'*errore stimato* se **si pota**  $T$  sostituendo l'intero sottoalbero con *una sola foglia* etichettata con la classe di maggioranza  $C$  del nodo  $X$ . Il numeratore conta le istanze di  $S_x$  che verrebbero sbagliate da tale foglia, con una *correzione*  $\epsilon$  (tipicamente  $\epsilon = \frac{1}{2}$ ) per evitare stime troppo ottimistiche su campioni piccoli.
- $E'_p(T)$  è l'*errore stimato* se **si mantiene lo split corrente** di  $X$  nei suoi  $k$  figli, ma *troncando* ognuno di essi a foglia (ognuna etichettata con la propria maggioranza  $C_i$ ). Si sommano gli errori dei  $k$  figli e si aggiunge una correzione  $\epsilon$  per *ciascuna* foglia ( $k\epsilon$ ).

**Decisione di pruning.** Confrontando  $E_p(T)$  ed  $E'_p(T)$ :

- per  $E_p(T) \leq E'_p(T)$ , *potare* il nodo  $X$  è preferibile, poiché l'errore stimato come foglia è minore o uguale a quello del sottoalbero.
- per  $E_p(T) > E'_p(T)$ , conviene *mantenere* lo split, in quanto l'errore stimato del sottoalbero è inferiore a quello della singola foglia.

Il valore  $\epsilon$  è una sorta di “costo fisso” per ogni foglia aggiunta all'albero. L'aggiunta di questo valore agisce da *regolarizzatore*: penalizza strutture con molte foglie, evitando che piccole fluttuazioni del campione giustifichino split inutili.

### Cost-complexity pruning (CART)

Si valuta il vantaggio dello *split* di un nodo  $X$  confrontando la riduzione di *error rate* con l'aumento di complessità (nuove foglie).

**Errore prima e dopo lo split.** Sia  $S_X$  l'insieme dei campioni che arrivano al nodo  $X$  e  $C$  la classe maggioritaria in  $S_X$ .

$$E(X) = \frac{|\{t \in S_X : \text{class}(t) \neq C\}|}{|S_X|}.$$

Se  $X$  viene diviso in  $k$  figli  $X_1, \dots, X_k$  (con classi maggioritarie  $C_1, \dots, C_k$ ), l'errore *atteso dopo lo split* è

$$E'(X) = \frac{\sum_{i=1}^k |\{t \in S_{X_i} : \text{class}(t) \neq C_i\}|}{|S_X|}.$$

**Indice di costo–complessità per lo split.** Definiamo il guadagno medio per foglia aggiunta:

$$\alpha(X) = \frac{E(X) - E'(X)}{k - 1}.$$

Il valore  $\alpha$  misura *quanto* diminuisce l'errore per ogni foglia extra introdotta dallo split. Se  $\alpha$  è sufficientemente piccolo, ovvero quando  $\alpha$  è minore di una soglia prefissata  $\alpha_0$ , lo split non è conveniente e si pota il nodo  $X$ .

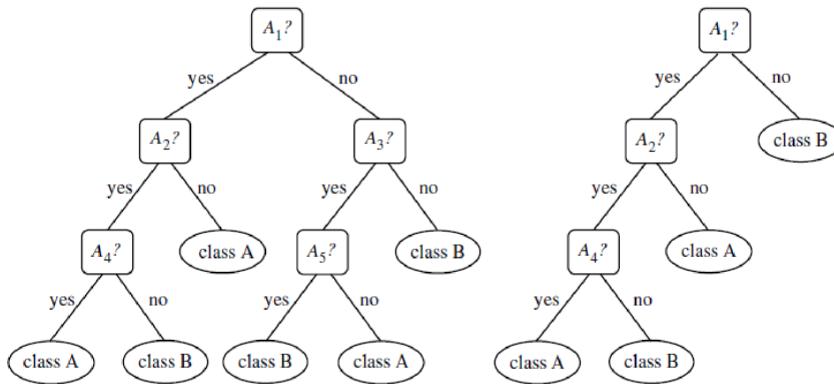


Figura 7.4: Pruning: confronto errore stimato del sottoalbero vs foglia (C4.5) e principio costo–complessità (CART).

**Pro/contro degli alberi decisionali.** *Pro:* interpretabili, veloci in predizione, gestiscono mix di attributi (continui/categoriali), poca preparazione dei dati. *Contro:* instabili rispetto a piccole variazioni dei dati, propensi all'overfitting, separazioni per soglie assiali (forme complesse richiedono molti nodi), accuratezza spesso inferiore a ensemble o SVM su dati ad alta dimensionalità.

## 7.4 Classificatori generativi

I modelli generativi producono un **modello probabilistico** a partire dai dati, predicendo la **classe** di appartenenza *più probabile* per un nuovo dato a partire dal modello sviluppato. Questi modelli si basano sul **teorema di Bayes**.

### 7.4.1 Teorema di Bayes e regola di decisione

Sia  $\mathbf{x}$  un'osservazione e  $c \in \mathcal{C}$  una classe candidata. Per decidere la classe usiamo il **teorema di Bayes**:

$$P(c | \mathbf{x}) = \frac{P(\mathbf{x} | c) P(c)}{P(\mathbf{x})}.$$

- **Probabilità a priori**  $P(c)$ : quanto la classe  $c$  è probabile *prima* di vedere i dati (in pratica: frequenza della classe nel train).
- **Likelihood**  $P(\mathbf{x} | c)$ : quanto è plausibile osservare  $\mathbf{x}$  se la classe fosse  $c$ .
- **Evidenza**  $P(\mathbf{x})$ : probabilità complessiva di osservare  $\mathbf{x}$  (uguale per tutte le classi).

La decisione ottima *MAP* (Maximum A Posteriori) è

$$\hat{c}(\mathbf{x}) = \arg \max_{c \in \mathcal{C}} P(c | \mathbf{x}) = \arg \max_{c \in \mathcal{C}} P(\mathbf{x} | c) P(c),$$

poiché  $P(\mathbf{x})$  non dipende da  $c$  e non influisce sull'arg max.

### 7.4.2 Naive Bayes

**Idea.** Assumiamo che, fissata la classe  $c$ , le feature siano indipendenti (*assunzione naive*). Allora la verosimiglianza fattorizza:

$$P(\mathbf{x} | c) = \prod_{j=1}^d P(x_j | c).$$

**Regola di decisione (MAP, in scala logaritmica).** Le probabilità condizionali sono molto piccole e un prodotto di tante quantità prossime a 0 può portare problemi di underflow. Per ovviare a questi problemi si considera il **log-likelihood**:

$$\hat{c}(\mathbf{x}) = \arg \max_{c \in \mathcal{C}} \left[ \log P(c) + \sum_{j=1}^d \log P(x_j | c) \right].$$

Questo si traduce in una somma anziché in un prodotto di termini.

#### Stima essenziale delle probabilità.

- **Prior**  $P(c)$ : frequenza della classe nel training.
- **Attributi discreti**: frequenze condizionate con *Laplace smoothing* ( $+\alpha$ ) per evitare zeri.
- **Attributi continui**: modello gaussiano per  $x_j | c$  con media e varianza stimate sui dati della classe.

**Vantaggi e svantaggi.** Molto veloce e facile da implementare, l'assunzione di indipendenza condizionale potrebbe non essere sempre vera e potrebbe portare ad una perdita di accuratezza (tali dipendenze non possono essere modellate da questo modello).

### 7.4.3 Reti Bayesiane

Una **rete bayesiana** è un DAG le cui variabili  $\{X_1, \dots, X_d\}$  fattorizzano come

$$P(X_1, \dots, X_d) = \prod_{i=1}^d P(X_i | \text{Pa}(X_i)),$$

dove  $\text{Pa}(X_i)$  sono i genitori di  $X_i$  nel grafo. Il DAG codifica indipendenze condizionali; i CPT (tabelle di probabilità condizionate) specificano i parametri.

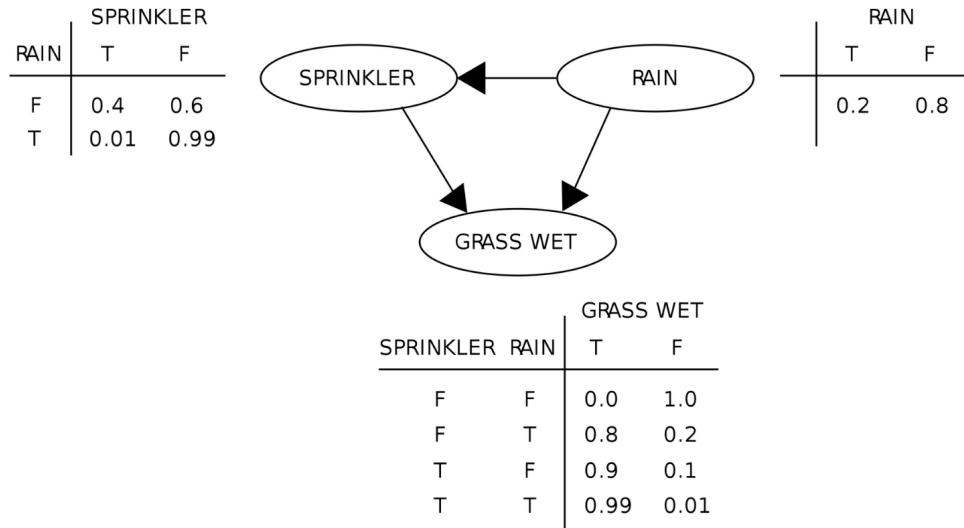


Figura 7.5: Rete bayesiana *Sprinkler–Rain–Grass*: il DAG orientato specifica le dipendenze e indetermina la fattorizzazione della congiunta  $P(\text{Rain}) P(\text{Sprinkler} | \text{Rain}) P(\text{GrassWet} | \text{Sprinkler}, \text{Rain})$ . Le tabelle mostrano le CPD (Conditional Probability Tables) dei nodi. A differenza di Naive Bayes, le feature possono essere dipendenti dato la/e causa/e (qui Rain), e tale dipendenza è resa esplicita dagli archi.

**Uso per la classificazione.** Dato  $\mathbf{x}$ , si calcolano (o si approssimano)  $P(y | \mathbf{x})$  tramite inferenza sul DAG (*esatta* o *approx* con sampling/variational). Le reti bayesiane generalizzano Naive Bayes (che è un caso particolare con  $Y$  genitore di tutte le feature e nessun’altra dipendenza).

## 7.5 Classificatori discriminativi

I classificatori discriminativi stimano direttamente una funzione di decisione  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  (o, optionalmente, la probabilità condizionata  $P(y | \mathbf{x})$ ) senza modellare la distribuzione congiunta  $P(\mathbf{x}, y)$ . Dato un nuovo esempio  $\mathbf{x}$  si valuta  $f(\mathbf{x})$  e si assegna l’etichetta corrispondente. Rispetto ai modelli generativi richiedono generalmente meno assunzioni sui dati. Tipici esempi sono il Perceptron e le Support Vector Machines (SVM).

### 7.5.1 Classificazione lineare e non lineare

**Lineare:** la regola di decisione è basata su una combinazione lineare degli attributi

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b,$$

Dove  $\mathbf{w} \in \mathbb{R}^d$  è il vettore dei pesi e  $b \in \mathbb{R}$  è il bias (termine di soglia).

**Non lineare:** Il problema di avere spazi non lineari è che non è possibile separare le classi con un iperpiano. Per risolvere questo problema si fa una trasformazioni di spazi vettoriali in spazi di dimensione superiore dove la separazione lineare è possibile. Questo si ottiene tramite il **kernel trick**, che permette di calcolare prodotti scalari in spazi trasformati senza dover esplicitamente mappare i dati.

*Esistono anche classificazioni lineari e non lineari binarie, dove si utilizzano approcci simili.*

### 7.5.2 Perceptron

**Definizione.** Il Perceptron è un classificatore lineare binario che produce la funzione

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b.$$

Con etichette  $y \in \{-1, +1\}$ . Si parla di un algoritmo di machine learning, quindi è necessario un modo, per il perceptron, di apprendere i parametri  $\mathbf{w}$  e  $b$  dai dati di addestramento.

**Regola di aggiornamento.** Dato un esempio  $(\mathbf{x}^{(i)}, y^{(i)})$ , se la previsione  $\hat{y}^{(i)}$  è errata (ossia  $\hat{y}^{(i)} \neq y^{(i)}$ ) si aggiorna ogni peso component-wise secondo la notazione usata in figura:

$$\hat{\vartheta}_j \leftarrow \vartheta_j + \alpha (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}, \quad j = 0, \dots, d,$$

dove  $\alpha > 0$  è il learning rate,  $\vartheta_j$  indica il valore corrente del peso e  $\hat{\vartheta}_j$  il valore aggiornato. Equivalentemente, in forma vettoriale si ottiene

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha (y^{(i)} - \hat{y}^{(i)}) \mathbf{x}^{(i)}, \quad b \leftarrow b + \alpha (y^{(i)} - \hat{y}^{(i)}).$$

Se  $\hat{y}^{(i)} = y^{(i)}$  il modello non viene modificato.

**Proprietà.** Se i dati sono linearmente separabili, il Perceptron converge in un numero finito di aggiornamenti (teorema di Novikoff). In pratica si itera per più epoch o fino a soddisfare un criterio di stop.

**Algoritmo.** L'algoritmo del perceptron è definito come:

1. Inizializza i pesi  $\mathbf{w}$  e il bias  $b$  a zero o a valori casuali.
2. Per ogni tupla  $y_j$  nel training set:
  - (a) Calcola la previsione  $\hat{y}^{(i)}$

- (b) Se  $\hat{y}^{(i)} \neq y^{(i)}$ , aggiorna i pesi e il bias secondo la regola di aggiornamento.
  - (c) Incrementa  $i$  e ripeti fino a completare il training set.
3. Ripeti il passo 2 per un numero prefissato di epoche o fino a soddisfare un criterio di stop. Nel caso di learning offline, si ripete il passo 2 finché l'errore medio di classificazione sul training set non scende sotto una soglia prefissata.

**One–Vs–One (OVO).** One–Vs–One costruisce un classificatore binario Perceptron per ogni coppia di classi  $(C_p, C_q)$ ; per  $K$  classi si addestrano  $K(K - 1)/2$  modelli. In fase di predizione, ogni classificatore vota per una delle due classi che confronta e si assegna la classe con il maggior numero di voti (voting).

Motivazione: OVO è utile quando le classi sono relativamente poche e si desidera che ogni modello risolva una decisione binaria semplice; ogni modello vede dati di due sole classi, spesso permettendo separazioni più semplici e modelli più piccoli. Lo svantaggio principale è il numero di classificatori e la gestione del voto/pareggio.

Algoritmo (per ciascuna coppia  $(C_p, C_q)$ ):

1. Costruisci il training set rimuovendo istanze non appartenenti a  $C_p$  o  $C_q$ .
2. Inizializza i pesi  $\vartheta_j$  e il bias.
3. Per ogni epoca e per ogni esempio  $(\mathbf{x}^{(i)}, y^{(i)})$  nel sottoinsieme: calcola  $\hat{y}^{(i)} = \text{sign}(f(\mathbf{x}^{(i)}))$ ; se  $\hat{y}^{(i)} \neq y^{(i)}$  aggiorna

$$\hat{\vartheta}_j \leftarrow \vartheta_j + \alpha (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} \quad (j = 0, \dots, d).$$

**One–Vs–All (OVA).** One–Vs–All costruisce un classificatore Perceptron per ogni classe  $C_k$  dove il problema è  $C_k$  vs "resto". Si addestrano  $K$  modelli; alla predizione si calcola il punteggio  $f_k(\mathbf{x})$  per ogni modello e si sceglie la classe con il punteggio più alto.

Motivazione: OVA è più parsimonioso in termini di numero di modelli rispetto a OVO (si addestrano  $K$  modelli invece di  $K(K - 1)/2$ ) e può essere più efficiente quando  $K$  è grande. Tuttavia ogni modello OVA affronta un problema sbilanciato (una classe vs tutte le altre), il che può richiedere tecniche di bilanciamento o regolarizzazione.

Algoritmo (per ciascuna classe  $C_k$ ):

1. Crea etichette binarie  $y^{(i)} = +1$  se l'esempio appartiene a  $C_k$ , altrimenti  $y^{(i)} = -1$ .
2. Inizializza i pesi  $\vartheta_j$  e il bias.
3. Per ogni epoca e per ogni esempio  $(\mathbf{x}^{(i)}, y^{(i)})$ : calcola  $\hat{y}^{(i)} = \text{sign}(f(\mathbf{x}^{(i)}))$ ; se  $\hat{y}^{(i)} \neq y^{(i)}$  aggiorna

$$\hat{\vartheta}_j \leftarrow \vartheta_j + \alpha (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} \quad (j = 0, \dots, d).$$

*Breve nota comparativa:* OVO tende a produrre modelli più specialistici e può funzionare meglio quando le classi sono ben separate a coppie; OVA è più semplice ed efficiente per molti problemi pratici ma richiede attenzione allo sbilanciamento delle classi.

### 7.5.3 Support Vector Machines (SVM)

Le SVM hanno un'idea diversa dal perceptron: cercano di trovare l'iperpiano che massimizza il **margine** tra le classi, ovvero la distanza minima tra l'iperpiano e i punti dati più vicini di ciascuna classe (i *support vectors*).

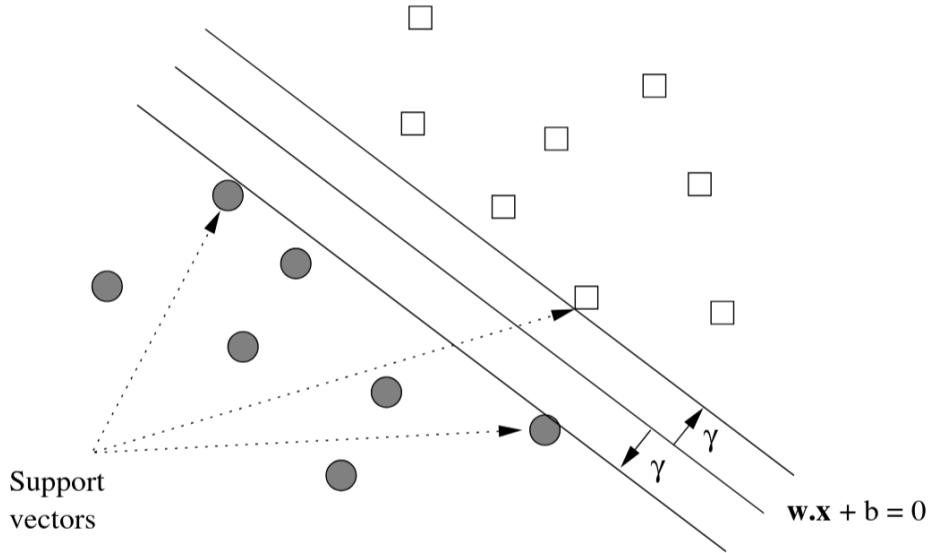


Figura 7.6: Esempio di Support Vector Machine che massimizza il margine tra due classi. I punti cerchiati sono i support vectors che definiscono l'iperpiano ottimale.

Per dati linearmente separabili, esistono infinite iperpiani che separano le classi; le SVM scelgono quello con il margine massimo, che tende a generalizzare meglio su dati non visti.

**Formulazione del problema.** Sia  $D$  un dataset di addestramento formato da punti  $(\mathbf{x}^{(i)}, y^{(i)})$ , con etichette  $y^{(i)} \in \{-1, +1\}$ , pesi  $\mathbf{w}$  e bias  $b$ . L'iperpiano di decisione è definito da:

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b.$$

Se indichiamo con  $\gamma$  il margine, l'obiettivo delle SVM è trovare i parametri  $w, b$  che massimizzano  $\gamma$ . Indicando con

$$H_1 : \mathbf{w} \cdot \mathbf{x} + b = \gamma$$

$$H_2 : \mathbf{w} \cdot \mathbf{x} + b = -\gamma$$

le equazioni dei due iperpiani paralleli che definiscono il margine, passanti per i vettori di supporto delle due classi. Da questo si può dedurre che i punti di classe  $+1$  soddisfano la diseguaglianza  $\mathbf{w} \cdot \mathbf{x}^{(i)} + b \geq \gamma$ , mentre i punti di classe  $-1$  soddisfano  $\mathbf{w} \cdot \mathbf{x}^{(i)} + b \leq -\gamma$ .

Combinando queste due condizioni, si ottiene la seguente diseguaglianza per tutti i punti del dataset:

$$y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \geq \gamma, \quad \forall i.$$

Poiché  $y^{(i)}$  può essere  $+1$  o  $-1$ , questa condizione assicura che ogni punto sia correttamente classificato e si trovi al di fuori del margine. Tuttavia si può riscrivere la formulazione in modo più semplice, fissando  $\gamma = 1$ , il che porta alla condizione:

$$y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \geq 1, \quad \forall i.$$

Purtroppo questa formulazione "libera" in termini di  $\gamma$  non è direttamente utilizzabile: se  $(\mathbf{w}, b)$  soddisfa i vincoli, allora anche  $(\lambda\mathbf{w}, \lambda b)$  soddisfa i vincoli per ogni scalare  $\lambda > 1$  e fornisce un margine  $\lambda\gamma$  più grande. Di conseguenza non esiste un massimo finito per  $\gamma$  senza imporre una normalizzazione aggiuntiva. Per evitare questa degenerazione si fissa implicitamente la scala del vettore dei pesi fissando il valore del margine (ad es.  $\gamma = 1$ ) e si minimizza la norma di  $\mathbf{w}$ , che è equivalente a massimizzare il margine in modo ben definito. La soluzione del problema è quella di normalizzare il vettore dei pesi in modo che il margine sia fissato a 1, quindi trasformare il vettore  $\mathbf{w}$  in un vettore:

$$\hat{\mathbf{w}} = \frac{\mathbf{w}}{\|\mathbf{w}\|}.$$

Il vettore  $\hat{\mathbf{w}}$  è il vettore unità (modulo pari a 1) dei pesi normalizzato con stessa direzione di  $\mathbf{w}$ . Inoltre, poiché ha la stessa direzione è perpendicolare all'iperpiano separatore:

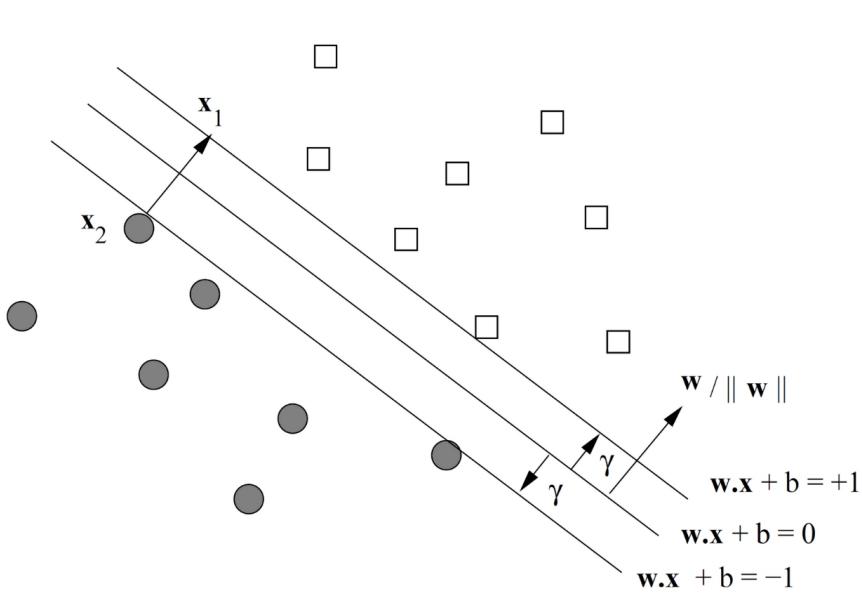


Figura 7.7: Iperpiani con pesi  $\mathbf{w}$  normalizzati per margine unitario.

Indicando con  $x_2$  un vettore di supporto che giace su  $H_2$  e  $x_1$  la sua proiezione in  $H_1$  (figura 7.7), la distanza tra i due iperpiani è data da:

$$x_1 = x_2 + 2\gamma = x_2 + 2\gamma \frac{\mathbf{w}}{\|\mathbf{w}\|}$$

Possiamo ricalcolare l'equazione dell'iperpiano come:

$$\begin{aligned} \mathbf{w} \cdot (x_2 + 2\gamma \frac{\mathbf{w}}{\|\mathbf{w}\|}) + b &= 0 \\ \Rightarrow \mathbf{w} \cdot x_2 + b + 2\gamma \|\mathbf{w}\| &= 0 \end{aligned}$$

Osservando che  $x_2$  giace su  $H_2$  abbiamo  $\mathbf{w} \cdot x_2 + b = -\gamma$ . Sostituendo nell'ultima equazione:

$$-\gamma + 2\gamma \|\mathbf{w}\| = 0 \implies 2\gamma \|\mathbf{w}\| = \gamma.$$

Se  $\gamma \neq 0$  segue

$$\|\mathbf{w}\| = \frac{1}{2},$$

ma ricordando che nella definizione del margine la distanza tra le due hyperplane considerate è  $2\gamma$ , fissando la normalizzazione standard si ricava l'identità

$$\gamma = \frac{1}{\|\mathbf{w}\|},$$

da cui la conclusione voluta: massimizzare  $\gamma$  equivale a minimizzare  $\|\mathbf{w}\|$  (e quindi, per praticità numerica, si minimizza spesso  $\frac{1}{2}\|\mathbf{w}\|^2$ ).

Il problema di ottimizzazione da risolvere, quindi, è:

$$\begin{cases} \min \|\mathbf{w}\| \\ y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \forall 1 \leq i \leq n \end{cases}$$

che si può risolvere applicando il metodo di discesa del gradiente.

Nella pratica, per rendere più semplice il calcolo dei gradienti si preferisce considerare il quadrato della norma:

$$\begin{cases} \min \frac{1}{2} \|\mathbf{w}\|^2 \\ y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \forall 1 \leq i \leq n \end{cases}$$

**SVM Soft margin.** Esiste un problema nella classificazione di SVM chiamato **Hard Margin**: ovvero una classificazione dei dati troppo precisa, in quanto non può ricadere all'interno dell'intervallo dei due iperpiani. Una variante si chiama **Soft Margin**, che permette ad alcuni punti di cadere all'interno del margine o addirittura di essere classificati in modo errato, introducendo delle variabili di slack  $\xi_i \geq 0$  per ogni punto dati.

Per riformulare il problema, si introducono  $n$  variabili  $\epsilon_1, \epsilon_2, \dots, \epsilon_n$  tale che, quando  $\epsilon_i = 0$ , il punto ricade al di fuori del margine e soddisfa la relazione:

$$y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \geq 1,$$

Altrimenti se  $\epsilon_i > 0$ , il punto ricade all'interno del margine o è classificato in modo errato,

e la relazione diventa:

$$y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \geq 1 - \epsilon_i.$$

Da qui si introduce un parametro  $\lambda$  di **trade-off**<sup>1</sup> tra la minimizzazione della norma di  $\mathbf{w}$  e la penalizzazione degli errori di classificazione, portando alla seguente formulazione del problema di ottimizzazione:

$$\begin{cases} \min \left( \frac{1}{n} \sum_{i=1}^n \epsilon_i + \lambda \|\mathbf{w}\|^2 \right) \\ y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \geq 1 - \epsilon_i, \quad \forall i = 1, \dots, n \\ \epsilon_i \geq 0, \quad \forall i = 1, \dots, n \end{cases}$$

Più piccolo è  $\lambda$ , più trascurabile è  $\|\mathbf{w}\|$ , meno importante è la dimensione del margine.

Il problema può essere risolto anche considerando il suo duale tramite la Lagrangiana. Introducendo i moltiplicatori di Lagrange  $\alpha_i$  per i vincoli di classificazione (e  $\mu_i$  per i vincoli  $\epsilon_i \geq 0$ ) si ottiene una funzione quadratica in  $\alpha_i$ , quindi il duale è una QP risolvibile con algoritmi dedicati. Ricavati i moltiplicatori  $\alpha_i$ , il vettore dei pesi si esprime come

$$\mathbf{w} = \sum_{i=1}^n \alpha_i y^{(i)} \mathbf{x}^{(i)}.$$

Il bias  $b$  si ricava:

$$y^{(i)}(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) = 1 \implies b = \frac{1}{y^{(i)}} - \mathbf{w} \cdot \mathbf{x}^{(i)}.$$

Nel caso di dati non linearmente separabili si effettua un *mapping* in uno spazio di dimensione superiore tramite una funzione  $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^D$  (con  $D > d$ ) e si applica SVM in tale spazio. Quindi, creiamo un nuovo problema di ottimizzazione dove al posto di  $\mathbf{x}^{(i)}$  usiamo  $\phi(\mathbf{x}^{(i)})$ :

$$\begin{cases} \max \left( \sum_{i=1}^n c_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n y^{(i)} c_i (\phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)})) y^{(j)} c_j \right) \\ \sum_{i=1}^n c_i y^{(i)} = 0 \\ 0 \leq c_i \leq \frac{1}{2n\lambda}, \quad i = 1, \dots, n \end{cases}$$

**Kernel Trick.** Il prodotto scalare  $\phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)})$  è **dispendioso** da calcolare. Per questo motivo, si usa una **funzione kernel**  $K$  tale che soddisfi tale condizione:

$$K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)}) \cdot \phi(\mathbf{x}^{(j)}).$$

---

<sup>1</sup>Il parametro di trade-off  $\lambda$  bilancia l'importanza tra la massimizzazione del margine e la minimizzazione dell'errore di classificazione. Un valore più alto di  $\lambda$  dà più peso alla riduzione degli errori, mentre un valore più basso enfatizza la massimizzazione del margine.

Grazie a questa funzione, si definisce il mapping del nuovo spazio permettendo di sostituire il prodotto scalare con la funzione kernel, risparmiando tempo di calcolo (esempi di kernel sono il kernel polinomiale, gaussiano e sigmoide).

**SVM Multi-classe.** SVM, come il perceptron, può essere esteso a problemi di classificazione multi-classe usando strategie come One-Vs-One (OVO) o One-Vs-All (OVA), addestrando più modelli binari e combinando le loro predizioni per ottenere la classificazione finale.

## 7.6 Apprendimento Lazy

Fino a qui abbiamo visto tipi di apprendimento chiamati **eager**, in cui il modello viene costruito durante la fase di training. Esiste un secondo tipo di apprendimento chiamato **lazy**, dove il modello memorizza un training set di dati e calcola la funzione di classificazione localmente per ogni nuovo dato.

La funzione di predizione è quindi approssimata *localmente*, inoltre questi metodi funzionano su grandi dataset con pochi attributi e che si aggiornano continuamente.

### 7.6.1 K-Nearest Neighbor

K-Nearest Neighbor (KNN) è un algoritmo di classificazione lazy che assegna la classe di un nuovo esempio basandosi sulle classi dei suoi  $k$  vicini più prossimi nel training set.

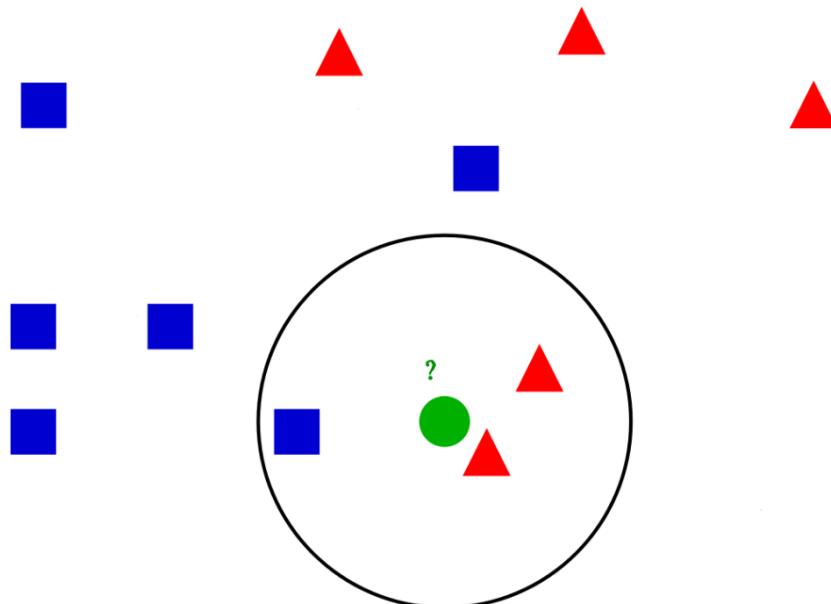


Figura 7.8: Esempio di classificazione con K-NN ( $k=5$ ). Il punto da classificare (in verde) è circondato dal cerchio che indica i  $k$  vicini più prossimi: tra i 5 vicini ci sono tre triangoli rossi e due quadrati blu, quindi per maggioranza il punto viene assegnato alla classe rossa.

**Algoritmo.**

1. Memorizza il training set  $D = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$ .
2. Per ogni nuovo esempio  $\mathbf{x}$ :
  - (a) Calcola la distanza tra  $\mathbf{x}$  e ogni esempio in  $D$  (es. distanza euclidea).
  - (b) Identifica i  $k$  esempi più vicini.
  - (c) Assegna a  $\mathbf{x}$  la classe più frequente tra i  $k$  vicini.

**Varianti.** Esistono alcune varianti di KNN:

- **Distanza pesata:** i punti a distanza più vicina hanno un peso maggiore nel voto.
- **K-NN per valori continui:** Assegna agli attributi ignoti della tupla da classificare la media dei valori degli attributi delle tuple più vicine (rispetto agli attributi noti).

## 7.7 Ensemble Learning

L'ensemble learning ha lo scopo di combinare più modelli, solitamente omogenei, per migliorare le prestazioni di classificazione rispetto a un singolo modello. L'idea è che combinando le predizioni di più modelli si possa ridurre l'errore complessivo, sfruttando la diversità tra i modelli. Poiché richiede molta potenza di calcolo, si combinano modelli semplici e veloci da addestrare.

### 7.7.1 Bagging

Il paradigma *Bagging* consiste nell'addestrare diversi classificatori in parallelo, combinando le predizioni finali.

**Algoritmo.** Siano  $M_1, M_2, \dots, M_k$  i  $k$  modelli da combinare:

1. Addestra ciascun modello  $M_i$  su un sottoinsieme casuale del training set o su un campione di dati estratto tramite bootstrapping<sup>2</sup>.
2. Combina opportunamente le predizioni dei modelli:
  - **Classificazione:** voto di maggioranza tra le classi predette.
  - **Regressione:** media delle predizioni.

**Random Forest.** Una Random Forest combina molti alberi decisionali addestrati su sottoinsiemi casuali del training set e su sottoinsiemi casuali di feature. Ogni albero viene addestrato su un campione bootstrap del dataset e, ad ogni split, si considera solo un sottoinsieme casuale delle feature per determinare la migliore divisione. La predizione finale viene effettuata tramite voto di maggioranza tra gli alberi.

---

<sup>2</sup>Il bootstrapping è una tecnica di campionamento con reinserimento che consente di creare più sottoinsiemi di dati a partire da un dataset originale. Ogni sottoinsieme può contenere duplicati e viene utilizzato per addestrare modelli diversi nell'ensemble learning.

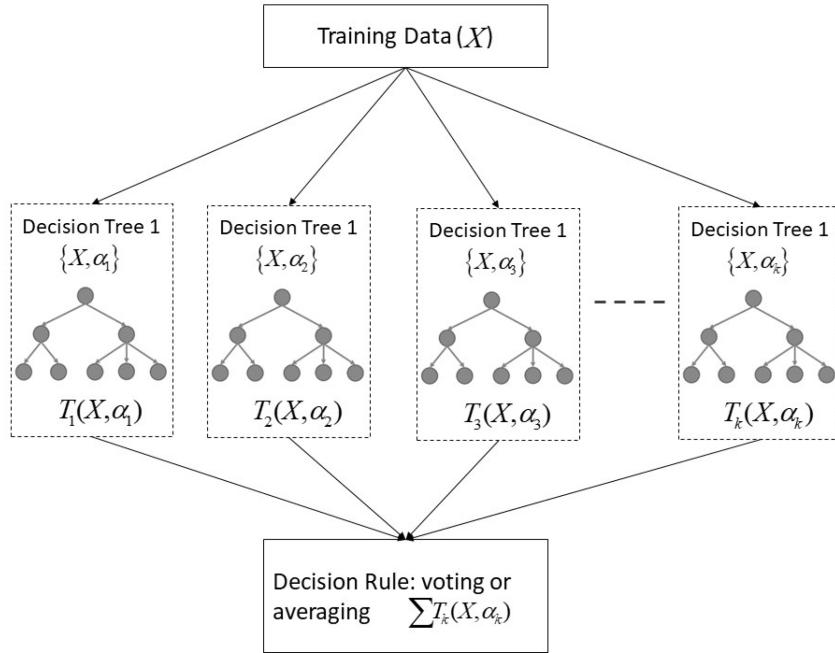


Figura 7.9: Schema di Random Forest: dai dati di training ( $X$ ) si costruiscono  $k$  alberi decisionali  $T_1(X, \alpha_1), T_2(X, \alpha_2), \dots, T_k(X, \alpha_k)$ , ciascuno con parametri  $\alpha_i$  diversi. La predizione finale è ottenuta combinando i risultati attraverso voting o averaging ( $\sum T_i(X, \alpha_i)$ ).

Il vantaggio è quello di usarlo su dataset di dati molto predittivi, in quanto riduce la correlazione tra gli alberi e migliora la generalizzazione.

### 7.7.2 Boosting

Il *Boosting* è un paradigma di ensemble learning che costruisce sequenzialmente una serie di modelli deboli, dove ogni modello successivo si concentra sugli errori commessi dai modelli precedenti.

**Algoritmo.** Siano  $M_1, M_2, \dots, M_k$  i  $k$  modelli da combinare:

1. Inizializza i pesi delle istanze del training set in modo uniforme.
2. Per ogni modello  $M_i$ :
  - (a) Addestra  $M_i$  sul training set ponderato.
  - (b) Calcola l'errore di  $M_i$  e aggiorna i pesi delle istanze: aumenta i pesi delle istanze mal classificate e diminuisce quelli delle istanze correttamente classificate.
3. Combina le predizioni dei modelli, pesando ciascuna predizione in base alla sua accuratezza (anche qui, nel caso di classificazione si usa il voto ponderato, mentre nella regressione si usa una media ponderata).

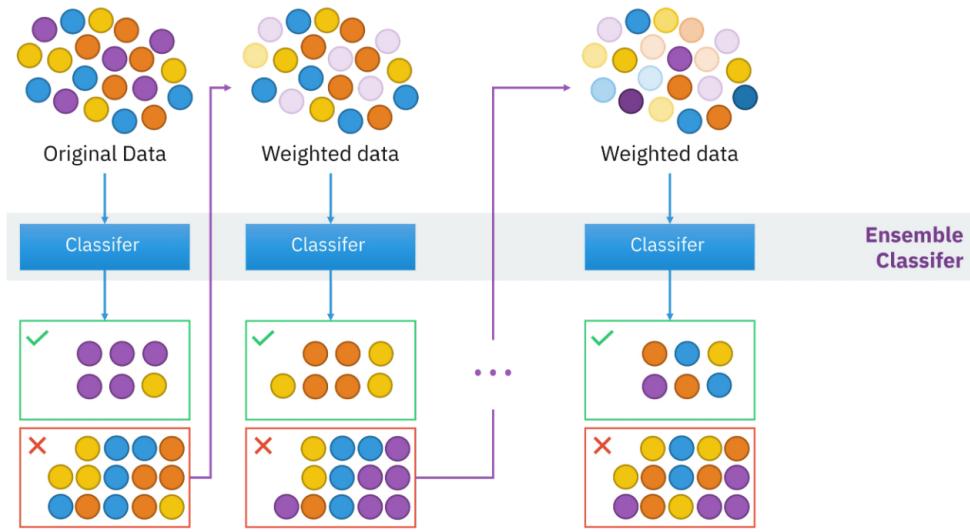


Figura 7.10: Schema di Boosting: ogni classificatore viene addestrato su dati pesati in base agli errori dei modelli precedenti. I dati mal classificati ricevono peso maggiore, così i modelli successivi si concentrano sulle istanze più difficili. Le predizioni finali sono combinate in un ensemble ponderato.

### 7.7.3 Adaboost

In questo algoritmo si usano solitamente **alberi decisionali** con due foglie chiamati *stumps*. Combinando opportunamente tali stumps, che sono semplici e veloci, si ottiene un classificatore accurato.

1. Inizializza i pesi delle istanze del training set in modo uniforme.
2. Per  $i = 1$  a  $k$ :
  - (a) Crea lo stump  $M_i$  che minimizza l'errore ponderato sul training set.
  - (b) Per lo stump  $M_i$ , calcola un peso  $P_i$  basato sulla sua accuratezza:
  - (c) Aumenta i pesi delle tuple classificate in modo errato e diminuisci quelli delle tuple classificate correttamente, dopodiché normalizza tra 0 e 1 i nuovi pesi.
3. Combina linearmente le predizioni dei modelli usando i pesi  $P_i$ .

**Gini Index Pesato.** Sia  $D$  un dataset con  $N$  tuple e  $k$  classi  $C_1, C_2, \dots, C_k$  e sia  $w(\mathbf{x})$  il peso associato alla tupla  $\mathbf{X}$ . Indico con  $T_{C_i}$  l'insieme delle tuple di  $D$  aventi classe  $C_i$ . La probabilità di osservare una tupla di classe  $C_i$  è:

$$p_i = \frac{\sum_{\mathbf{X} \in T_{C_i}} w(\mathbf{X})}{\sum_{\mathbf{Y} \in D} w(\mathbf{Y})}$$

Il **Gini index** dello splitting è dato da:

$$gini_{split}(S_X) = \sum_{i=1}^k \frac{|S_i|}{|S_X|} gini(S_i) \quad gini(S_i) = 1 - \sum_{i=1}^n p_i^2$$

**Peso dello stump.** Considerando TotalError la somma dei pesi delle tuple classificate erroneamente dallo stump  $M_i$ , il peso  $P_i$  dello stump si calcola come:

$$P_i = \frac{1}{2} \log\left(\frac{1 - \text{TotalError}}{\text{TotalError}}\right).$$

**Aggiornamento dei pesi.** Sia  $w^{(t)}(\mathbf{X})$  il peso della tupla  $\mathbf{X}$  all'iterazione  $t$ . Dopo aver calcolato lo stump  $M_i$  e il suo peso  $P_i$ , si aggiorna il peso della tupla come:

Sia  $w^{(i)}(\mathbf{X})$  il peso della tupla  $\mathbf{X}$  al passo  $i$ -esimo.

- Se  $\mathbf{X}$  non è classificata correttamente dallo stump  $M_i$ , il peso **aumenta**:

$$w^{i+1}(\mathbf{X}) = w^i(\mathbf{X}) \cdot e^{w(D_i)}$$

- Se  $\mathbf{X}$  è classificata **correttamente** dallo stump  $M_i$ , il peso **diminuisce**:

$$w^{i+1}(\mathbf{X}) = w^i(\mathbf{X}) \cdot e^{-w(D_i)}$$

Esempi che mostrano l'andamento di  $e^{w(D_i)}$  e  $e^{-w(D_i)}$  in funzione di  $w(D_i)$ :

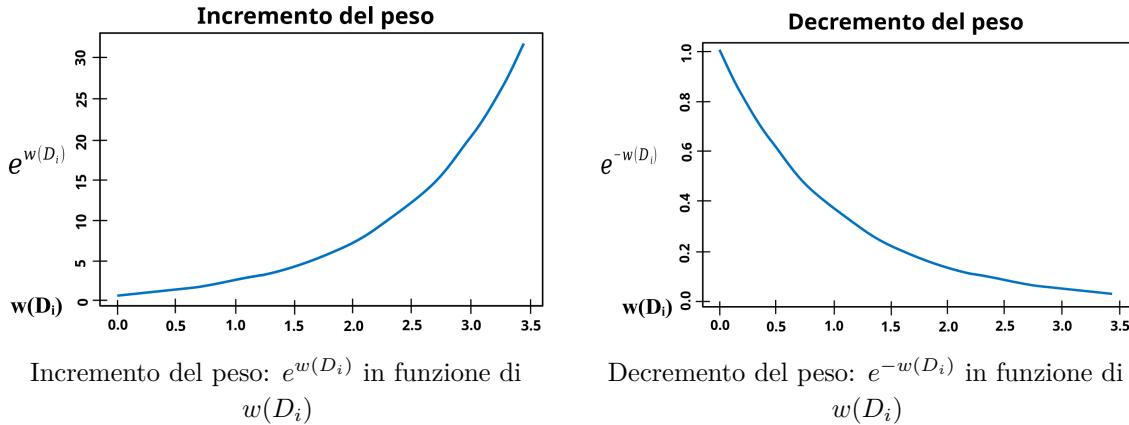


Figura 7.11: Andamento dell'incremento e del decremento dei pesi delle tuple per Adaboost.

## Bagging vs Boosting

Il bagging è utilizzato maggiormente per ridurre *overfitting*, mentre il boosting è più utilizzato quando si vogliono combinare classificatori molto semplici che presi singolarmente hanno bassa accuracy.

## 7.8 Validazione di un classificatore

### 7.8.1 Matrice di confusione

La matrice di confusione viene usata per rappresentare l'accuracy dove ogni riga contiene valori reali mentre ogni colonna i valori predetti. Un modello che funziona bene si vede dalle colonne, poiché se hanno valori diversi da 0 allora il modello predice bene la classe.

		Predetti			Somma
		Gatto	Cane	Coniglio	
Reali	Gatto	5	2	0	7
	Cane	3	3	2	8
	Coniglio	0	1	11	12
Somma		8	6	13	27

Figura 7.12: Esempio di matrice di confusione per un classificatore a tre classi (Gatto, Cane, Coniglio): le righe indicano le classi reali, le colonne le classi predette. I numeri nelle celle sono le frequenze assolute; le somme marginali mostrano il totale per riga/colonna. Un buon classificatore presenta valori elevati sulla diagonale principale (corretta assegnazione).

**Misure di accuratezza con due classi.** Si possono considerare due classi "Positiva"  $P$  e "Negativa"  $N$ . Da questo possono nascere:

- **True positive:** tuple di classe  $P$  che vengono classificate come  $P$ .
- **False positive:** tuple di classe  $P$  che vengono classificate come  $N$ .
- **True negative:** tuple di classe  $N$  che vengono classificate come  $N$ .
- **False negative:** tuple di classe  $N$  che vengono classificate come  $P$ .

**Misure di accuratezza (due classi).** Per il caso binario, indicando con  $|Tpos|$  il numero di true positive, con  $|Tneg|$  il numero di true negative, con  $|Fpos|$  il numero di false positive, con  $|Fneg|$  il numero di false negative, e con  $|Pos|, |Neg|$  i totali veri delle classi positive e

negative, si definiscono le misure più comuni:

$$\begin{aligned}
 \text{Recall (Rec) / Sensitivity / True Positive Rate (TPR)} : & \quad \text{Rec} = \frac{|\text{Tpos}|}{|\text{Pos}|}, \\
 \text{Specificity (Spec) / True Negative Rate (TNR)} : & \quad \text{Spec} = \frac{|\text{Tneg}|}{|\text{Neg}|}, \\
 \text{False Positive Rate (FPR)} : & \quad \text{FPR} = \frac{|\text{Fpos}|}{|\text{Neg}|}, \\
 \text{False Discovery Rate (FDR)} : & \quad \text{FDR} = \frac{|\text{Fpos}|}{|\text{Tpos}| + |\text{Fpos}|}, \\
 \text{Precision (Prec)} : & \quad \text{Prec} = \frac{|\text{Tpos}|}{|\text{Tpos}| + |\text{Fpos}|}, \\
 \text{Accuracy (Acc)} : & \quad \text{Acc} = \frac{|\text{Tpos}| + |\text{Tneg}|}{|\text{Pos}| + |\text{Neg}|}, \\
 \text{F1 (armonica di Precision e Recall)} : & \quad \text{F1} = 2 \cdot \frac{\text{Prec} \cdot \text{Rec}}{\text{Prec} + \text{Rec}}.
 \end{aligned}$$

Queste misure sono utili per valutare diversi aspetti del classificatore: la recall misura la capacità di trovare le istanze positive, la precision misura la correttezza delle predizioni positive, mentre l'accuracy fornisce una visione globale. In presenza di sbilanciamento di classe conviene preferire precision/recall o F1 all'accuracy.

### 7.8.2 Soglia discriminativa in un classificatore binario

In alcuni casi, la distinzione tra due classi (caso binario) si basa su un valore soglia  $\sigma$ . Ad esempio, predico se una mail è spam oppure no sulla base di tale soglia. Le performance del classificatore sono valutate al variare di  $\sigma$  tramite **curve ROC**. Esse rappresentano il *True Positive Rate (TPR)* in funzione del *False Positive Rate (FPR)*, al variare di  $\sigma$ .

Se la soglia è molto alta, tutte le tuple sono classificate come *negative*: nessuna tupla negativa è classificata come positiva ( $FPR = 0$ ) e nessuna tupla positiva è classificata come positiva ( $TPR = 0$ ).

### 7.8.3 Curva ROC

Se la soglia è molto bassa, tutte le tuple sono classificate come *positive*: tutte le tuple negative sono classificate come positive ( $FPR = 1$ ) e tutte le tuple positive sono classificate come positive ( $TPR = 1$ ). Per valori intermedi di soglia si ottengono valori di TPR e FPR compresi tra 0 e 1.

Situazione ideale: TPR aumenta fino a 1 mentre FPR si mantiene pari a 0 (*miglior classificatore*). Un *classificatore random* ha sempre valori di TPR e FPR uguali al variare di  $\sigma$ .

### 7.8.4 Curva di Precision-Recall

A differenza della curva ROC, la **curva di Precision-Recall** rappresenta la **Precision** in funzione della **Recall** al variare di  $\sigma$ .

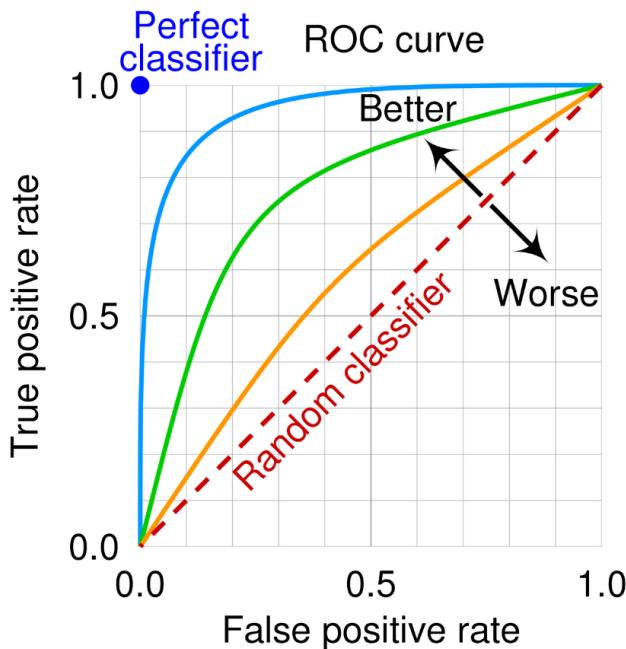


Figura 7.13: Esempio di curva ROC al variare della soglia  $\sigma$ .

Le curve **ROC** si utilizzano nel caso di dataset *bilanciati* (frequenza simile delle due classi), mentre la **curva di Precision-Recall** è preferibile nel caso di dataset *sbilanciati*.

Come indicatore dell'accuratezza del classificatore si usa l'area al di sotto delle curve ROC e Precision-Recall, detta **Area Under the Curve (AUC)**.  $AUC \in [0, 1]$ ;  $AUC = 1$  indica un classificatore perfetto.

### 7.8.5 Validazione di un classificatore

Per validare un classificatore si considerano solitamente diversi *partizionamenti* del dataset in *training* e *test set* e si addestra il classificatore sul training set così ottenuto. Generalmente si utilizzano due metodi:

**Holdout:** Si fissa una percentuale  $X$ , dopo si partiziona il dataset in due set indipendenti, training e test sets, in base a  $X$ , si addestra il modello sul training set e si applica il classificatore al test set per misurare l'accuratezza. Una variante prevede di ripetere l'holdout  $k$  volte e di calcolare la media delle accuratezze ottenute.

**K-Fold Cross Validation:** Si partiziona il dataset in  $k$  sottosets (fold) di dimensioni approssimativamente uguali. Si eseguono  $k$  iterazioni, in ciascuna si usa un fold come test set e gli altri  $k - 1$  fold come training set. Si calcola l'accuratezza per ogni iterazione e si fa la media delle  $k$  accuratezze ottenute per avere una stima complessiva delle prestazioni del classificatore.



## Capitolo 8

# Cenni di Regressione

La regressione è una tecnica di apprendimento supervisionato utilizzata per prevedere valori continui basandosi su un insieme di dati di input. A differenza della classificazione, che assegna etichette discrete, la regressione mira a modellare la relazione tra una variabile dipendente (target) e una o più variabili indipendenti (predittori).

Da notare che la regressione può essere analizzata su diversi aspetti:

- Un aspetto statistico: la regressione viene utilizzata per stimare le relazioni tra variabili e per fare inferenze sui dati.
- Un aspetto di machine learning: la regressione viene utilizzata per costruire modelli predittivi che possono generalizzare su nuovi dati.

Questo capitolo darà una panoramica delle principali tecniche di regressione utilizzate nel data mining e nel machine learning.

### 8.1 Regressione lineare semplice

La regressione lineare semplice è il caso più elementare di regressione, in cui si cerca di modellare la relazione tra due variabili: una variabile indipendente  $x$  e una variabile dipendente  $y$ .

#### 8.1.1 Formulazione del modello

La relazione viene rappresentata da una retta di regressione, espressa dall'equazione:

$$y = \beta_0 + \beta_1 x + \epsilon$$

dove:

- $\beta_0$  è l'intercetta (valore di  $y$  quando  $x = 0$ ). Questo parametro viene anche chiamato **bias** e rappresenta il punto in cui la retta interseca l'asse delle ordinate,
- $\beta_1$  è il coefficiente di regressione (pendenza della retta) che indica quanto varia  $y$  al variare di  $x$ .

- $\epsilon$  è l'errore casuale. Questo errore viene introdotto per tenere conto delle variazioni non spiegate dal modello lineare.

### 8.1.2 Stima dei parametri

Nel contesto del machine learning, l'obiettivo è stimare i parametri  $\beta_0$  e  $\beta_1$ . In particolare si può utilizzare quella che viene chiamata la **metodologia dei minimi quadrati** (Ordinary Least Squares, OLS) che minimizza la somma dei quadrati degli errori tra i valori osservati e quelli predetti dal modello. La funzione di costo da minimizzare è:

$$J(\beta_0, \beta_1) = \sum_{i=1}^n (y_i - (\beta_0 + \beta_1 x_i))^2$$

dove  $n$  è il numero di osservazioni nel dataset.

### 8.1.3 Interpretazione geoemetrica

Geometricamente, la regressione lineare semplice cerca di trovare la retta che meglio si adatta ai punti dati in uno spazio bidimensionale. La pendenza  $\beta_1$  indica l'inclinazione della retta, mentre l'intercetta  $\beta_0$  indica il punto in cui la retta interseca l'asse delle ordinate.

## 8.2 Regressione lineare multipla

Un altro caso di regressione lineare è quella multipla, in cui si considerano più variabili indipendenti  $x_1, x_2, \dots, x_p$  per prevedere la variabile dipendente  $y$ .

### 8.2.1 Formulazione del modello

La relazione viene rappresentata dall'equazione:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p + \epsilon$$

o, in forma matriciale:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \epsilon$$

dove:

- $\mathbf{y}$  è il vettore delle osservazioni della variabile dipendente,
- $\mathbf{X}$  è la matrice delle variabili indipendenti (inclusa una colonna di 1 per l'intercetta),
- $\boldsymbol{\beta}$  è il vettore dei coefficienti di regressione,
- $\epsilon$  è il vettore degli errori casuali.

### 8.2.2 Stima dei parametri

Anche in questo caso, si può utilizzare la metodologia dei minimi quadrati per stimare i parametri  $\beta$ . La soluzione analitica è data dalla formula:

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

dove  $\hat{\beta}$  rappresenta i coefficienti stimati.

### 8.2.3 Interpretazione geometrica

Geometricamente, la regressione lineare multipla cerca di trovare un iperpiano in uno spazio multidimensionale che meglio si adatta ai punti dati. Ogni coefficiente  $\beta_j$  rappresenta l'effetto marginale della variabile  $x_j$  sulla variabile dipendente  $y$ , mantenendo costanti tutte le altre variabili indipendenti.

## 8.3 Regressione non lineare

In molti casi, la relazione tra le variabili indipendenti e la variabile dipendente non è lineare. In questi casi, si possono utilizzare modelli di regressione non lineare, che possono assumere varie forme, come polinomiali, esponenziali o logaritmiche. Questi modelli possono essere stimati utilizzando tecniche di ottimizzazione numerica, poiché spesso non esiste una soluzione analitica semplice come nel caso della regressione lineare.

Un esempio di regressione polinomiale è dato dall'equazione:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \epsilon$$

dove i termini  $x^2$  e  $x^3$  permettono di modellare relazioni più complesse tra  $x$  e  $y$ . Tuttavia questa formulazione si può convertire in una funzione lineare, considerando  $x_2 = x^2$  e  $x_3 = x^3$  come nuove variabili indipendenti:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon$$

## 8.4 Regressione logistica

La regressione logistica è una tecnica utilizzata per problemi di classificazione binaria, ma può essere vista come un'estensione della regressione lineare. In questo caso, l'obiettivo è prevedere la probabilità che un'osservazione appartenga a una delle due classi. La relazione viene modellata utilizzando la funzione logistica (sigmoide):

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \dots + \beta_p x_p)}}$$

dove  $P(y = 1|x)$  rappresenta la probabilità che la variabile dipendente  $y$  sia uguale a 1 dato il vettore delle variabili indipendenti  $x$ .

### 8.4.1 Regressione logistica binaria semplice

Nel caso più semplice di regressione logistica binaria con una sola variabile indipendente  $x$ , la probabilità viene espressa come:

$$P(y = 1|x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}}$$

In questo caso, il modello stima la probabilità che l'evento  $y = 1$  si verifichi in funzione della variabile  $x$ .

**Stima dei parametri.** Per stimare i parametri ottimali  $\beta_0$  e  $\beta_1$  si può andare a minimizzare la somma delle log-loss delle singole tuple. Data una tupla  $T$  avente variabile predittiva  $x$  e variabile target  $y \in \{0, 1\}$ , la log-loss è definita come:

$$\text{log-loss}(x) = \begin{cases} -\log p(x) & \text{se } y = 1 \\ -\log(1 - p(x)) & \text{se } y = 0 \end{cases}$$

dove  $p(x) = P(y = 1|x)$  è la probabilità stimata dal modello. La funzione di costo complessiva da minimizzare è quindi:

$$J(\beta_0, \beta_1) = -\sum_{i=1}^n [y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i))]$$

## Riferimenti

I riferimenti di questo capitolo sono:

- Capitolo 12 di [8].
- Fondamenti con esempi e pratica guidata nel libro [7].
- Approfondimenti teorici nel libro [6].

# Capitolo 9

## Subgraph Matching

L'operazione di *subgraph matching* consiste nel trovare tutte le occorrenze di un grafo più piccolo (detto *query graph*) all'interno di un grafo più grande (detto *data graph*). Questa operazione è fondamentale in molte applicazioni, come il rilevamento di pattern in reti sociali, l'analisi di reti biologiche e la ricerca di strutture specifiche in database grafici.

### 9.1 Isomorfismo di Grafi

L'**isomorfismo** tra due grafi  $G_1$  e  $G_2$  è una corrispondenza biunivoca tra i loro insiemi di nodi che preserva le relazioni di adiacenza. In altre parole, due grafi sono isomorfi se esiste una mappatura tra i nodi di  $G_1$  e  $G_2$  tale che due nodi sono connessi da un arco in  $G_1$  se e solo se i loro corrispondenti nodi sono connessi da un arco in  $G_2$ .

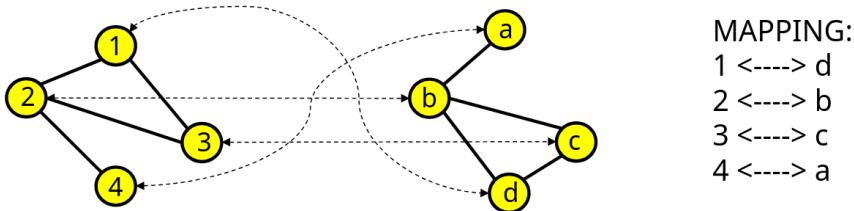


Figura 9.1: Esempio di subgraph mapping: il grafo di query (a sinistra) viene trovato all'interno del grafo di dati (a destra).

#### 9.1.1 Automorfismo

Un caso particolare di *isomorfismo* è l'**automorfismo**: si verifica quando un grafo è isomorfo a se stesso. In altre parole, un automorfismo è una mappatura dei nodi di un grafo su se stessi che preserva le relazioni di adiacenza. Gli automorfismi sono importanti nello studio delle simmetrie nei grafi e possono essere utilizzati per semplificare la rappresentazione di un grafo.

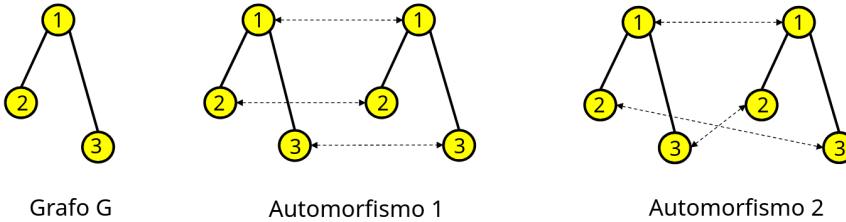


Figura 9.2: Esempi di *automorfismi* del grafo  $G$ : a sinistra il grafo originale; al centro un automorfismo che scambia i vertici 2 e 3 mantenendo 1 fisso; a destra un secondo automorfismo che permuta i vertici come indicato dalle frecce tratteggiate. In entrambi i casi la struttura di adiacenza è preservata.

## 9.2 Operazione di subgraph matching

L'operazione di *subgraph matching* consiste, definita in modo più formale, nel verificare se il *grafo query* è contenuto nel *grafo dati* attraverso una mappatura che preserva le relazioni di adiacenza.

La differenza rispetto al concetto di isomorfismo risiede nel fatto che, nel subgraph matching, il grafo query può essere più piccolo del grafo dati e non è necessario che tutti i nodi del grafo dati siano coinvolti nella mappatura. Quindi, il subgraph matching cerca una corrispondenza parziale, **iniettiva**, tra i due grafi, mentre l'isomorfismo richiede una corrispondenza completa, **biunivoca**, tra tutti i nodi e gli archi dei due grafi.

Definiamo  $G_1 = (V_1, E_1)$  come il grafo query e  $G_2 = (V_2, E_2)$  come il grafo dati. L'operazione di subgraph matching cerca una funzione iniettiva  $f : V_1 \rightarrow V_2$  chiamata **mapping** tale che per ogni arco  $(u, v) \in E_1$ , l'arco  $(f(u), f(v))$  appartiene a  $E_2$ .

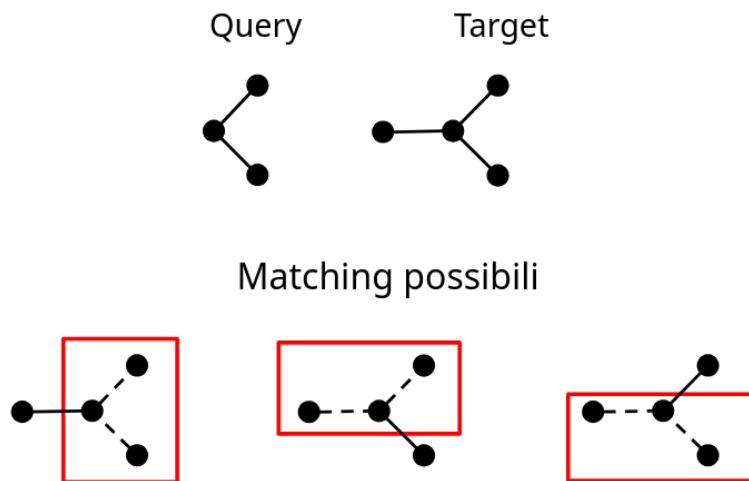


Figura 9.3: Esempio di *subgraph matching*: il grafo *Query* (in alto a sinistra) viene ricercato all'interno del grafo *Target* (in alto a destra). In basso sono mostrati i possibili match del sottografo query all'interno del grafo target, evidenziati dai riquadri rossi.

Come si vede dalla figura 9.3, il processo di subgraph matching ritorna più occorrenze del grafo query all'interno del grafo dati, evidenziando le diverse mappature possibili che soddisfano le condizioni di adiacenza.

## 9.3 Complessità computazionale

L'operazione di graph matching è nota per essere computazionalmente complessa. Infatti, il problema del graph matching è NP-Hard, ovvero non esiste un algoritmo noto che possa risolverlo in tempo polinomiale per tutti i casi. Questo rende il graph matching un algoritmo particolarmente esoso.

Per quanto riguarda il subgraph matching, la complessità dipende dalla dimensione del grafo query e del grafo dati, nonché dalla struttura dei grafici stessi. In generale è un problema NP-Completo, il che significa che non esiste un algoritmo noto che possa risolverlo in tempo polinomiale per tutti i casi, tuttavia esistono algoritmi euristici e approssimativi che possono essere utilizzati per affrontare il problema in modo più efficiente in casi specifici o con vincoli particolari.

## 9.4 Algoritmi di subgraph matching

### 9.4.1 Soluzione Bruteforce

La soluzione più semplice per il problema del subgraph matching è l'approccio **bruteforce**, che consiste nel generare tutte le possibili mappature dei nodi del grafo query sui nodi del grafo dati e verificare se ciascuna mappatura preserva le relazioni di adiacenza.

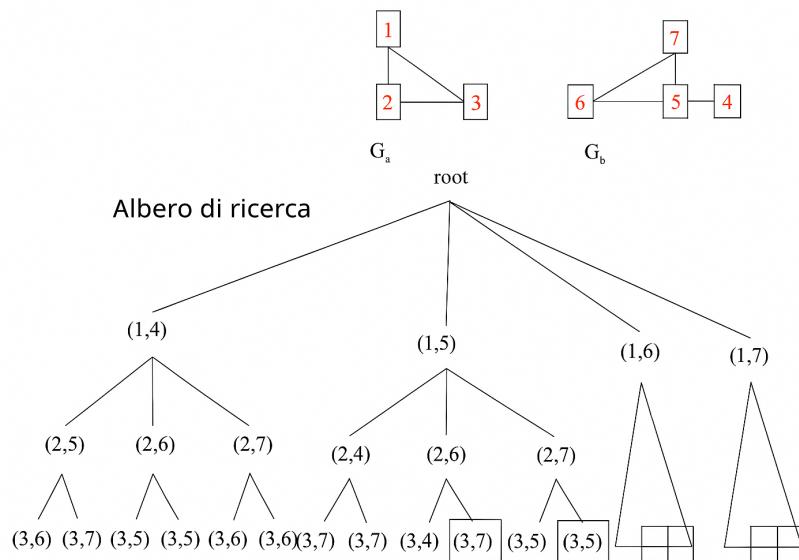


Figura 9.4: Ricerca *brute force* per *subgraph matching*: in alto i grafi  $G_a$  (query) e  $G_b$  (target); sotto l'albero di ricerca che esplora tutte le corrispondenze possibili tra vertici (es. (1,4), (1,5), (1,6), (1,7)). Le foglie evidenziate indicano i mapping completi che preservano le adiacenze (match isomorfi).

Questo è chiaramente una soluzione NP-Hard, inefficiente. Si possono adottare strategie migliori per velocizzare la computazione:

- **Look-ahead:** prima di verificare una mappatura completa, si può controllare se i nodi parzialmente mappati soddisfano le condizioni di adiacenza. Se non lo fanno, si può scartare immediatamente quella mappatura.
- **Backtracking:** si può utilizzare una strategia di backtracking per esplorare lo spazio delle mappature in modo più efficiente. Se si scopre che una mappatura parziale non può essere estesa a una mappatura completa valida, si torna indietro e si prova una diversa mappatura.

**Esempio di backtracking.** Partendo dalla radice dell'albero di ricerca (figura 9.5), assegnamo progressivamente i vertici del grafo *query* ai vertici del *target*. Dopo le prime scelte ( $1 \rightarrow a$ ) e ( $2 \rightarrow b$ ), proviamo ( $3 \rightarrow c$ ). A questo punto un controllo locale sui vincoli struttura-preservanti rivela un'incoerenza: il grado di 3 nel query è 3, mentre quello di *c* nel target è 2; quindi nessun proseguimento potrà produrre un mapping valido. L'algoritmo esegue allora *pruning* del ramo e *backtracking* allo stato precedente per provare alternative (ad es. ( $3 \rightarrow d$ )). Questo comportamento evita esplorazioni inutili e riduce drasticamente lo spazio di ricerca.

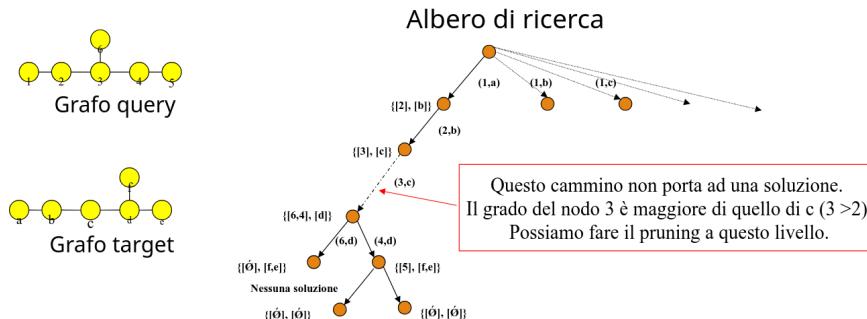


Figura 9.5: Esempio di *backtracking* con *pruning* nel *subgraph matching*. A sinistra il grafo *query* e il grafo *target*; a destra l'albero di ricerca. Il ramo che mappa ( $3 \rightarrow c$ ) viene potato perché viola un vincolo di grado ( $\deg(3) = 3 > \deg(c) = 2$ ), quindi non può condurre a una soluzione.

#### 9.4.2 Algoritmo di Ullmann

L'algoritmo di Ullmann utilizza il grado del nodo come criterio di selezione per ridurre lo spazio di ricerca. In particolare, prima di tentare una mappatura, l'algoritmo verifica se il grado del nodo nel grafo *query* è minore o uguale al grado del nodo corrispondente nel grafo dati. Se questa condizione non è soddisfatta, la mappatura viene scartata immediatamente (*look-ahead*).

Dopo aver costruito una mappatura iniziale, l'algoritmo applica una procedura di **refinement** per eliminare ulteriori mappature non valide. Questa procedura verifica che

per ogni arco nel grafo query, l'arco corrispondente esista nel grafo dati. Se un arco manca, la mappatura viene scartata (*backtracking*).

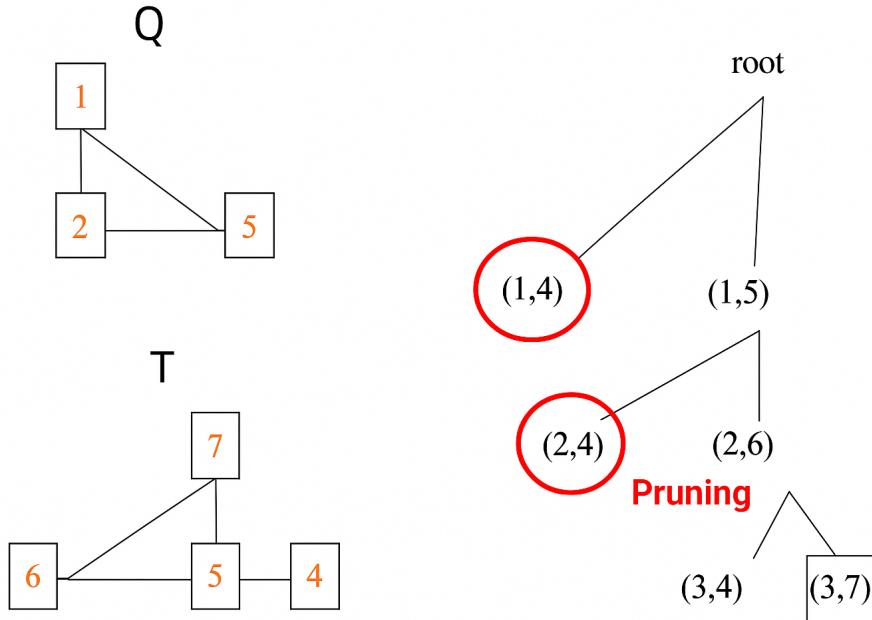


Figura 9.6: Algoritmo di Ullmann: a sinistra il grafo *query*  $Q$  e il grafo *target*  $T$ ; a destra l'albero di ricerca con le corrispondenze provate. I nodi cerchiati in rosso indicano scelte scartate tramite *pruning*; il ramo con  $(3, 7)$  completa un mapping valido.

**Esempio della figura 9.6.** Per prima cosa si calcola l'ordine di ogni vertice, nel grafo  $Q$  abbiamo  $\deg(1) = \deg(2) = \deg(5)$ , nel grafo  $T$  abbiamo  $\deg(6) = \deg(7) = \deg(5) = 2, \deg(4) = 1$ .

L'obiettivo è quello di trovare un *isomorfismo* tra i due grafi, iniziando dalla radice dell'albero di ricerca. Si prova l'assegnazione  $(1, 4)$  ma viene scartata perché il grado di 1 è maggiore del grado di 4. Si prova quindi  $(1, 5)$  e si procede con  $(2, 4)$  (non funziona per lo stesso motivo  $(1, 4)$ ), quindi si prova  $(2, 6)$  e così via. Alla fine si trova il mapping valido  $(1, 5), (2, 6), (5, 7)$ .

#### 9.4.3 Algoritmo VF

L'algoritmo VF (Vento-Foggia) è un altro approccio per il problema del subgraph matching che utilizza una strategia di backtracking con pruning basata su vincoli locali. L'algoritmo costruisce una mappatura incrementale dei nodi del grafo query sui nodi del grafo dati, verificando a ogni passo se la mappatura parziale soddisfa i vincoli di adiacenza.

L'algoritmo costruisce 3 insiemi per ognuno dei grafi  $Q, T$ :

- $M_Q, M_T$ : nodi già mappati
- $T_Q, T_T$ : nodi adiacenti a quelli mappati

- $U_Q, U_T$ : nodi non ancora considerati

Dove il pedice  $Q$  o  $T$  indica se l'insieme appartiene al grafo query o target.

**N.B.** È importante notare che l'algoritmo VF è ricorsivo ma lavora a **stati**: ad ogni passo ricorsivo considera lo stato attuale in cui, definito dagli insiemi di nodi mappati, adiacenti e non ancora considerati.

**Algoritmo.** Ad ogni passo dell'algoritmo, per lo stato corrente  $s$ , si eseguono i seguenti passi:

1. Si seleziona un nodo  $q \in T_Q(s)$  e si prova a mappare  $\forall t \in T_T(s)$ .
2. Si verifica la regola di **fattibilità** per un certa coppia  $(q, t) \in T_Q(s) \times T_T(s)$ . Se la coppia è fattibile, si crea un nuovo stato  $s'$  aggiornando gli insiemi  $M_Q, M_T, T_Q, T_T, U_Q, U_T$  di entrambi i grafi. Da notare che si preferisce sempre scegliere una coppia di nodi che il grado più alto possibile, in modo da massimizzare le possibilità di pruning e ridurre la computazione nel caso medio.
3. Si ripete il processo fino a quando tutti i nodi del grafo query sono stati mappati (trovando così un match) o fino a quando non ci sono più nodi da mappare (in tal caso si esegue il backtracking).

**Regola di fattibilità per grafi indiretti.** Una coppia di nodi  $(q, t) \in T_Q(s) \times T_T(s)$  è considerata fattibile se soddisfa le seguenti condizioni:

- Per ogni nodo query  $q' \in M_Q(s)$  connesso a  $q$ , esiste un nodo target  $t' \in M_T(s)$  connesso a  $t$ .
- Il numero di nodi query in  $T_Q(s)$  deve essere minore o uguale<sup>1</sup> al numero di nodi target in  $T_T(s)$  (regola look-ahead a un livello).
- Il numero di nodi query in  $U_Q(s)$  deve essere minore o uguale<sup>2</sup> al numero di nodi target in  $U_T(s)$  (regola look-ahead a due livelli).

**Regola di fattibilità per grafi diretti.** Nel caso di grafi diretti, la regola di fattibilità deve considerare anche la direzione degli archi. Quindi, ogni grafo non ha più 3 insiemi di nodi ma 6:

- $M_Q^{in}, M_T^{in}$ : nodi già mappati considerando gli archi entranti
- $M_Q^{out}, M_T^{out}$ : nodi già mappati considerando gli archi uscenti
- $T_Q^{in}, T_T^{in}$ : nodi adiacenti a quelli mappati considerando gli archi entranti
- $T_Q^{out}, T_T^{out}$ : nodi adiacenti a quelli mappati considerando gli archi uscenti
- $U_Q, U_T$ : nodi non ancora considerati

La regola di fattibilità in questo caso, deve essere modificata per tenere conto delle direzioni degli archi:

---

<sup>1</sup>Minore o uguale nel caso di subgraph matching, nel caso di graph matching invece deve essere uguale.

<sup>2</sup>Anche qui, minore uguale solo nel caso di subgraph matching. Nel caso di Graph deve essere uguale.

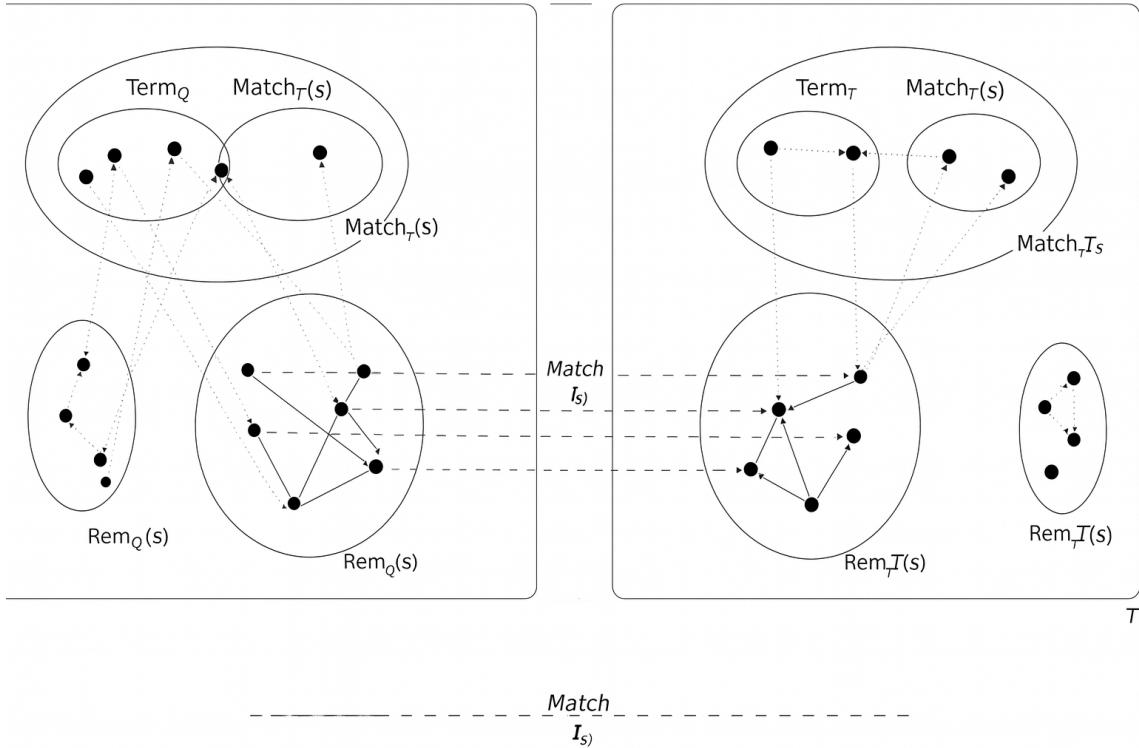


Figura 9.7: Rappresentazione grafica dei sei insiemi utilizzati dall'algoritmo VF per verificare la regola di fattibilità durante la costruzione incrementale della mappatura tra nodi del grafo query  $Q$  e nodi del grafo target  $T$ . La figura mostra, per uno stato  $s$  dell'algoritmo, la suddivisione di entrambi i grafi nei tre insiemi fondamentali: i nodi già mappati ( $\text{Match}_Q(s)$  e  $\text{Match}_T(s)$ ), i nodi adiacenti a quelli mappati ( $\text{Term}_Q(s)$  e  $\text{Term}_T(s)$ ) e i nodi non ancora considerati ( $\text{Rem}_Q(s)$  e  $\text{Rem}_T(s)$ ). Le frecce tratteggiate indicano le possibili connessioni tra nodi dei diversi insiemi, mentre le frecce orizzontali centrali rappresentano la mappatura parziale  $M(s)$  costruita dallo stato corrente. Questa struttura consente di verificare la regola di fattibilità: ogni nuova coppia  $(q, t)$  può essere aggiunta alla mappatura se preserva le adiacenze verso i nodi già mappati, se esistono sufficienti nodi adiacenti disponibili (look-ahead a un livello) e se esistono sufficienti nodi non ancora considerati per supportare eventuali mappature future (look-ahead a due livelli).

- Per ogni nodo query  $q' \in M_Q^{in}(s)$ , predecessore di  $q$ , esiste un nodo target  $t' \in M_T^{in}(s)$  predecessore di  $t$ .
- Per ogni nodo query  $q' \in M_Q^{out}(s)$ , successore di  $q$ , esiste un nodo target  $t' \in M_T^{out}(s)$  successore di  $t$ .
- Il numero di nodi query in  $T_Q^{in}(s)$  deve essere minore o uguale al numero di nodi target in  $T_T^{in}(s)$ .
- Il numero di nodi query in  $T_Q^{out}(s)$  deve essere minore o uguale al numero di nodi target in  $T_T^{out}(s)$ .
- Il numero di nodi query in  $U_Q(s)$  deve essere minore o uguale al numero di nodi target in  $U_T(s)$ .

**Complessità e considerazioni** La complessità, nell’ipotesi che i due grafi abbiano  $N$  nodi, si può calcolare come:

1. Sapendo che ogni nodo ha in media  $O(N)$  nodi adiacenti, quindi per ogni nodo query si devono considerare  $O(N)$  possibili mappature nel grafo target.
2. Per ogni mappatura, si devono verificare le condizioni di fattibilità, che richiedono di controllare gli archi adiacenti. Questo richiede  $O(N)$  operazioni.

Quindi nel caso migliore, la complessità dell’algoritmo dipende dalla visita di un solo nodo candidato per ogni nodo query, portando a una complessità di  $O(N^2)$ . Tuttavia, nel caso peggiore, l’algoritmo potrebbe dover esplorare tutte le possibili mappature, portando a una complessità esponenziale di  $O(N!)$ .

La complessità spaziale, al più  $k$  stati (dove  $k$  è il numero di nodi del grafo query) porta a una complessità spaziale di  $O(k \cdot N)$ , considerando che per ogni stato si devono memorizzare gli insiemi di nodi mappati, adiacenti e non ancora considerati per entrambi i grafi.

#### 9.4.4 Algoritmo VF2

L’algoritmo VF2 è un miglioramento dell’algoritmo VF, progettato per essere più efficiente nel risolvere il problema del subgraph matching. VF2 ottimizza lo spazio utilizzato, fino ad ottenere una complessità spaziale pari a  $O(N)$ , utilizzando strutture dati globali e condivise tra i vari stati. Si introducono sei strutture dati:

- $\text{core}_1$  e  $\text{core}_2$ : array che memorizzano la mappatura corrente dei nodi del grafo query e del grafo target, rispettivamente.
- $\text{in}_1$  e  $\text{in}_2$ : array booleani che indicano se un nodo è adiacente a un nodo già mappato (insiemi  $\text{Term}_Q$  e  $\text{Term}_T$ ).
- $\text{out}_1$  e  $\text{out}_2$ : array booleani che indicano se un nodo non è ancora considerato (insiemi  $\text{Rem}_Q$  e  $\text{Rem}_T$ ).

Grazie a queste strutture, si può *tracciare* contemporaneamente l’appartenenza dei nodi agli insiemi necessari per la verifica della regola di fattibilità, senza dover mantenere

insiemi separati per ogni stato dell'algoritmo. Questo riduce significativamente l'overhead di memoria, permettendo all'algoritmo di gestire grafi più grandi in modo più efficiente.

#### 9.4.5 Algoritmo RI

L'algoritmo RI è un approccio alternativo per il problema del subgraph matching. Gli algoritmi VF si basano su una strategia di backtracking con pruning, mentre l'algoritmo RI si concentra invece **sull'ordine** in cui i nodi della query vengono processati nell'albero di ricerca. Un ordinamento efficace, infatti, *velocizza* molto il matching anche in presenza di regole di pruning meno restrittive rispetto a VF.

In particolare, l'ordinamento in RI è calcolato indipendentemente dal grafo target (static ordering) e si può riassumere nei seguenti passi:

1. Si ordinano i nodi del grafo query in modo da **massimizzare** la probabilità che un cammino parziale nell'albero di ricerca venga tagliato il prima possibile.
2. Seguendo l'ordinamento calcolato al passo 1, si mappano i nodi della query a nodi del target verificando per ogni coppia candidata  $(q, t)$  che:
  - (a)  $q$  e  $t$  non siano già stati mappati ad altri nodi.
  - (b) Il grado di  $q$  sia minore o uguale al grado di  $t$ .
3. Si ripete il passo 2 finché l'intero spazio di ricerca non viene esplorato.

**Ordinamento dei nodi.** Il punto principale di questo algoritmo è capire come ordinare i nodi della query per massimizzare l'efficacia del pruning. L'obiettivo dell'ordinamento è quello di costruire una sequenza ordinata di nodi  $[q_1, q_2, \dots, q_n]$  tale che, durante la ricerca, i nodi connessi ai nodi già mappati vengano processati il prima possibile. Questo aumenta la probabilità che le condizioni di fattibilità vengano violate precocemente, permettendo di tagliare rami dell'albero di ricerca in modo più efficiente. Per farlo, si può definire una **regola generale**: al passo  $i$ , si sceglie il nodo  $q_i$  con grado più alto e con un elevato numero di connessioni con i nodi già presenti nell'ordinamento  $\bigcup^{i-1}$ .

Sia  $\bigcup^{i-1} = \{q_1, q_2, \dots, q_{i-1}\}$  la sequenza parziale già ordinata al passo  $i - 1$ . Allora, il punteggio di un nodo candidato  $v$  da inserire nell'ordinamento è definito sulla base di 3 insiemi:

1.  $V_{adj}(i)$ : insieme dei nodi adiacenti a  $v$  che sono già presenti in  $\bigcup^{i-1}$ .
2.  $V_{conn}(i)$ : insieme dei nodi in  $\bigcup^{i-1}$  adiacenti ad almeno un nodo che non appartiene a  $\bigcup^{i-1}$  ed è connesso a  $v$ .
3.  $V_{rem}(i)$ : insieme dei nodi non ancora considerati (cioè non in  $\bigcup^{i-1}$ ) e non connessi a nessun nodo in  $\bigcup^{i-1}$ , ma connessi a  $v$ .

Da questo si crea un punteggio per il successivo nodo da inserire:

1. Massimo valore di  $|V_{adj}(i)|$ .
2. In caso di parità, massimo valore di  $|V_{conn}(i)|$ .
3. In caso di ulteriore parità, massimo valore di  $|V_{rem}(i)|$ .

Se ancora dovesse esserci parità, si sceglie il nodo con il grado più alto.

**Esempio.**

Consideriamo il grafo di query  $Q$  nella figura 9.8

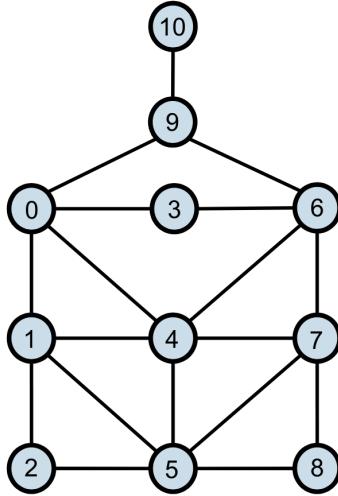


Figura 9.8: Grafo non diretto di esempio (11 nodi, 0-10), usato per illustrare l'ordinamento dei nodi della query nell'algoritmo RI.

**Passo  $i = 1$ .** Per prima cosa, dobbiamo ordinare i nodi. Lo stato iniziale dell'ordinamento è  $U^0 = \emptyset$ , per ogni nodo  $v$  calcoliamo i 3 insiemi:

$$|V_{\text{adiac}}(1, v)| = 0, \quad |V_{\text{conn}}(1, v)| = 0, \quad |V_{\text{rem}}(1, v)| = \deg(v)$$

Quindi vince il nodo con grado massimo, ovvero il nodo 4. Ora lo stato dell'ordinamento è  $U^1 = (4)$ .

**Passo  $i = 2$ .** Possiamo calcolare una tabella per tutti i candidati al passo 2, in quanto ora  $U^1 = (4)$ .

Nodo $v$	$V_{\text{adiac}}(2, v)$	$V_{\text{conn}}(2, v)$	$V_{\text{rem}}(2, v)$
0	{4}	{4}	{9}
1	{4}	{4}	{2}
2	$\emptyset$	{4}	$\emptyset$
3	{4}	{4}	$\emptyset$
5	{4}	{4}	{2,8}
6	{4}	{4}	{9}
7	{4}	{4}	{8}
8	$\emptyset$	{4}	$\emptyset$
9	$\emptyset$	{4}	{10}
10	$\emptyset$	$\emptyset$	{9}

In questo caso, si nota che i nodi candidati hanno  $|V_{\text{adiac}}| = 1$ . e  $|V_{\text{conn}}| = 1$ . Quindi si passa a guardare  $|V_{\text{rem}}|$ , che è massimo per il nodo 5 (2 nodi). Quindi il secondo nodo dell'ordinamento è  $u_2 = 5$ , e lo stato diventa  $U^2 = (4, 5)$ .

**Tabella finale** Dopo aver ripetuto questo processo per tutti i nodi, otteniamo l'ordinamento finale:

$$U = (4, 5, 1, 3, 7, 0, 6, 9, 2, 8, 10)$$

#### 9.4.6 RI-DS

Una versione ottimizzata dell'algoritmo RI, chiamata RI-DS (RI with Domain Size ordering), introduce un'ulteriore strategia di ordinamento basata sulle dimensioni del **dominio** di ciascun nodo della query. In particolare si cerca di applicare la regola del grado di Ullman in fase di matching una sola volta per una stessa coppia  $(q, t)$  e ridurre a monte le possibili coppie candidate. Questo perché se un nodo  $q$  della query ha un grado maggiore di un nodo  $t$  del target, allora non può esistere una mappatura valida che includa la coppia  $(q, t)$ .

Si può definire, per un certo nodo  $q$  della query, il suo **dominio**  $D(q)$  come l'insieme dei nodi del grafo target che possono essere mappati a  $q$  rispettando la condizione del grado:

$$D(q) = \{t \in V_T : \deg(q) \leq \deg(t)\}$$

dove  $\deg(x)$  indica il grado del nodo  $x$ . Da questo, l'algoritmo esegue i seguenti passi:

1. Ordina i nodi del grafo query.
2. Calcola i domini per ogni nodo query.
3. SEguendo l'ordinamento calcolato al passo 1, si mappano i nodi della query a nodi del target verificando per ogni coppia candidata  $(q, t)$  che:
  - (a)  $q$  e  $t$  non siano già stati mappati ad altri nodi.
  - (b)  $t \in D(q)$ , ovvero il nodo target  $t$  appartiene al dominio del nodo query  $q$ .
4. Si ripete il passo 3 finché l'intero spazio di ricerca non viene esplorato.

## 9.5 Subgraph Matching in Database di Grafi

Il problema del subgraph matching è di fondamentale importanza nei database di grafi, dove si desidera trovare sottografi specifici all'interno di uno o più grafi target. Il problema diventa particolarmente rilevante in applicazioni come la bioinformatica, l'analisi delle reti sociali e la ricerca di pattern in grandi dataset.

La vera sfida in questi scenari è l'efficienza: i database di grafi possono contenere milioni o addirittura miliardi di nodi e archi, rendendo il subgraph matching un compito computazionalmente intensivo.

### 9.5.1 Indicizzazione

Una soluzione *naif* consisterebbe nell'eseguire il subgraph matching su ogni grafo del database; tuttavia, ciò sarebbe proibitivo in termini di tempo.

Per ottenere tempi ragionevoli, si indicizzano i grafi del database così da **filtrare** rapidamente i candidati rilevanti per una data query. L'idea è il classico paradigma *filter-and-verify*: prima si usa l'indice per selezionare soltanto i grafi che condividono con la query determinate *caratteristiche discriminanti* (ad es. frequenti sottografi, cammini/percorsi *q*-gram, firme di vicinato, conteggi di etichette e gradi), poi si applica il subgraph matching *solo* su questo sottoinsieme ristretto. In questo modo si riduce drasticamente il numero di confronti, concentrando l'attenzione sui grafi con una reale possibilità di contenere il sottografo cercato, e migliorando sensibilmente i tempi di risposta senza sacrificare la correttezza del risultato.

Esistono due tipi di indicizzazione:

**Indicizzazione basata su feature** - il grafo viene rappresentato tramite un insieme di **feature** (sottografi frequenti, cammini, alberi, ecc.) che catturano le sue proprietà strutturali. Queste feature vengono poi memorizzate in un indice che consente di recuperare rapidamente i grafi che contengono determinate feature.

**Indicizzazione non basata su feature** - i grafi vengono mappati in uno spazio metrico e memorizzato in un **albero** (come un B-tree) dove fare ricerca basata su distanze. In questo modo, si possono recuperare rapidamente i grafi che sono "vicini" alla query in termini di struttura.

## 9.6 Features dei grafi

Le **feature** sono delle proprietà strutturali dei grafi che possono essere utilizzate per l'indicizzazione e il filtraggio nei database di grafi. Si possono estrarre dal grafo diverse feature:

- **Piccoli sottografi**: si possono identificare e memorizzare piccoli sottografi frequenti all'interno del grafo. Questi sottografi possono essere utilizzati come feature per rappresentare la struttura del grafo.
- **Cammini e percorsi**: si possono estrarre cammini di lunghezza fissa (chiamati *q*-gram) o percorsi specifici all'interno del grafo. Questi cammini possono essere utilizzati per confrontare la struttura dei grafi.
- **Alberi**: si possono identificare alberi radicati o alberi di profondità limitata all'interno del grafo. Questi alberi possono essere utilizzati come feature per rappresentare la struttura gerarchica del grafo.

**Esempio di filtraggio tramite profili di feature.** L'idea è rappresentare sia la query  $Q$  sia ciascun grafo  $G$  con un vettore/multinsieme di *feature* e delle loro occorrenze (cammini, piccoli pattern etichettati, ecc.). Nel riquadro di sinistra,  $F_Q$  riporta per ogni feature

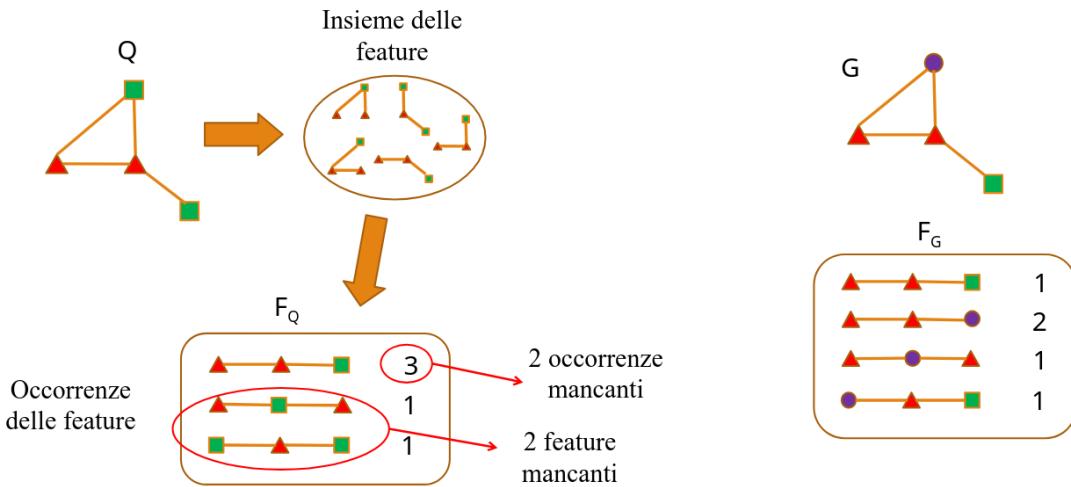


Figura 9.9: Indicizzazione *feature-based* per il filtraggio dei candidati. A sinistra: dalla query  $Q$  si estraggono piccole caratteristiche (feature) e si costruisce il profilo di frequenza  $F_Q$ . A destra: per ogni grafo del database (es.  $G$ ) è precomputato il profilo  $F_G$ .

il numero richiesto dalla query: una di esse richiede 3 occorrenze, altre due compaiono almeno una volta. Nel riquadro di destra,  $F_G$  mostra le occorrenze del medesimo insieme di feature nel grafo candidato  $G$ .

Regola di filtro: se esiste una feature  $f$  tale che  $F_Q(f) > F_G(f)$ , allora  $G$  non può contenere  $Q$  e viene scartato. Nell'immagine, per una feature  $Q$  richiede 3 occorrenze mentre  $G$  ne ha 1 ( $\rightarrow$  2 occorrenze mancanti); inoltre due feature presenti in  $Q$  non compaiono in  $G$  (2 feature mancanti). Il grafo  $G$  viene dunque escluso già in fase di filtro, riducendo il numero di candidati da passare al subgraph matching.

### 9.6.1 Schema di subgraph matching in database di grafi

Nel caso di database di grafi, il subgraph matching segue lo schema *filter-and-verify* (filtra e verifica):

1. **Preprocessing:** per ogni grafo del database, si estraggono tutte le features rilevanti che contiene.
2. **Filtering:** dalla query si estraggono tutte le feature contenute e si confrontano con quelle dei grafi del database, scartando quelli che non le contengono tutte.
3. **Matching:** per ogni grafo rimanente, si esegue l'algoritmo di subgraph matching (ad es. VF2, RI-DS) per verificare se la query è effettivamente un sottografo del grafo target.

Grazie a questo schema, si riduce drasticamente il numero di grafi su cui eseguire il subgraph matching, migliorando l'efficienza complessiva del processo.

### 9.6.2 Indicizzazione inversa

Un'altra tecnica di indicizzazione nei database di grafi è l'uso di un indice inverso<sup>3</sup> basato sulle feature. Ad ogni feature (intesa come chiave) si associa la lista dei grafi che la contengono, con il relativo numero di occorrenze della feature.

L'insieme dei candidati per il matching si può ottenere per intersezione delle liste dei grafi associate a ciascuna feature della query. In questo modo, si ottiene rapidamente l'insieme ristretto di grafi che contengono tutte le feature richieste dalla query, riducendo il numero di confronti necessari per il subgraph matching.

### 9.6.3 Algoritmo SING

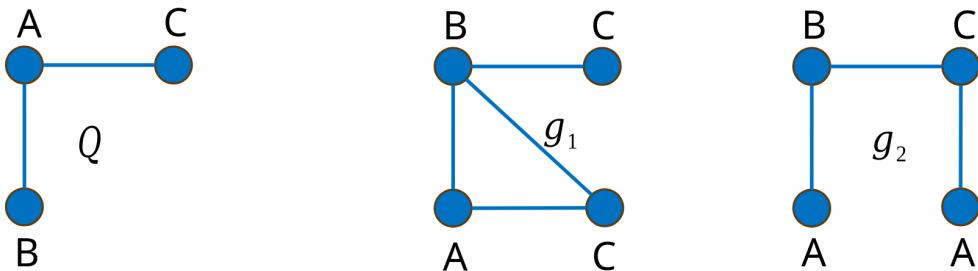


Figura 9.10: Query  $Q$  (angolo con nodo  $A$  adiacente a  $B$  e  $C$ ) e due grafi candidati. Nel grafo  $g_1$  il pattern è presente come *subgrafo non indotto*, ma non come *indotto* a causa dell'arco diagonale aggiuntivo; nel grafo  $g_2$  il pattern è presente anche come *subgrafo indotto*.

L'algoritmo SING (Subgraph search In Non-homogeneous Graphs) è un algoritmo di subgraph matching in un database di grafi basato su indexing tramite cammini di nodi. Per migliorare l'efficacia dell'indicizzazione si associa ad ogni feature la frequenza di occorrenza in ogni grafo del database ed il nodo da cui parte.

L'algoritmo utilizza due tipi di **indicizzazione** (come si vede in figura 9.11):

**Indice inverso globale** - ad ogni feature si associa la lista dei grafi del database che la contengo e il relativo conteggio di occorrenze. Questo aiuta a filtrare rapidamente i grafi che non contengono tutte le feature della query.

**Indice inverso locale per ciascun grafo  $g$**  - per ogni feature presente in  $g$  è associato un vettore binario dove l' $i$ -esimo bit indica se la feature parte dal nodo  $i$  di  $g$ . Questo aiuta a velocizzare il processo di matching all'interno di ciascun grafo candidato.

**Processamento della query.** L'algoritmo procede a eseguire una query di subgraph matching seguendo questi passi:

<sup>3</sup>per indice inverso si intende una struttura dati che mappa feature a grafi contenenti tali feature

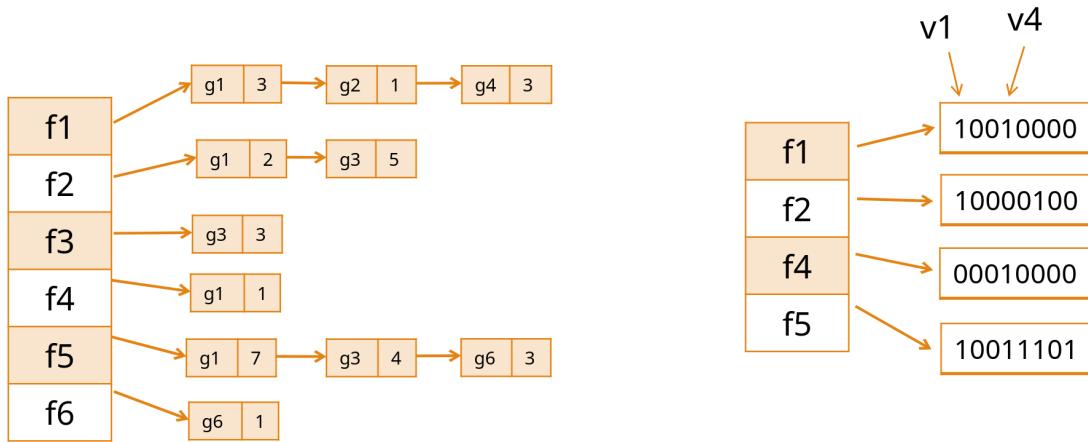


Figura 9.11: Indicizzazione inversa globale e locale per l'algoritmo SING. A sinistra: indice inverso globale che associa ad ogni feature a lista dei grafi che la contengono e il conteggio delle occorrenze. A destra: indice inverso locale per il grafo  $g_1$ , dove per ogni feature è indicato un vettore binario che mostra i nodi di partenza della feature in  $g_1$ .

1. Si esegue il **primo filtraggio**, ovvero per ogni feature  $F$  della query si recuperano i grafi che la contengono, per un maggiore numero di volte del grafo query, dall'indice inverso globale. Si ottiene così un insieme di grafi candidati che contengono tutte le feature della query e poi insieme si calcola l'intersezione  $R$ .
2. Dopo si esegue un **secondo filtraggio**, per ogni grafo  $G \in R$  si usa l'indice locale per calcolare gli insiemi di nodi compatibili con i nodi della query. Da quello, si scartano i grafi che non hanno almeno un nodo compatibile per ogni nodo della query. Questo, dato un vertice  $v$  della query, calcola l'insieme dei vertici di un grafo  $G$  del database che sono compatibili con  $v$ . Si dice compatibile un nodo  $u$  di  $G$  se esiste almeno una feature che parte da  $v$  nella query e da  $u$  in  $G$ . Dato  $S$  insieme di queste features che partono da  $v$ , è sufficiente calcolare nell'indice locale del grafo  $G$  un AND logico tra i vettori binari associati alle feature di  $S$  e se restituisce un vettore non nullo, allora esiste almeno un nodo compatibile con  $v$  in  $G$ .
3. Infine, si esegue il **matching** vero e proprio sui grafi rimanenti, usando un algoritmo di subgraph matching (ad es. VF2, RI-DS) per verificare se la query è effettivamente un sottografo del grafo target.

## Riferimenti

I riferimenti per questo capitolo sono:

- Articolo originale dell'algoritmo Vento-Foggia[5]
- Articolo originale dell'algoritmo VF2[4]
- Articolo dell'algoritmo RI e RI-DS [2]



## Capitolo 10

# Subgraph Matching di Grafi Frequenti

Come nel caso del capitolo 5, l'estrazione di sotto-grafi frequenti consiste nell'**identificare** tutti i sotto-grafi che appaiono frequentemente in un database di grafi.

Anche in questo caso si conta il numero di grafi del dataset che contengono il sottografo, chiamato **supporto** e se il supporto del sotto-grafo è maggiore o uguale ad una soglia minima di supporto, allora il sotto-grafo è considerato **frequente**. Per fare una relazione con il mining di insiemi frequenti, possiamo considerare una mappatura del tipo:

- Transazioni → Grafi
- Itemsets → Sotto-grafi
- Item → Nodi
- Relazioni tra gli item → Archi tra i nodi

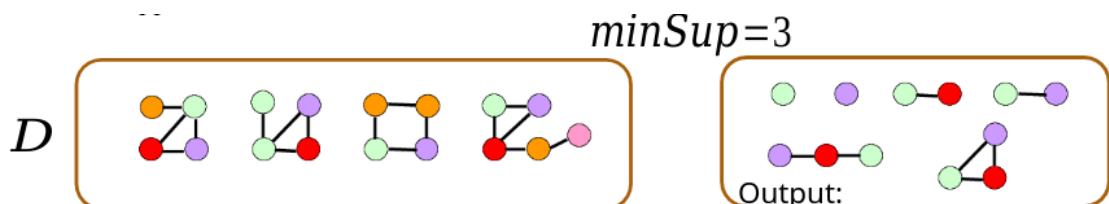


Figura 10.1: Esempio di estrazione di sotto-grafi frequenti da un database di grafi.

Spesso non si calcola il supporto di un sotto-grafo, ma bensì alla sua frequenza: il conteggio relativo delle occorrenze del sotto-grafo all'interno di un singolo grafo. In questo caso, il sotto-grafo è considerato frequente se la somma delle frequenze nei grafi del database è maggiore o uguale alla soglia minima di supporto, che indichiamo con  $\sigma$  ed è una percentuale.

Non si fa riferimento al supporto perché così abbiamo un valore *normalizzato*<sup>1</sup>.

<sup>1</sup>La frequenza di un elemento è definita come il numero di elementi (il supporto in questo caso) diviso il numero totale (il numero di grafi nell'intero database in questo caso).

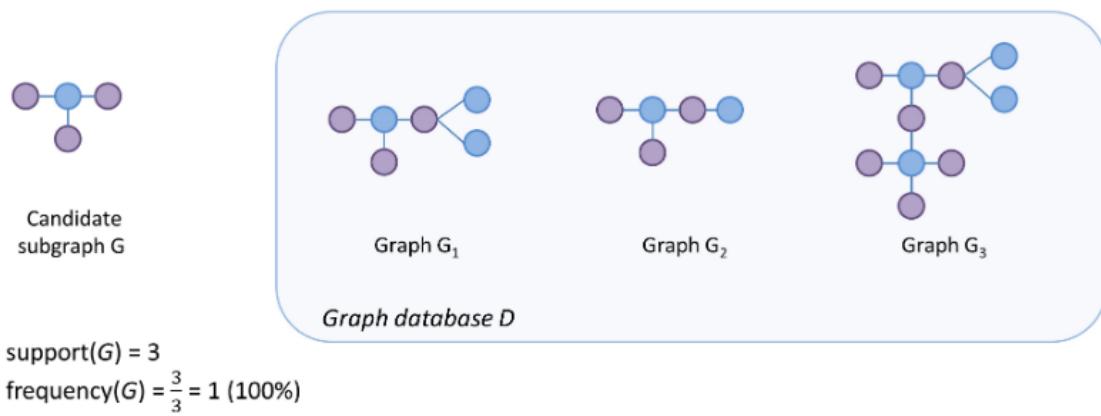


Figura 10.2: Esempio di estrazione di sotto-grafi frequenti basata sulla frequenza nei grafi del database.

Il problema dell'estrazione di sotto-grafi frequenti pone sfide molto più complesse rispetto all'estrazione di insiemi frequenti, in quanto il problema dell' **isomorfismo tra sotto-grafi** è NP-completo. Questo significa che non esistono algoritmi efficienti noti per risolvere questo problema in tutti i casi.

Per risolvere questo problema ci si pone delle domande:

- Quale strategia adottare?
- Come generare in maniera efficiente i sottografi candidati?
- Come evitare o gestire le ridondanze nella generazione dei candidati?

Esistono due tipologie di algoritmi principali per l'estrazione di sotto-grafi frequenti:

- **Algoritmi Apriori:** algoritmi che sono basati sulla regola *apriori* e la generazione di candidati tramite join di sottografi frequenti.
- **Algoritmi pattern-growth:** algoritmi basati sulla generazione dei candidati tramite *aggiunta di nodi/archi* ai sottografi frequenti esistenti.

## 10.1 Algoritmo FSG

Uno dei primi algoritmi proposti per l'estrazione di sotto-grafi frequenti è l'algoritmo **FSG** (Frequent Subgraph Discovery). FSG è un algoritmo basato sulla strategia *apriori* che utilizza un approccio di generazione di candidati tramite join di sottografi frequenti.

### 10.1.1 Regola Apriori per sotto-grafi

Nel contesto dell'estrazione di sotto-grafi frequenti, la regola *apriori* afferma che se un sotto-grafo è frequente, allora tutti i suoi sotto-grafi devono essere anch'essi frequenti. Questo implica che se un sotto-grafo non è frequente, allora nessuno dei suoi super-grafi può essere frequente (esattamente come nel caso degli itemsets<sup>5</sup>).

### 10.1.2 Join tra sotto-grafi

Per capire come funzionano gli algoritmi basati sulla regola apriori, è importante comprendere come avviene la generazione dei candidati tramite join tra sotto-grafi. Due sotto-grafi  $g_1$  e  $g_2$  possono essere uniti se condividono un sottografo con archi in comune, chiamato **grafo core**.

Il problema di questo tipo di Join è che, a differenza degli itemset, si possono generare più candidati distinti a partire dalla stessa coppia di sotto-grafi. Esistono 3 scenari possibili.

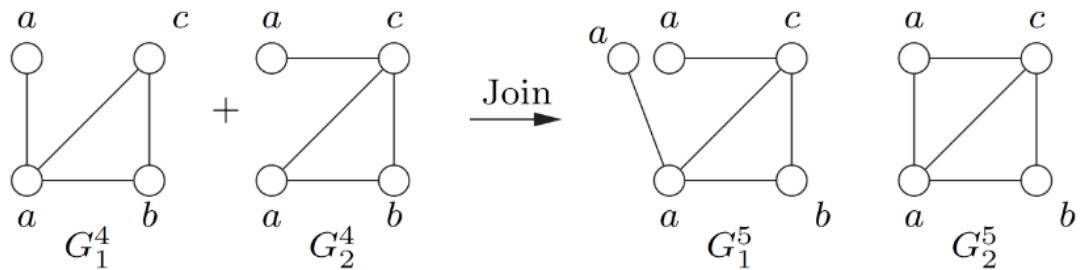


Figura 10.3: Join tra due sottografi che differiscono per un nodo.

**Scenario 1: i due sottografi differiscono per un nodo.** Come si vede dall'esempio in figura 10.3, i due sottografi condividono un sottografo comune, composto dalle etichette  $a, b, c$ , ma il secondo nodo  $a$  differisce nel modo in cui è collegato (per  $G_1^4$  è collegato a  $c$ , mentre per  $G_2^4$  è collegato ad  $a$ ). In questo caso esistono due possibili risultati:

1. Tenere separate le due occorrenze del nodo  $a$ , ottenendo il grafo  $G_1^5$ .
2. Unire i due nodi  $a$  in un unico nodo, ottenendo il grafo  $G_2^5$ .

In questo caso i candidati vengono scelti in base al *contesto*:

**Esempio 1 :** Se i nodi  $a, b, c$  rappresentano rispettivamente *atomi di carbonio, idrogeno e ossigeno*, allora le due occorrenze del nodo  $a$  rappresentano due atomi di carbonio distinti e quindi si sceglie il grafo  $G_1^5$ .

**Esempio 2 :** Se i nodi  $a, b, c$  rappresentano rispettivamente *persone, pubblicazioni e conferenze*, allora le due occorrenze del nodo  $a$  rappresentano la stessa persona che ha pubblicato due articoli distinti e quindi si sceglie il grafo  $G_2^5$ .

**Scenario 2: il grafo core ha più automorfismi.** Nel caso di figura 10.4 il *core* comune dei due sottografi  $G_1^5$  e  $G_2^5$  è un ciclo di lunghezza 4 (etichettato tutto con  $a$ ). Poiché tutti i vertici del core hanno la stessa etichetta, il core è altamente simmetrico: il ciclo  $C_4$  ammette 8 automorfismi (le 4 rotazioni e le 4 riflessioni del core).

Quando si esegue la *join*, bisogna quindi decidere *come* mappare il core di  $G_2^5$  sul core di  $G_1^5$ . Automorfismi diversi producono posizionamenti diversi del nodo extra (in  $G_1^5$  il

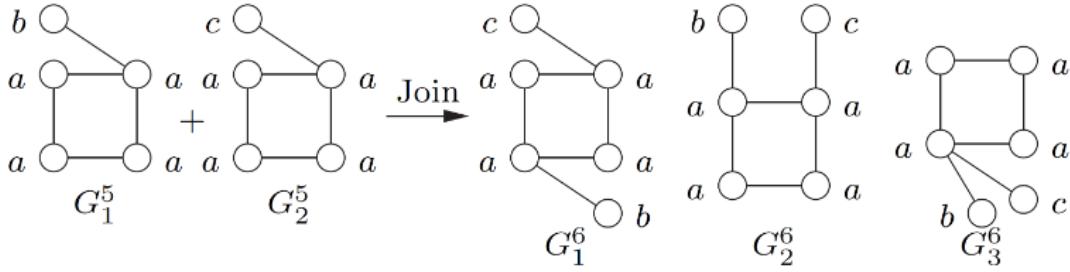


Figura 10.4: Join tra due sottografi con grafo core avente più automorfismi.

nodo  $b$ , in  $G_2^5$  il nodo  $c$ ) rispetto al core. Dopo l'eliminazione dei duplicati isomorfi, restano tre candidati non isomorfi (raffigurati a destra nella figura):

1.  **$G_1^6$  (spigoli opposti).** Il nodo  $b$  si collega a un estremo di un lato del quadrato e il nodo  $c$  all'estremo del lato opposto.
2.  **$G_2^6$  (stesso lato).** I nodi  $b$  e  $c$  si collegano sullo *stesso* lato del quadrato, ciascuno collegato ai due estremi di quel lato.
3.  **$G_3^6$  (condivisione di un vertice).** I nodi  $b$  e  $c$  condividono un estremo: entrambi sono collegati allo stesso vertice del core (e all'altro estremo del rispettivo lato adiacente).

Come nello Scenario 1, la scelta tra i candidati dipende dal *contesto*:

**Esempio 1 :** Se  $a$  rappresenta atomi di *carbonio* in un ciclo (ad es. un anello  $C_4$ ) e  $b, c$  sono due *sostituenti*, i tre candidati corrispondono a tre posizionamenti non equivalenti: *opposti* ( $G_1^6$ ), *adiacenti sullo stesso lato* ( $G_2^6$ ) e *geminali* sullo stesso carbonio ( $G_3^6$ ). Il dominio (vincoli chimici o esempi osservati) decide quale configurazione mantenere.

**Esempio 2 :** Se  $a$  sono *autori* connessi da collaborazioni,  $b$  e  $c$  sono due *pubblicazioni* che coinvolgono coppie di autori adiacenti, allora:

- due pubblicazioni scritte da *coppie disgiunte* di autori  $\Rightarrow G_1^6$ ;
- due pubblicazioni scritte dalla *stessa coppia*  $\Rightarrow G_2^6$ ;
- due pubblicazioni che *condividono un autore* ma non l'altro  $\Rightarrow G_3^6$ .

**Scenario 3: i sottografi candidati hanno più grafi core in comune.** Nell'esempio di figura 10.5 i due pattern di partenza  $G_1^4$  e  $G_2^4$  condividono *due* massimi sottografi comuni (raffigurati in basso a sinistra), che chiamiamo  $H_1^3$  e  $H_2^3$ . La join va quindi eseguita due volte, una per ciascun core, perché allineamenti diversi del nucleo comune portano a sovrapposizioni differenti dei nodi esterni.

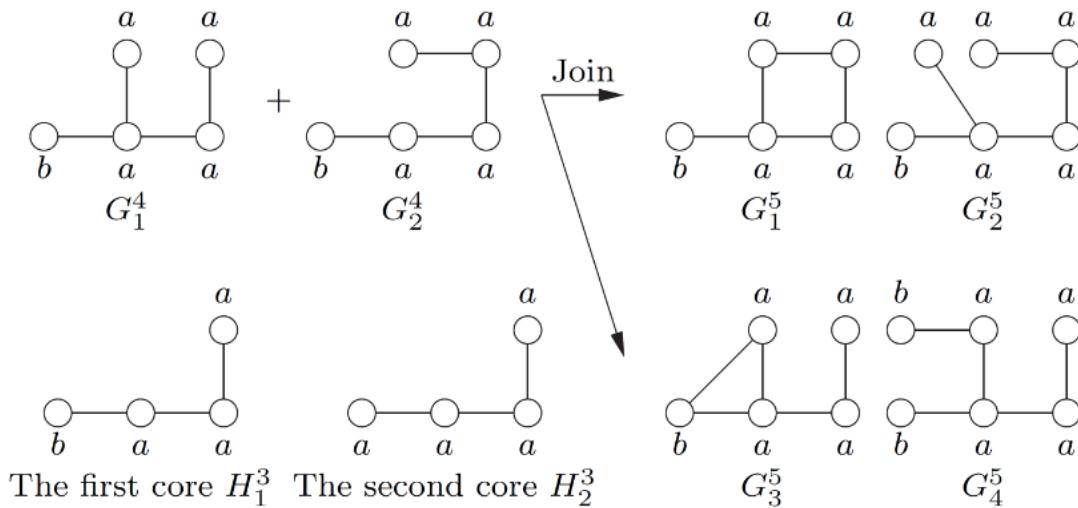


Figura 10.5: Join tra due sottografi con più grafi core in comune.

**Join rispetto al primo core  $H_1^3$ .** Prendendo  $H_1^3$  come nucleo condiviso, i nodi rimanenti nei due pattern sono due vertici etichettati *a* collegati a posizioni diverse del core. Come nello Scenario 1, abbiamo due possibilità:

1. mantenere *separate* le due occorrenze del nodo *a*  $\Rightarrow$  candidato  $G_1^5$  (in alto a destra, primo grafo);
2. *fondere* le due occorrenze del nodo *a* in un unico vertice  $\Rightarrow$  candidato  $G_2^5$  (in alto a destra, secondo grafo).

**Join rispetto al secondo core  $H_2^3$ .** Ripetendo lo stesso procedimento usando il core  $H_2^3$ , otteniamo altri due candidati non isomorfi:

1. mantenere *separate* le due occorrenze del nodo *a*  $\Rightarrow$  candidato  $G_3^5$  (in basso a destra, primo grafo);
2. *fondere* le due occorrenze del nodo *a* in un unico vertice  $\Rightarrow$  candidato  $G_4^5$  (in basso a destra, secondo grafo).

Come per gli scenari precedenti, la scelta del candidato dipende dal *contesto*:

**Esempio 1 :** Sia *a* un *carbonio* e *b* un *eteroatomo*<sup>2</sup> (ad es. ossigeno) su una catena. Se i due frammenti osservati provengono da *due sostituenti distinti* allora si mantengono i nodi separati (si seleziona  $G_1^5$  o  $G_3^5$  a seconda del core coerente con i dati); se invece i dati indicano che si tratta dello *stesso* sostituente visto in due prospettive compatibili, si effettua la fusione ( $G_2^5$  o  $G_4^5$ ).

**Esempio 2 :** Sia *a* un *autore* e *b* un *progetto*. I due core alternativi rappresentano due gruppi di co-autori sovrapposti in modo diverso. Se le osservazioni mostrano che le due occorrenze di *a* sono *persone diverse*, si sceglie un candidato con nodi separati

<sup>2</sup>Un eteroatomo è un atomo diverso dal carbonio in un composto organico.

$(G_1^5 \text{ o } G_3^5)$ ; se sono la *stessa persona* che compare in entrambi i pattern, si sceglie il candidato con merge  $(G_2^5 \text{ o } G_4^5)$ .

**Caso generale.** Il caso generale della Join tra due sottografi  $g_1$  e  $g_2$  prevede i seguenti passi:

1. Sia  $F^k$  l'insieme dei sottografi frequenti con  $k$  archi e sia  $C^{k+1}$  l'insieme dei candidati di sotto-grafi di dimensione  $k + 1$  (inizialmente vuoto).
2. Per ogni coppia di sottografi frequenti con  $k$  archi  $(G_1^k, G_2^k)$  in  $F^k$ :
  - (a) Calcola l'insieme dei core condivisi da  $G_1^k$  e  $G_2^k$ .
  - (b) Per ogni core condiviso e per ogni automorfismo del core, effettua la join e aggiungi i sottografi candidati trovati a  $C^{k+1}$ .
3. Restituisci l'insieme  $C^{k+1}$  dei sottografi candidati di dimensione  $k + 1$ .

### 10.1.3 Procedura dell'algoritmo

L'algoritmo FSG segue i seguenti passi:

1. Per prima cosa calcola nodi e archi frequenti (sotto-grafi di dimensione 1 e 2) e li aggiunge all'insieme finale  $O$  dei risultati.
2. Per un certo  $k \geq 3$  ripete:
  - (a) Genera i candidati  $C_k$  di sotto-grafi di dimensione  $k$  tramite join dei sotto-grafi frequenti di dimensione  $k - 1$ .
  - (b) Per ogni grafo  $G$  nel database verifica la regola Apriori, scartando tutti i candidati che contengono almeno un sottografo con archi che non è frequente.
  - (c) Calcola il supporto di ogni grafo candidato e l'insieme dei sottografi frequenti con  $k$  archi. Poi aggiunge questi sottografi all'insieme finale  $O$  dei risultati.
3. Restituisci  $O$

### 10.1.4 Generazione dei candidati

Il passo più complesso dell'algoritmo FSG è la generazione dei candidati. Per generare i candidati di dimensione  $k$ , l'algoritmo esegue una join tra tutti i sottografi frequenti di dimensione  $k - 1$ . Una volta eseguita la join si potrebbe presentare il problema dei **candidati ridondanti**, ovvero sottografi isomorfi generati più volte. Per evitare questo problema, FSG utilizza una tecnica di **canonical labeling** (forma canonica), che assegna ad ogni sottografo un'etichetta unica basata sulla sua struttura e sulle etichette dei nodi e degli archi. In questo modo, se due sottografi isomorfi hanno la stessa etichetta canonica, uno dei due può essere scartato come ridondante.

### 10.1.5 Stringa di adiacenza

La stringa di adiacenza di un grafo è una stringa ottenuta concatenando le righe della matrice di adiacenza (nel caso di grafi indiretti si considera la metà superiore della matrice) come si vede nell'esempio in figura 10.6

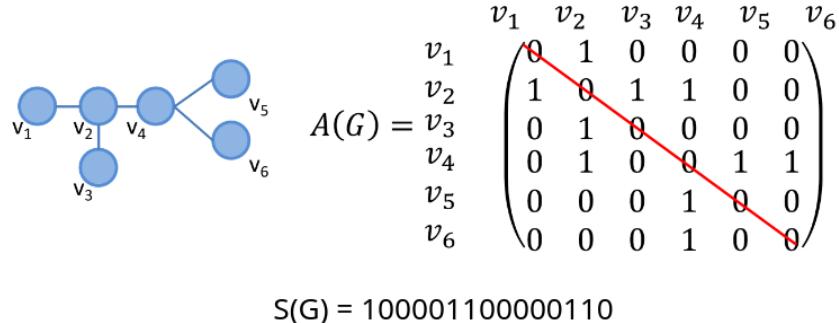


Figura 10.6: Esempio di stringa di adiacenza di un grafo.

### 10.1.6 Forma canonica

Per un grafo  $G$  con  $n$  nodi, esistono  $n!$  possibili modi di etichettare i nodi (ovvero  $n!$  stringhe di adiacenza) considerando tutte le possibili permutazioni dei nodi di  $G$  (diverse permutazioni possono produrre la stessa stringa di adiacenza). La **forma canonica** di un grafo  $G$  è definita come la stringa di adiacenza minima (o massima), in ordine lessicografico, tra tutte le possibili stringhe di adiacenza generate dalle permutazioni dei nodi di  $G$ . Il problema di questo calcolo è che *computazionalmente oneroso* e FSG risolve questo problema utilizzando **gradi dei nodi** e **etichette** per ridurre il numero di permutazioni da considerare.

**Algoritmo.** Il calcolo della forma canonica segue questi passi:

1. Partiziona i nodi in gruppi sulla base del grado.
2. Partiziona ogni gruppo ottenuto al passo precedente in sottogruppi sulla base delle etichette dei nodi.
3. Considera le  $k$  stringhe di adiacenza che è possibile ottenere permutando in tutti i modi possibili i nodi in ciascun sottogruppo.
4. Tra le  $k$  stringhe scegli quella lessicograficamente più piccola.

**Esempio.** Considerando l'immagine in figura 10.7 il grafo ha etichette  $\{a, b\}$  e gradi  $\deg(v_1) = 3, \deg(v_2) = \deg(v_3) = \deg(v_4) = 1$ . Il partizionamento per grado produce i blocchi  $D_3 = \{v_1\}$  e  $D_1 = \{v_2, v_3, v_4\}$ .

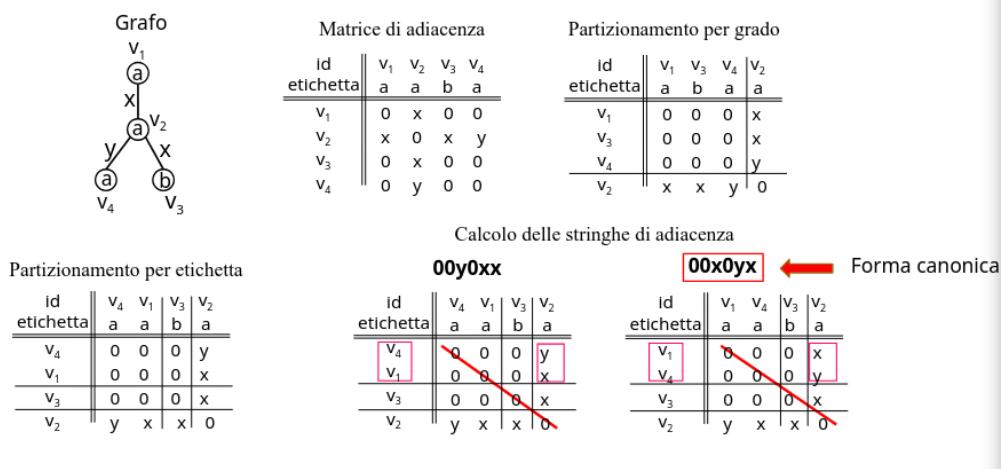


Figura 10.7: Esempio di calcolo della forma canonica di un grafo.

Raffinando<sup>3</sup> per etichetta otteniamo:

$$D_1^a = \{v_4, v_2\}, \quad D_1^b = \{v_3\}, \quad D_3^a = \{v_1\}.$$

Gli unici nodi permutabili sono quelli nel blocco  $D_1^a$ , quindi le permutazioni ammesse (mostrate in figura) generano due ordini dei vertici e, di conseguenza, due stringhe di adiacenza (si leggono le 6 voci sopra la diagonale nell'ordine  $A_{12}A_{13}A_{14}A_{23}A_{24}A_{34}$ ):

$$\pi_1 = (v_4, v_1, v_3, v_2) \Rightarrow s_1 = 00y0xx, \quad \pi_2 = (v_1, v_4, v_3, v_2) \Rightarrow s_2 = 00x0yx.$$

Assumendo l'ordinamento lessicografico  $0 < x < y$ , risulta  $00x0yx < 00y0xx$ . Pertanto, la **forma canonica** del grafo è la stringa

00x0yx

corrispondente all'ordine dei vertici  $\pi_2 = (v_1, v_4, v_3, v_2)$  mostrato nella figura.

### 10.1.7 Verifica della regola Apriori

Per ogni sottografo candidato  $G$  con  $k$  archi, l'algoritmo FSG verifica la regola Apriori controllando che tutti i sottografi di  $G$  con  $k - 1$  archi siano frequenti. Se almeno uno di questi sottografi non è frequente, allora  $G$  viene scartato come candidato.

Per contare la frequenza degli  $m$   $k$ -sottografi candidati al passo  $k$  si dovrebbero risolvere  $n \cdot k$  problemi di subgraph matching, con  $n$  numero di grafi nel database. Grazie all'indicizzazione inversa, si può ridurre il numero di problemi di subgraph matching da risolvere. L'idea è di costruire un indice che mappa ogni sottografo frequente di dimensione  $k - 1$  ai sottografi candidati di dimensione  $k$  che lo contengono.

In particolare

<sup>3</sup>In questo contesto, raggruppare i nodi per etichetta

- Ad ogni sottografo  $S$  frequente di dimensione  $k - 1$  viene associata una lista di sottografi candidati (TID List: Transaction IDentifier List) di dimensione  $k$  che lo contengono.
- Per ogni  $(k + 1)$ -sottografo candidato  $C$ , si recuperano le TID List di tutti i suoi  $k$ -sottografi e si calcola l'intersezione di queste liste per ottenere la lista dei grafi che contengono  $C$ .
- Se la dimensione della lista ottenuta è minore della soglia di supporto, allora  $C$  viene scartato come candidato.

## 10.2 Algoritmo gSpan

Un altro algoritmo popolare per l'estrazione di sotto-grafi frequenti è l'algoritmo **gSpan** (graph-based Substructure pattern mining). gSpan utilizza un approccio di **pattern-growth** per generare i candidati, evitando la necessità di eseguire join tra sottografi frequenti. In modo analogo si potrebbe pensare l'approccio *pattern-growth* come una **depth-first search** (DFS) nello spazio dei sottografi, dove si parte da un sottografo frequente e si aggiungono nodi o archi per generare nuovi sottografi candidati.

### 10.2.1 Visita DFS

La visita DFS è un algoritmo di ricerca che esplora il grafo dando priorità ai nodi più profondi prima di tornare indietro. Gli archi vengono esplorati infatti a partire dall'**ultimo nodo** scoperto  $v$  che presenta archi non ancora esplorati. Terminata l'esplorazione di tutti gli archi uscenti da  $v$ , si torna indietro al nodo precedente nella pila e si continua l'esplorazione da lì. Un esempio può essere visto in figura 10.8.

**Albero DFS.** Durante la visita DFS di un grafo  $G$  viene prodotto un albero  $T$ , chiamato *albero DFS*, i cui nodi sono i nodi di  $G$ , mentre gli archi sono gli archi di  $G$  esplorati durante la visita e che hanno portato alla scoperta di nuovi nodi.

Gli archi durante la costruzione dell'albero vengono divisi in due tipi:

- **Archi forward:** archi di  $G$  presenti in  $T$  che collegano un nodo a uno dei suoi discendenti nell'albero DFS.
- **Archi back:** archi di  $G$  non presenti in  $T$  che collegano un nodo a uno dei suoi antenati nell'albero DFS.

Un esempio di albero DFS può essere visto in figura 10.9.

### 10.2.2 Codifica DFS

L'algoritmo gSpan utilizza una codifica DFS per rappresentare i sottografi. La codifica DFS è una sequenza di tuple che rappresentano gli archi del grafo in base all'ordine di visita DFS. Considerando un arco  $(u, v)$  con etichette  $l_u$  e  $l_v$  sui nodi  $u$  e  $v$  e  $l_{(u,v)}$  sull'arco,

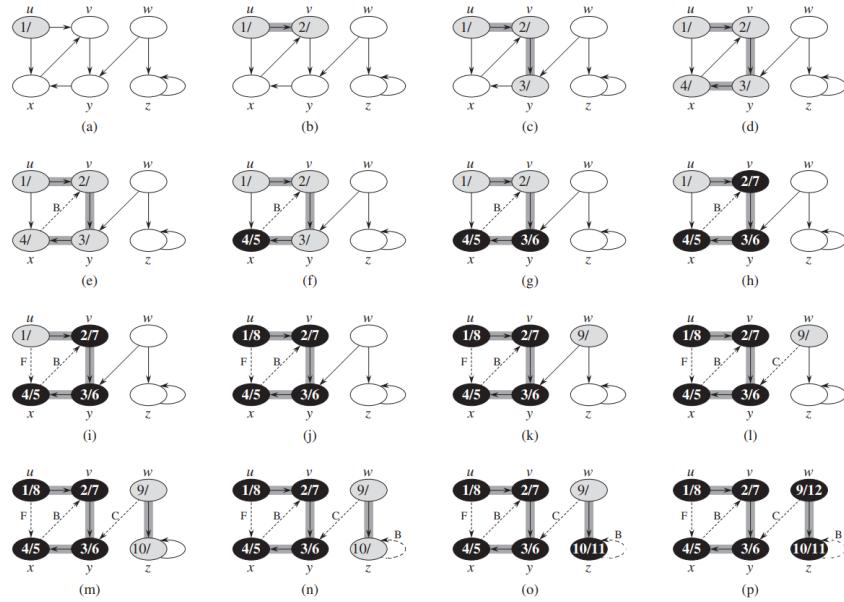


Figura 10.8: Esempio di visita in profondità (DFS) su un grafo diretto. I numeri all'interno dei nodi indicano i tempi di scoperta e di completamento (d/f) di ciascun vertice. I nodi colorati in grigio o nero rappresentano rispettivamente i vertici scoperti e completati, mentre le frecce tratteggiate indicano archi di ritorno o di attraversamento classificati durante l'esecuzione.

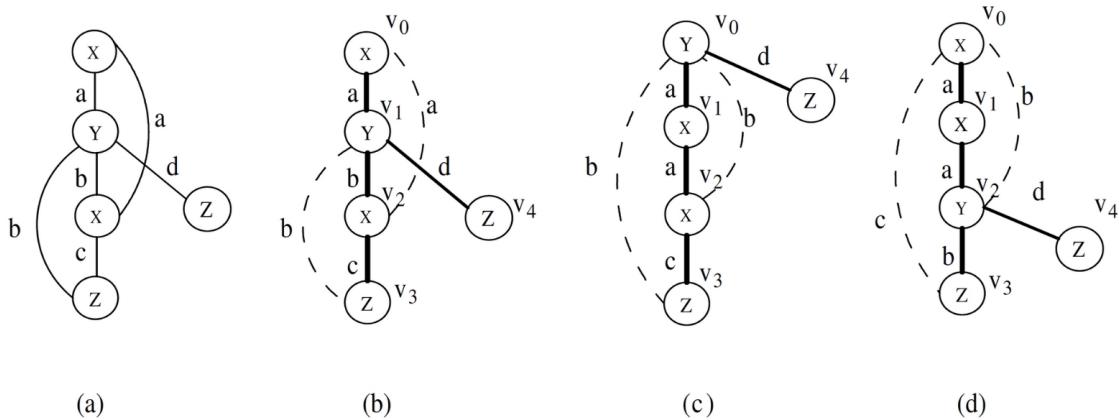


Figura 10.9: A sinistra il grafo di partenza; a destra alcuni alberi di visita in profondità (DFS) ottenuti con diversi ordini di esplorazione. Gli archi forward (dell'albero) sono in nero continuo, mentre gli archi backward sono tratteggiati, come indicato in legenda.

la tupla che rappresenta l'arco è definita come:

$$\langle i, j, l_u, l_{(u,v)}, l_v \rangle$$

dove  $i$  e  $j$  indicano **il tempo di visita** del nodo  $u$  e del nodo  $v$  durante la visita DFS.

**Costruzione della codifica DFS.** Per costruire la codifica DFS di un grafo  $G$  si seguono i seguenti passi:

1. Dato un nodo  $v$ , tutti i suoi *archi uscenti backward* devono essere elencati per primi, ordinati in base alle etichette dei nodi di destinazione e delle etichette degli archi.
2. Tra gli *archi forward* che partono dallo stesso nodo  $v$ , si seleziona l'arco con destinazione un nodo già visitato prima:  $(u, v) < (u, v') \Leftrightarrow t(v) < t(v')$ .
3. Tra gli *archi forward* che partono da nodi diversi, si seleziona quello con sorgente con tempo di visita minore:  $(u, v) < (u', v') \Leftrightarrow t(u) < t(u')$ .
4. Tra gli *archi backward* che partono dallo stesso nodo, si seleziona quello con destinazione un nodo visitato prima, ovvero con tempo di visita minore:  $(u, v) < (u, v') \Leftrightarrow t(v) < t(v')$ .
5. Tra gli *archi backward* che partono da nodi diversi, si seleziona quello con sorgente con tempo di visita minore:  $(u, v) < (u', v') \Leftrightarrow t(u) < t(u')$ .

Si trova la codifica DFS minima del grafo in figura 10.9 nella tabella 10.1.

edge no.	(b) $\alpha$	(c) $\beta$	(d) $\gamma$
0	$(0, 1, X, a, Y)$	$(0, 1, Y, a, X)$	$(0, 1, X, a, X)$
1	$(1, 2, Y, b, X)$	$(1, 2, X, a, X)$	$(1, 2, X, a, Y)$
2	$(2, 0, X, a, X)$	$(2, 0, X, b, Y)$	$(2, 0, Y, b, X)$
3	$(2, 3, X, c, Z)$	$(2, 3, X, c, Z)$	$(2, 3, Y, b, Z)$
4	$(3, 1, Z, b, Y)$	$(3, 0, Z, b, Y)$	$(3, 0, Z, c, X)$
5	$(1, 4, Y, d, Z)$	$(0, 4, Y, d, Z)$	$(2, 4, Y, d, Z)$

Tabella 10.1: Codifica degli archi per i tre schemi (b)  $\alpha$ , (c)  $\beta$  e (d)  $\gamma$ .

**Codice DFS minimo.** Si definisce il particolare, tra tutti i possibili codici DFS di un grafo  $G$ , il **codice DFS minimo** come il codice che è lessicograficamente più piccolo tra tutti i codici DFS di  $G$ . Questo codice viene utilizzato come rappresentazione unica del grafo per evitare ridondanze durante la generazione dei candidati. Nella figura 10.9 il codice DFS minimo, rappresentato in tabella 10.1, corrisponde allo schema: (d)  $\gamma$ .

**DFS Code Tree.** Sempre dalla visita DFS e dall'albero DFS si può generare un albero chiamato **DFS Code Tree**, in cui ogni nodo rappresenta un codice DFS minimo di un sottografo. La radice dell'albero rappresenta il codice vuoto, il primo livello rappresenta i sottografi con un arco, il secondo livello rappresenta i sottografi con due archi, il livello

$k$  rappresenta i sottografi con  $k + 1$  archi, e così via. Ogni figlio di un nodo rappresenta un'estensione del sottografo rappresentato dal nodo padre, ottenuta aggiungendo un arco.

### 10.2.3 Generazione dei candidati

In gSpan un sottografo candidato con  $k + 1$  archi viene generato per aggiunta di un singolo arco a partire da un sottografo frequente con  $k$  archi. Per non generare ridondanze, gSpan permette di aggiungere archi solo in due modi:

- Un arco da un nodo che sta nel cammino **più a destra** dalla radice ad un nodo foglia nell'albero DFS associato al codice DFS minimo.
- Un arco dal nodo foglia del cammino *più a destra* dell'albero DFS associato al codice DFS minimo ad un nuovo nodo (qualsiasi).

Un esempio può essere visto in figura 10.10.

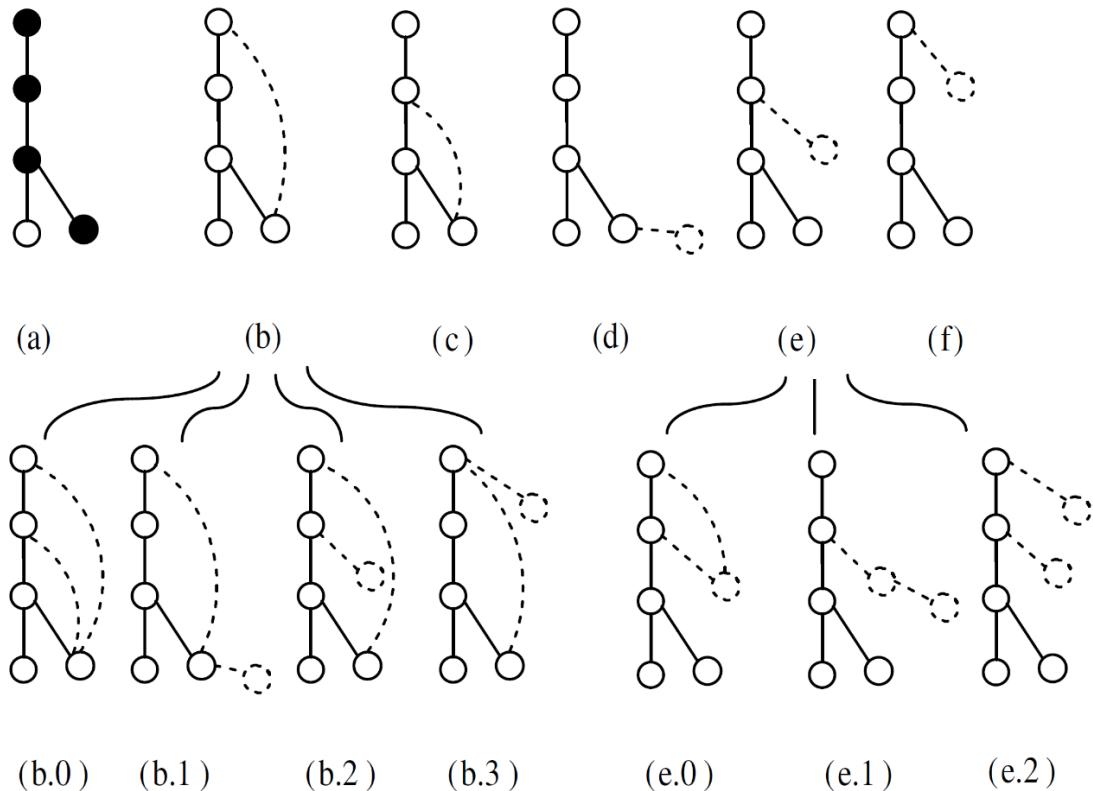


Figura 10.10: Esempio di generazione dei candidati in GSPAN. (a) Sottografo frequente di partenza. I candidati con  $k+1$  archi si ottengono aggiungendo un solo arco secondo la regola del *cammino più a destra* del codice DFS minimo: (b-d) aggiunte *backward* da un nodo del cammino più a destra verso un antenato; (e-f) aggiunte *forward* dal nodo foglia del cammino più a destra verso un nuovo nodo. Gli archi tratteggiati indicano l'arco aggiunto. La riga inferiore (b.0-b.3, e.0-e.2) illustra le varianti non ridondanti prodotte per ciascuna estensione rispettando l'ordine del codice DFS minimo.

**Pruning dello spazio di ricerca.** I grafi candidati con  $k + 1$  archi generati a partire da un grafo frequente con  $k$  archi vengono processati seguendo l'ordine lessicografico dei loro codici DFS minimi. Se generiamo un grafo candidato  $G_1$  con lo stesso codice di un grafo  $G_0$  già esaminato, possiamo fare un pruning dell'intero sottoalbero radicato in  $G_1$ , evitando di esaminare ulteriori grafi candidati generati a partire da  $G_1$ . Questo perché tutti i grafi in questo sottoalbero avranno codici DFS minimi maggiori o uguali a quello di  $G_1$ , e quindi saranno già stati esaminati quando abbiamo esaminato  $G_0$ .

## Riferimenti

I riferimenti di questo capitolo includono:

- Materiale visto a lezione.
- Approfondimenti nel libro [3].



# Capitolo 11

## Elementi di Reti neurali

Una rete neurale è un modello computazionale ispirato alla struttura e al funzionamento del cervello umano. È composta da unità chiamate neuroni artificiali, organizzati in strati (layer), che elaborano informazioni attraverso connessioni ponderate.

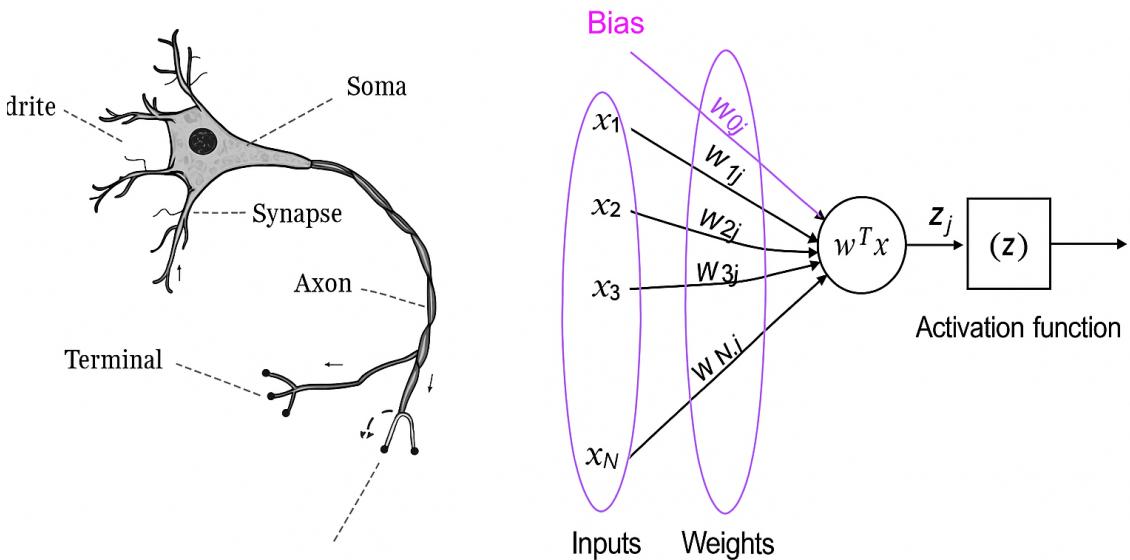


Figura 11.1: Confronto tra un neurone biologico (a sinistra) e un neurone artificiale (a destra). Nel neurone biologico, il segnale si propaga dai dendriti, attraverso il soma e lungo l'assone fino ai terminali sinaptici. Nel neurone artificiale, gli ingressi  $x_i$  vengono pesati con i corrispondenti pesi  $w_i$ , sommati e combinati con un termine di bias; il risultato  $z_j$  viene poi trasformato da una funzione di attivazione per generare l'uscita del neurone.

La struttura di una rete neurale è generalmente definita come una sequenza di layer:

- **Input layer:** (strato di ingresso) riceve i dati grezzi e li trasmette agli strati successivi.

- **Hidden layers:** (strati nascosti) elaborano le informazioni ricevute dall'input layer attraverso una serie di trasformazioni non lineari.
- **Output layer:** (strato di uscita) produce il risultato finale della rete neurale, come una classificazione o una previsione.

## 11.1 Strati

Ogni strato di una rete neurale è costituito da un insieme di neuroni artificiali (esempio in figura 11.2).

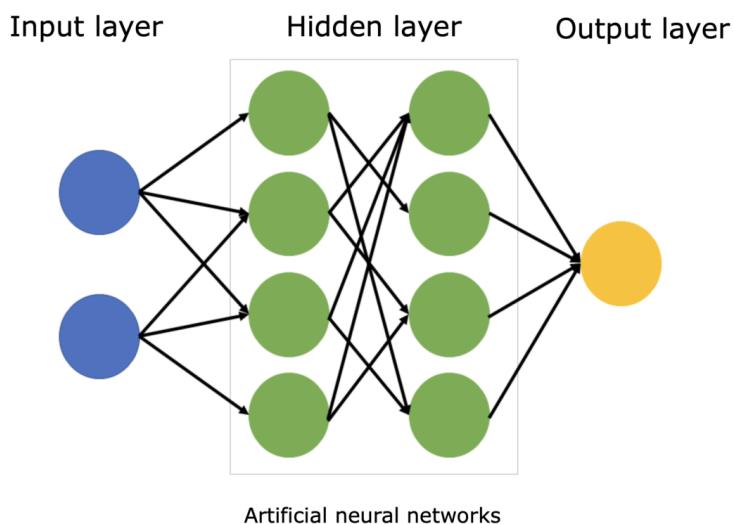


Figura 11.2: Esempio di una rete neurale con 1 input layers, 4 hidden layers e 1 output layer.

Ogni *hidden layer* è costituito da **nodi** che prendono in input *valori pesati* proveniente dallo strato precedente, li elaborano e producono un valore di output che verrà pesato ad uno o più nodi del layer successivo. L'output layer è costituito da uno o più nodi che restituiscono in output un valore.

**Deep neural network.** Si parla di *deep neural network* (DNN) quando una rete neurale possiede più di un hidden layer. Le DNN sono in grado di apprendere rappresentazioni più complesse dei dati rispetto alle reti neurali con un solo hidden layer, permettendo di risolvere problemi più sofisticati.

In generale le reti neurali lavorano con strutture dati chiamate **tensori**<sup>1</sup>.

---

<sup>1</sup>Un tensore è una struttura dati multidimensionale che generalizza i concetti di scalare (0D), vettore (1D) e matrice (2D) a dimensioni superiori. I tensori sono fondamentali nell'ambito del machine learning e delle reti neurali, poiché consentono di rappresentare e manipolare dati complessi in modo efficiente.

### 11.1.1 Connessioni tra layer

La rete neurale in figura 11.2 è un esempio di rete **densa**, poiché ogni nodo riceve tutti gli output dai nodi del layer precedente. Altre tipologie tra layer sono:

**Random** : fissato un certo  $m$ , ogni nodo riceve output solamente da  $m$  nodi random del precedente layer.

**Pooled** : i nodi di un layer sono partizionati in  $k$  cluster. Il layer successivo sarà formato da  $k$  nodi, uno per ogni cluster. Il nodo associato al cluster  $C$  riceverà solo gli output dei nodi del layer precedente appartenenti a tale certo cluster.

**Convoluzionale** : ogni nodo di un layer è connesso solo a un sottoinsieme di nodi del layer precedente, definiti da una *finestra di convoluzione* che si sposta lungo l'input. Questo tipo di connessione è particolarmente utile per l'elaborazione di dati strutturati, come immagini o segnali audio.

## 11.2 Progettare una rete neurale

Quando si addestra una rete neurale bisogna stabilire:

- Quanti layer nascosti definire.
- Quanto nodi in ciascun layer.
- Come connettere nodi di layer consecutivi.
- Quale funzione di attivazione scegliere per ogni layer.

Definita la struttura, il modello deve essere addestrato su un training set per calcolare i valori ottimali dei pesi grazie a una **funzione di costo** (o *loss function*) che misura l'errore tra le predizioni della rete e i valori reali. L'addestramento avviene tramite algoritmi di ottimizzazione come la *discesa del gradiente* (gradient descent) e il *backpropagation*, che aggiornano i pesi per minimizzare la funzione di costo.

Generalmente si va per tentativi, provando diverse architetture della rete partendo da un caso semplice, che non può dare magari buoni risultati che aiuta tuttavia capire meglio il problema e le caratteristiche dei dati.

## 11.3 Funzioni di attivazione

La funzione di attivazione di un neurone artificiale è quella funzione  $F$  che determina l'output in base all'input ricevuto. Guardando la figura 11.1, l'input del neurone artificiale è dato dalla somma pesata degli ingressi più un termine di bias:

$$z_j = \sum_i w_{ij}x_i + b_j$$

È importante notare che tutti i nodi di uno stesso layer utilizzano la stessa funzione di attivazione.

**Proprietà.** Le funzioni di attivazione devono possedere alcune proprietà:

- Devono essere **differenziabile** e **continua** per permettere l'addestramento della rete tramite algoritmi di ottimizzazione basati sul calcolo del gradiente.
- La derivata della funzione non deve *saturare*: ovvero non deve avvicinarsi a zero per valori estremi dell'input, altrimenti il processo di apprendimento diventa inefficace (in generale si ha uno stallo nell'aggiornamento dei pesi).
- Allo stesso modo, la derivata non deve "esplodere", ovvero non deve tendere a infinito, poiché ciò può causare instabilità nell'addestramento (numerica, in questo caso, nella ricerca dei pesi ottimali).

### 11.3.1 Funzione step

La funzione step (o funzione a gradino) è una funzione di attivazione semplice che restituisce 0 per input negativi e 1 per input positivi. È definita come:

$$F(z) = \begin{cases} 0 & \text{se } z < 0 \\ 1 & \text{se } z \geq 0 \end{cases}$$

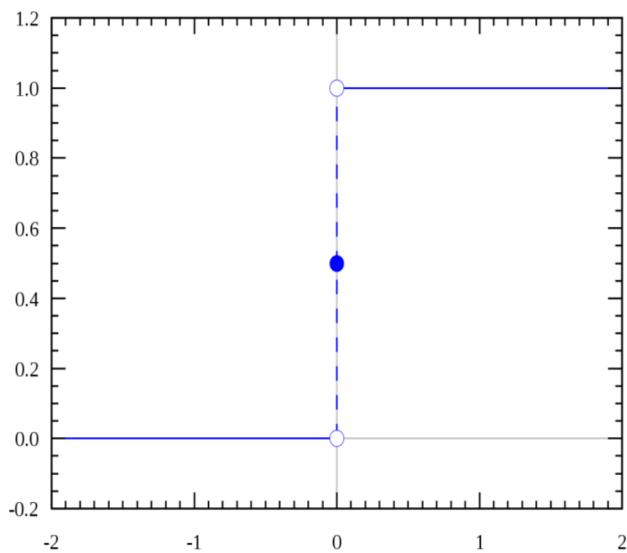


Figura 11.3: Grafico della funzione step.

La funzione step è utile per modelli di classificazione binaria, ma non è differenziabile nel punto  $z = 0$ , il che limita la sua efficacia nell'addestramento delle reti neurali tramite metodi basati sul gradiente. Un modello che usa questa funzione è il *perceptron*: un semplice modello di rete neurale con un singolo layer di nodi che utilizza la funzione step per prendere decisioni binarie.

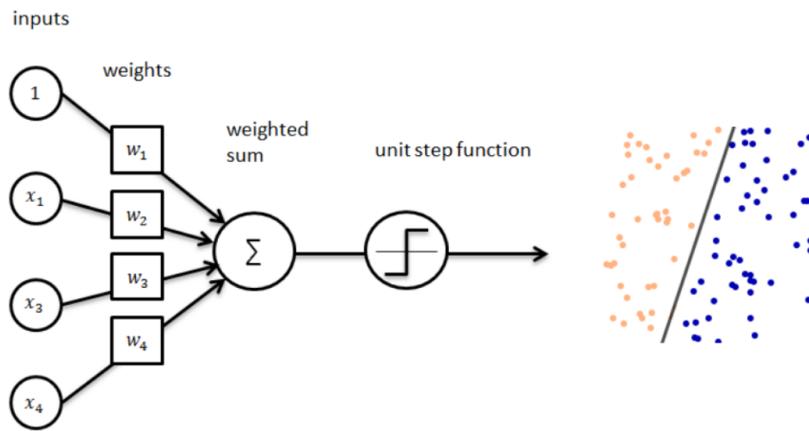


Figura 11.4: Esempio di un perceptron con 4 input, pesi associati e bias. L'output viene calcolato applicando la funzione step alla somma pesata degli input più il bias.

### 11.3.2 Funzione logistica

Un altro tipo di funzione di attivazione è la funzione logistica (o sigmoide), definita come:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

La funzione sigmoide è utilizzata spesso in reti neurali per problemi di classificazione binaria, poiché mappa qualsiasi input reale in un intervallo compreso tra 0 e 1, interpretabile come una probabilità. La funzione logistica rispecchia le proprietà richieste per una funzione di attivazione, essendo differenziabile e continua:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

tuttavia, la sua derivata può saturare per valori estremi di  $x$ , rallentando l'apprendimento in reti profonde.

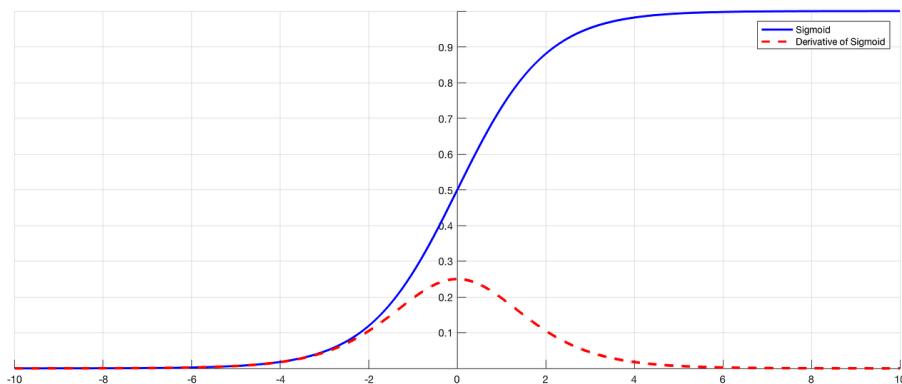


Figura 11.5: Funzione sigmoide  $\sigma(x)$  (blu) e sua derivata  $\sigma'(x)$  (rosso tratteggiato). La derivata raggiunge il massimo in  $x = 0$  (0.25); per  $|x| \gg 0$  la sigmoide satura e il gradiente tende a zero.

### 11.3.3 Tangente iperbolica

Un'altra funzione di attivazione molto simile alla sigmoide è la tangente iperbolica ( $\tanh$ ). Questa funzione mappa gli input reali nell'intervallo tra -1 e 1, ed è definita come:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

La tangente iperbolica è spesso preferita alla sigmoide perché la sua uscita è centrata intorno a zero, il che può facilitare l'apprendimento in alcune reti neurali. Questo si può notare riscrivendo la funzione in termini della sigmoide:

$$\tanh(x) = 2\sigma(2x) - 1$$

La derivata della tangente iperbolica è:

$$\tanh'(x) = 1 - \tanh^2(x)$$

Anche la tangente iperbolica può soffrire di saturazione per valori estremi di  $x$ , simile alla sigmoide, ma la sua uscita centrata intorno a zero può aiutare a mitigare questo problema in alcune situazioni.

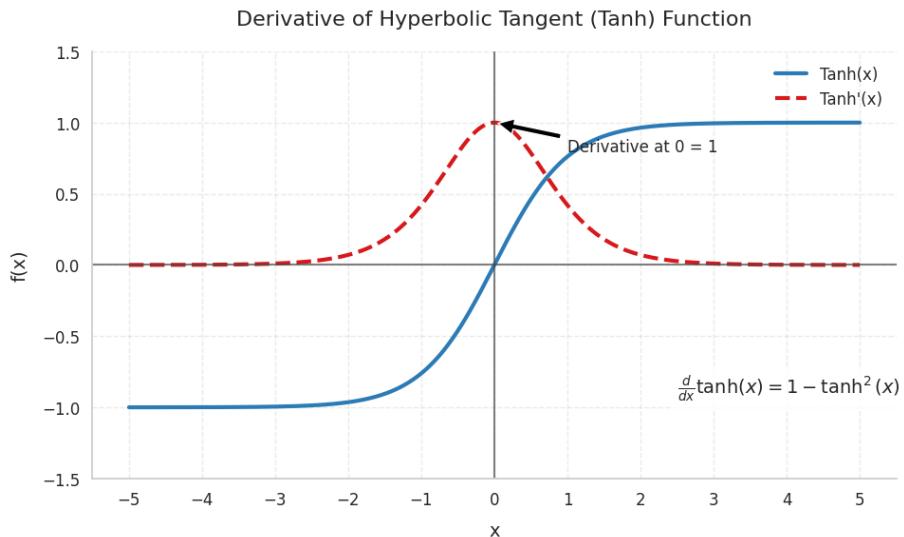


Figura 11.6: Funzione tangente iperbolica  $\tanh(x)$ : attivazione dispari, centrata in 0 con range  $(-1, 1)$ . Satura verso  $\pm 1$  per  $|x|$  grandi; derivata  $\tanh'(x)$  massima in  $x = 0$ , utile per ridurre il bias shift rispetto alla sigmoide.

### 11.3.4 Funzione softmax

A differenza della funzione sigmoide, che opera su un unico valore e ritorna un output tra 0 e 1, la funzione softmax agisce sull'intero vettore di output di un layer, trasformandolo in una distribuzione di probabilità. Sia  $x = (x_1, x_2, \dots, x_n)$  il vettore di input, la funzione

softmax è definita come:

$$\mu(x) = (\mu(x_1), \mu(x_2), \dots, \mu(x_n)) \quad \text{dove} \quad \mu(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

Si può dimostrare che restituisce una distribuzione di probabilità, sommando i singoli valori della funzione e ottenendo 1:

$$\sum_{i=1}^n \mu(x_i) = \sum_{i=1}^n \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} = 1$$

La funzione softmax è comunemente utilizzata nell'output layer di reti neurali per problemi di classificazione multi-classe, dove ogni output rappresenta la probabilità che l'input appartenga a una delle classi possibili.

Dalla formula si evince che il denominatore è una somma di esponenziali, quindi se gli input variano in un range ampio, allora anche gli esponenziali varieranno in un range ampio. Ciò può causare problemi numerici, come overflow o underflow, durante il calcolo della funzione softmax. Possiamo però sfruttare una proprietà del softmax, ovvero l'**iperparametrizzazione**: diversi valori di input possono produrre lo stesso output. In particolare, possiamo sottrarre un valore costante  $c$  da tutti gli input senza modificare l'output della funzione softmax:

$$\mu(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} = \frac{e^{x_i - c}}{\sum_{j=1}^n e^{x_j - c}}$$

Se scegliamo  $c = \max_j(x_j)$ , allora il massimo degli input diventa 0, evitando problemi di overflow negli esponenziali. Inoltre, il denominatore è una somma di valori tra 0 e 1, riducendo il rischio di underflow.

### 11.3.5 ReLU: Rectified Linear Unit

La funzione ReLU (Rectified Linear Unit) è una funzione di attivazione che prende spunto dai *raddrizzatori* a singola semionda utilizzati in elettronica: essi trasformano un segnale alternato in un segnale unidirezionale (sempre positivo o sempre negativo). La funzione ReLU è definita come:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0 & \text{se } x < 0 \\ x & \text{se } x \geq 0 \end{cases}$$

Questa funzione genera un output pari a zero per input negativi e un output lineare (uguale all'input) per input positivi.

Nella pratica, la funzione ReLU non satura mai per valori positivi di  $x$ , il che aiuta a mitigare il problema del gradiente che scompare (vanishing gradient) durante l'addestramento delle reti neurali profonde. Tuttavia, per input negativi, la derivata della ReLU è zero, il che può portare al problema dei "neuroni morti" (dead neurons), dove alcuni neuroni non si attivano mai durante l'addestramento.

**Variante ELU.** Per affrontare e risolvere il problema dei neuroni morti, esistono delle varianti della ReLU, come la Leaky ReLU e la Exponential Linear Unit (ELU). La ELU è definita come:

$$\text{ELU}(x) = \begin{cases} x & \text{se } x \geq 0 \\ \alpha(e^x - 1) & \text{se } x < 0 \end{cases}$$

dove  $\alpha$  è un iperparametro positivo che controlla il valore di saturazione per input negativi. La ELU permette un piccolo gradiente per input negativi, riducendo il rischio di neuroni morti e migliorando l'apprendimento della rete.

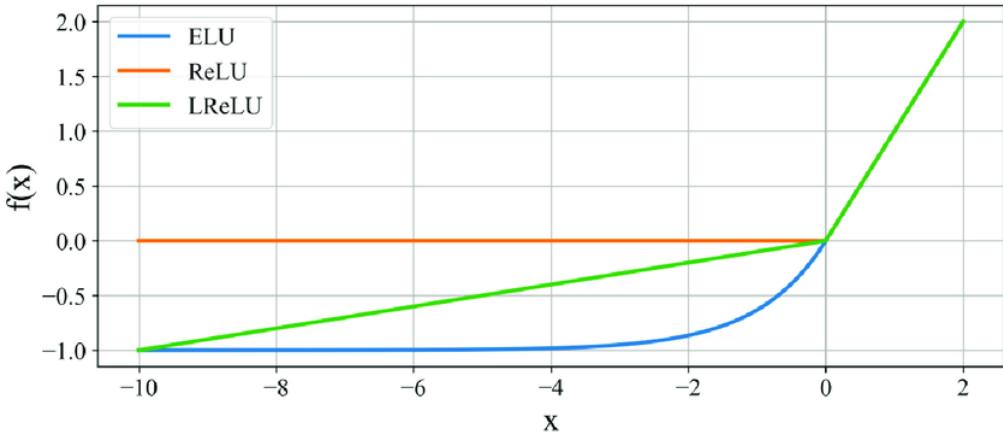


Figura 11.7: Grafici della funzione ReLU (rosso), della sua variante Leaky ReLU (verde) che permette un piccolo gradiente per input negativi, e della ELU (blu) che introduce una componente esponenziale per input negativi.

## 11.4 Funzioni Loss

Il problema nelle reti neurali è lo stesso che si ha in modelli più piccoli come la classificazione o la regressione: serve definire un modo di misurare l'errore tra le predizioni del modello e i valori reali, così che si possa addestrare il modello per minimizzare tale errore. Questa misurazione dell'errore è definita tramite una **funzione di costo** (o *loss function*).

La loss function è generalmente applicata sui pesi della rete neurale generando **pesi ottimali** per i nodi della rete. L'addestramento della rete avviene tramite algoritmi di ottimizzazione come la *discesa del gradiente* (gradient descent) e il *backpropagation*, che aggiornano i pesi per minimizzare la funzione di costo.

### 11.4.1 Regression Loss

In un problema di regressione, l'obiettivo è prevedere un valore continuo. Ipotizziamo quindi una regression loss function che misuri l'errore tra il valore predetto  $\hat{y}$  e il valore reale  $y$ . Chiamiamo questa funzione  $L$ :

$$L(\hat{y}, y) = (\hat{y} - y)^2$$

Già questa funzione, tuttavia, presenta un problema: per valori di errore molto grandi, la funzione cresce in modo quadratico, il che può portare a problemi numerici durante l'addestramento della rete. Una soluzione è utilizzare la funzione di **Huber loss**, che combina la perdita quadratica per piccoli errori e la perdita lineare per grandi errori:

$$L_\delta(\hat{y}, y) = \begin{cases} (y - \hat{y})^2 & \text{se } |y - \hat{y}| \leq \delta \\ 2\delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{se } |y - \hat{y}| > \delta \end{cases}$$

dove  $\delta$  è un iperparametro che determina il punto di transizione tra la perdita quadratico e lineare.

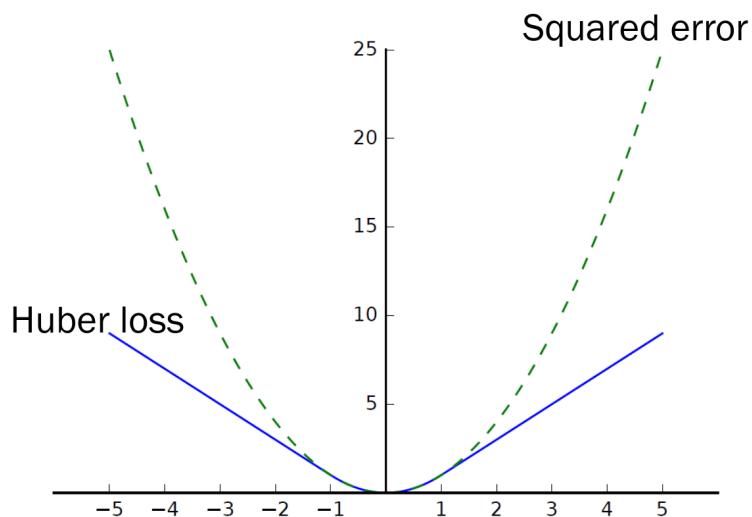


Figura 11.8: Confronto tra la squared error loss (tratteggiata, crescita quadratico) e la Huber loss (linea continua, crescita lineare oltre la soglia): la Huber loss penalizza meno fortemente gli errori molto grandi, risultando più robusta agli outlier.

**MSE: Mean Squared Error.** Una funzione di costo comunemente usata nei problemi di regressione è il **Mean Squared Error** (MSE), che calcola la media degli errori quadratici tra le predizioni e i valori reali su un dataset:

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2$$

dove  $N$  è il numero di campioni nel dataset,  $\hat{y}_i$  è la predizione per il campione  $i$  e  $y_i$  è il valore reale corrispondente. Un problema del MSE è che penalizza fortemente gli errori grandi, il che può essere problematico in presenza di outlier nei dati. Si può risolvere utilizzando la radice quadrata dell'MSE, chiamata **Root Mean Squared Error** (RMSE):

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^N (\hat{y}_i - y_i)^2}$$

La RMSE ha la stessa unità di misura delle predizioni e dei valori reali, rendendo più intuitiva l'interpretazione dell'errore medio.

### 11.4.2 Classification Loss

Nei problemi di classificazione, l'obiettivo è assegnare un'etichetta discreta a ciascun input. Il problema in questo caso, diverso da quello di regressione, è che l'output della rete neurale spesso non è un'etichetta diretta, ma una distribuzione di probabilità sulle possibili classi (ad esempio tramite la funzione softmax). Per misurare l'errore tra la distribuzione predetta e la distribuzione reale (spesso rappresentata come un vettore one-hot), si utilizzano funzioni di costo specifiche per la classificazione.

Consideriamo un problema di classificazione e siano  $C_1, C_2, \dots, C_n$  le classi possibili. Supponiamo che il training set sia formato da coppie  $(x, p)$  dove  $x$  è il vettore di input e  $p = (p_1, p_2, \dots, p_n)$  è una distribuzione di probabilità dove  $p_i$  indica la probabilità che  $x$  appartenga alla classe  $C_i$ . Supponiamo che la rete neurale produca in output una distribuzione di probabilità  $q = (q_1, q_2, \dots, q_n)$ , dove  $q_i$  è la probabilità predetta per la classe  $C_i$ , possiamo definire la funzione di costo come una misura della differenza tra le distribuzioni  $p$  e  $q$ .

**Entropia.** Per misurare la differenza tra due distribuzioni di probabilità è possibile utilizzare l'**entropia**, in quanto essa misura l'incertezza associata a una distribuzione di probabilità.

Il *Teorema di Shannon* afferma che, in un sistema di codifica ottimale, il numero di bit per simbolo necessario per rappresentare un messaggio è pari all'entropia della sorgente di informazione che genera il messaggio:

$$H(p) = - \sum_{i=1}^n p_i \log(p_i)$$

dove  $-\log p_i$  rappresenta il numero di bit necessari per codificare l'evento  $C_i$  con probabilità  $p_i$ , misura comunemente nota come **self-information**.

In altre parole, l'entropia fornisce un limite inferiore alla quantità di informazione necessaria per codificare i dati in modo efficiente.

**Entropia incrociata.** Supponendo di voler cambiare lo schema di codifica, utilizzando una distribuzione di probabilità  $q$  diversa dalla distribuzione reale  $p$ . Questo implica che per il nuovo schema occorrono  $-\log q_i$  bit per codificare l'evento  $C_i$ . La quantità media di bit necessari per codificare un messaggio generato dalla distribuzione  $p$  utilizzando lo schema di codifica basato su  $q$  è data dall'**entropia incrociata** (cross-entropy):

$$C(p||q) = - \sum_{i=1}^n p_i \log(q_i)$$

Possiamo utilizzare l'entropia incrociata per misurare la differenza tra la distribuzione reale  $p$  e la distribuzione predetta  $q$  dalla rete neurale. Questa differenza viene chiamata

**Kullback-Leibler divergence** (o KL divergence):

$$KL(p||q) = C(p||q) - H(p) = \sum_{i=1}^n p_i \log\left(\frac{p_i}{q_i}\right)$$

e, generalmente,  $C(p||q) \geq H(p)$ , con uguaglianza se e solo se  $p = q$ .

Minimizzare la KL divergenza equivale a minimizzare l'entropia incrociata, poiché l'entropia  $H(p)$  è costante rispetto a  $q$ . Quindi, possiamo utilizzare l'entropia incrociata come funzione di costo per addestrare la rete neurale:

$$L(p, q) = - \sum_{i=1}^n p_i \log(q_i)$$

Nella pratica però, poiché minimizzare la divergenza KL equivale a minimizzare l'entropia incrociata, si utilizza direttamente quest'ultima come funzione di costo per addestrare la rete neurale.

## 11.5 Training di una rete neurale

In un problema di regressione lineare, l'obiettivo è trovare i pesi dell'iperpiano che riescono ad approssimare i dati. Per le reti neurale il ragionamento è lo stesso ma su interpretazioni geometriche più complesse.

### 11.5.1 Ottimizzazione dei pesi

L'addestramento di una rete neurale consiste nel trovare i pesi che *minimizzano* la funzione di loss su un training set di dati. Per minimizzare i pesi, si può utilizzare il metodo della **discesa del gradiente** (gradient descent), che aggiorna iterativamente i pesi della rete nella direzione del gradiente negativo<sup>2</sup> della funzione di loss.

Sia  $x = (x_1, x_2, \dots, x_n)$  il vettore dei pesi della rete neurale e sia:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}$$

la funzione di loss da minimizzare. Il gradiente di  $f$  in un punto  $x$  è il vettore delle derivate parziali di  $f$  rispetto a ciascuna variabile:

$$\nabla f(x) = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)$$

### 11.5.2 Metodo di discesa del gradiente

L'idea di base della discesa del gradiente è quella di aggiornare iterativamente il vettore dei pesi  $x$  nella direzione del gradiente negativo della funzione di loss. Dato uno stato corrente  $x^{(t)}$ , l'aggiornamento è:

---

<sup>2</sup>Un gradiente negativo, infatti, indica la direzione di massima discesa della funzione ovvero il miglior modo per minimizzare la funzione.

$$x^{(t+1)} = x^{(t)} - \eta \nabla f(x^{(t)})$$

dove  $\eta > 0$  è il *learning rate*, un parametro che controlla l'ampiezza del passo lungo la direzione di discesa.<sup>3</sup>

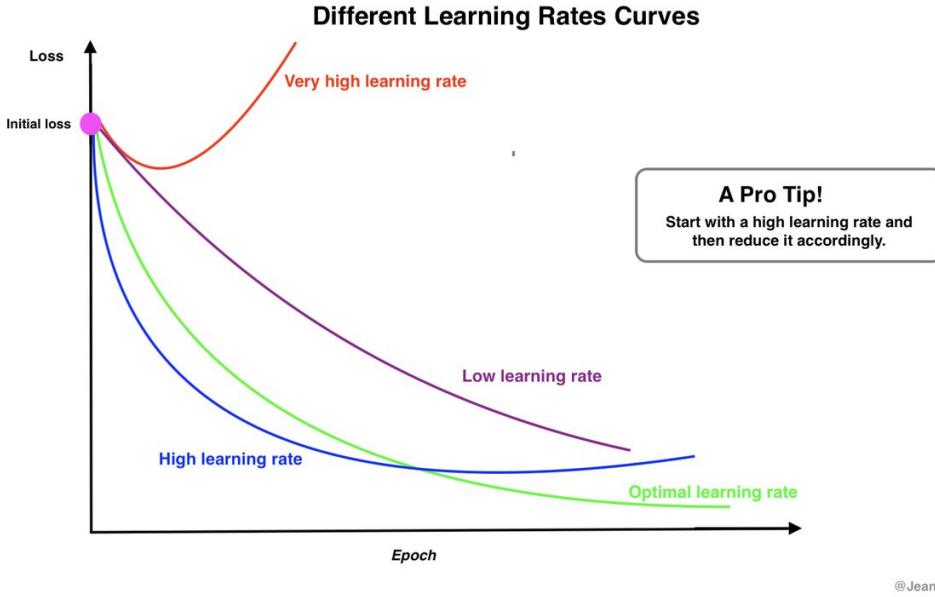


Figura 11.9: Andamento della funzione di loss al variare del *learning rate*. Un valore troppo elevato può causare instabilità o divergenza dell'addestramento, mentre un valore troppo basso rallenta significativamente la convergenza. Un *learning rate* ottimale permette una discesa rapida e stabile verso il minimo.

Fin qui abbiamo considerato  $f$  come una funzione scalare di un vettore di pesi  $x \in \mathbb{R}^n$ . Nelle reti neurali, però, la funzione di loss dipende da  $x$  in modo indiretto: i pesi determinano prima le *attivazioni* interne dei neuroni e, tramite queste, l'output finale della rete. Questo significa che la funzione di loss  $f$  può essere vista come una funzione composta:

$$f(x) = \mathcal{L}(\hat{y}, y),$$

dove  $\hat{y}$  è l'output della rete neurale (la predizione) e  $y$  è l'etichetta corretta. L'output  $\hat{y}$  dipende dai pesi  $x$  e dall'input  $u$  della rete:

$$\hat{y} = F_x(u),$$

dove  $F_x$  rappresenta la funzione computata dalla rete neurale con pesi  $x$ . Possiamo quindi vedere formalmente la rete neurale come una funzione vettoriale parametrica:

$$F_x : \mathbb{R}^d \rightarrow \mathbb{R}^k, \quad \hat{y} = F_x(u),$$

---

<sup>3</sup>Se  $\eta$  è troppo grande, l'algoritmo può divergere; se è troppo piccolo, la convergenza è molto lenta.

che mappa l'input  $u$  nello spazio delle predizioni  $\hat{y}$ . La funzione di loss si può allora scrivere come

$$f(x) = \mathcal{L}(F_x(u), y).$$

Per calcolare il gradiente  $\nabla f(x)$  in modo efficiente è fondamentale applicare la *regola della catena*<sup>4</sup> del calcolo differenziale in forma vettoriale. In questo contesto entra in gioco la **matrice Jacobiana**.

**Matrice Jacobiana.** Consideriamo una funzione vettoriale

$$g : \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad g(x) = \begin{pmatrix} g_1(x) \\ \vdots \\ g_m(x) \end{pmatrix}.$$

La *matrice Jacobiana* di  $g$  nel punto  $x$  è la matrice  $m \times n$  delle derivate parziali:

$$J_g(x) = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \frac{\partial g_1}{\partial x_2} & \dots & \frac{\partial g_1}{\partial x_n} \\ \frac{\partial g_2}{\partial x_1} & \frac{\partial g_2}{\partial x_2} & \dots & \frac{\partial g_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_m}{\partial x_1} & \frac{\partial g_m}{\partial x_2} & \dots & \frac{\partial g_m}{\partial x_n} \end{pmatrix}.$$

La Jacobiana è dunque la generalizzazione matriciale del concetto di derivata: mentre il gradiente  $\nabla f(x)$  descrive come una funzione *scalare* varia al variare delle sue variabili, la Jacobiana  $J_g(x)$  descrive come variano simultaneamente tutte le componenti di una funzione *vettoriale*  $g$ .

**Stochastic Gradient Descent.** Nelle reti neurali, il calcolo esatto del gradiente  $\nabla f(x)$  su tutto il training set può essere computazionalmente costoso, specialmente per dataset di grandi dimensioni. Per ovviare a questo problema, si utilizza spesso una variante chiamata **Stochastic Gradient Descent** (SGD). Invece di calcolare il gradiente su tutto il dataset, l'SGD calcola una stima del gradiente utilizzando un singolo campione (o un piccolo batch di campioni) alla volta scelto casualmente.

### 11.5.3 Esempio di computazione

Ipotizziamo di voler computare la rete neurale del grafo in figura 11.10. La rete prende in input un vettore  $\mathbf{x} \in \mathbb{R}^n$  a cui moltiplica un vettore di pesi  $W$  calcolando  $\mathbf{u}$ :

$$\mathbf{u} = W \cdot \mathbf{x}$$

---

<sup>4</sup>La regola della catena permette di calcolare la derivata di una funzione composta come prodotto delle derivate delle funzioni componenti.

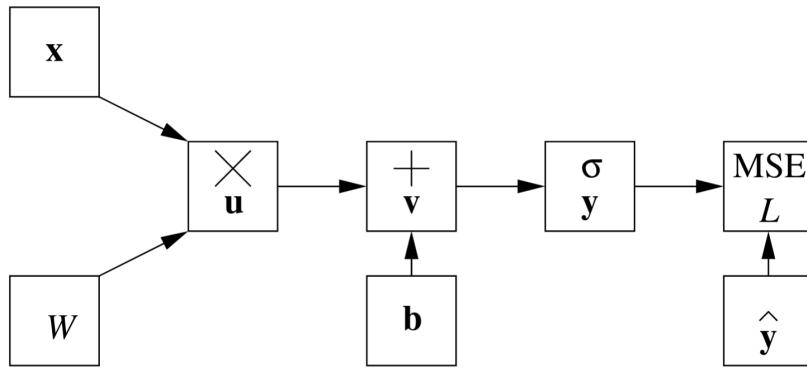


Figura 11.10: Grafo computazionale di una rete neurale feed-forward a singolo strato, con ingresso  $\mathbf{x}$ , pesi  $W$ , bias  $b$ , attivazione  $\sigma$  e perdita MSE rispetto al target  $\hat{y}$ .

Dopo aver calcolato il vettore  $\mathbf{u}$  bisogna aggiungere il vettore di bias  $b$  trovando  $\mathbf{v}$ :

$$\mathbf{v} = \mathbf{u} + b$$

Una volta calcolato il vettore  $v$  viene applicata la funzione di attivazione per computare la predizione, una sigmoide in questo caso, restituendo  $\hat{y}$ :

$$\hat{y} = \sigma(v) = \sigma(Wx + b)$$

E per valutare la bontà di questa rete si utilizza la funzione di Loss tra le predizioni e le etichette vere, MSE in questo caso, per tirare fuori  $L$ :

$$L = MSE(y, \hat{y})$$

#### 11.5.4 Backpropagation

La sola definizione di una funzione di loss e della discesa del gradiente non è ancora sufficiente per addestrare in modo efficiente una rete neurale profonda. Il problema principale è che la loss  $L$  dipende dai pesi di tutti i layer solo *indirettamente*, attraverso una lunga composizione di trasformazioni lineari e non lineari. Per poter aggiornare ogni parametro è quindi necessario calcolare, per ciascun peso  $w$ , la derivata  $\frac{\partial L}{\partial w}$ .

La backpropagation risolve il cosiddetto *credit assignment problem*: dato l'errore complessivo  $L$  osservato in uscita, stabilisce "di chi è la colpa" nei vari layer interni, assegnando a ciascun neurone un contributo di errore proporzionale alla sua responsabilità.

**Regola della catena.** Per fare questo, viene sfruttata la **regola della catena** (chain rule) del calcolo differenziale: se  $y = g(x)$ ,  $z = f(y) = f(g(x))$ , allora la derivata di  $z$  rispetto a  $x$  è:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

E questa regola può essere applicata anche su funzioni a più variabili: se

$$\begin{aligned} y &= (y_1, y_2, \dots, y_n) \\ z &= f(y) = f(y_1, y_2, \dots, y_n) \\ &= f(g_1(x), g_2(x), \dots, g_n(x)), \quad \forall i \in \{1, n\} \end{aligned}$$

allora:

$$\frac{dz}{dx} = \sum_{i=1}^n \frac{\partial z \cdot dy_i}{\partial y_i \cdot dx}$$

Questa regola può essere applicata anche a funzioni a più variabili vettore<sup>5</sup> come nel caso delle reti neurali, utilizzando gradienti e matrici jacobiane. Se  $y = g(x)$  e  $z = f(y) = f(g(x))$  allora:

$$\nabla_x z = J_x(y)^\top \cdot \nabla_y z$$

dove  $J_x(y)$  è la matrice Jacobiana di  $g$  calcolata in  $x$ .

Questo permette di calcolare efficientemente le derivate parziali della funzione di loss rispetto a tutti i pesi della rete neurale, propagando l'errore all'indietro attraverso la rete, dal layer di output fino ai layer di input. In questo modo, ogni peso viene aggiornato in base al suo contributo all'errore complessivo, permettendo alla rete di apprendere dai dati in modo efficace.

**Esempio.** Ipotizziamo di voler applicare la backpropagation alla rete neurale del grafo in figura 11.10. Indichiamo con

$$g(z) = \nabla_z L$$

il gradiente della loss rispetto alla variabile  $z$ . In questo esempio  $y$  è l'output della rete e  $\hat{y}$  il vettore target.

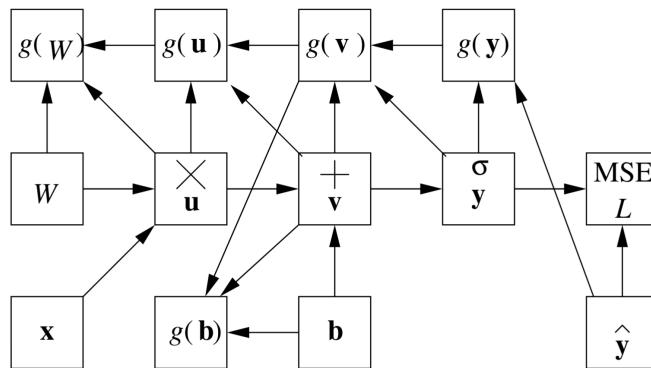


Figura 11.11: Grafo di backpropagation per la rete neurale in figura 11.10.

Per applicare la backpropagation, si procede come segue:

---

<sup>5</sup>Una variabile vettore è una variabile che può assumere valori rappresentati come vettori, ovvero insiemi ordinati di numeri. Ad esempio, un vettore in uno spazio tridimensionale può essere rappresentato come  $(x, y, z)$ , dove  $x$ ,  $y$  e  $z$  sono le componenti del vettore lungo ciascuna delle tre dimensioni.

- Si parte dall'ultimo nodo del grafo, la funzione di loss (MSE), e si calcola la derivata della loss rispetto all'output della rete neurale  $y$ . Per una singola istanza

$$L = \|y - \hat{y}\|^2 = \sum_i (y_i - \hat{y}_i)^2,$$

da cui, derivando rispetto a  $y$ ,

$$g(y) = \nabla_y L = \frac{\partial L}{\partial y} = 2(y - \hat{y}).$$

- Si propaga quindi l'errore all'indietro attraverso la funzione di attivazione  $y = \sigma(v)$ , applicata componente per componente. La Jacobiana di  $y$  rispetto a  $v$  è diagonale:

$$J_v(y) = \text{diag}(y \circ (1 - y)),$$

e la regola della catena in forma vettoriale dà

$$g(v) = J_v(y) g(y).$$

Poiché  $J_v(y)$  è diagonale, questa moltiplicazione corrisponde a un prodotto elemento per elemento:

$$g(v)_i = y_i(1 - y_i) g(y)_i \quad \text{ovvero} \quad g(v) = (y \circ (1 - y)) \circ g(y).$$

- Consideriamo ora il nodo di somma  $v = u + b$ . Ogni componente soddisfa  $v_i = u_i + b_i$ , quindi le Jacobiane rispetto a  $u$  e  $b$  sono matrici identità:

$$J_u(v) = I, \quad J_b(v) = I.$$

Applicando di nuovo la regola della catena otteniamo

$$g(u) = J_u(v) g(v) = g(v), \quad g(b) = J_b(v) g(v) = g(v).$$

Nel grafo di backpropagation (figura 11.11) questo corrisponde alle due frecce che partono da  $g(v)$  e arrivano a  $g(u)$  e  $g(b)$ : lo stesso segnale di errore viene copiato sui due rami.

- Infine propaghiamo l'errore fino ai pesi  $W$ , passando per il nodo di prodotto  $u = Wx$ . Scriviamo  $W$  per righe come

$$W = \begin{pmatrix} w_1^\top \\ \vdots \\ w_n^\top \end{pmatrix}, \quad u_i = w_i^\top x.$$

Per ogni riga  $w_i$  la Jacobiana di  $u$  rispetto a  $w_i$  ha tutti zeri tranne nella posizione

relativa a  $u_i$ , dove compare  $x$ ; di conseguenza

$$g(w_i) = J_{w_i}(u) g(u) = g(u_i) x.$$

Mettendo insieme tutte le righe otteniamo la forma compatta

$$g(W) = g(u) x^\top,$$

che corrisponde al nodo  $g(W)$  nella figura 11.11: il gradiente rispetto all'intera matrice dei pesi è il prodotto esterno tra il vettore di errori sul layer e il vettore di input  $x$ .

## 11.6 Tecniche di Regolarizzazione

Un problema che può nascere da modelli troppo complessi è l'**overfitting**, ovvero la capacità del modello di adattarsi troppo bene ai dati di training, perdendo la capacità di generalizzare a dati nuovi. Per risolvere questo problema, si utilizza la *regolarizzazione*, ovvero l'aggiunta di un termine di penalizzazione alla funzione di loss che scoraggia soluzioni troppo complesse. Questo funziona perché questo termine va a penalizzare i pesi troppo grandi, che spesso sono associati a modelli complessi.

### 11.6.1 Regolarizzazione L1 e L2

Le regolarizzazioni più comuni sono la **regolarizzazione L1** e la **regolarizzazione L2**:

**Regolarizzazione L1:** Aggiunge alla funzione di loss un termine proporzionale alla somma dei valori assoluti dei pesi:

$$L_{\text{reg}} = \lambda \sum_i |w_i|$$

dove  $\lambda$  è un iperparametro che controlla l'importanza della regolarizzazione. La regolarizzazione L1 tende a produrre modelli più semplici, in quanto favorisce soluzioni con molti pesi esattamente uguali a zero, portando a una forma di selezione delle caratteristiche (feature selection).

**Regolarizzazione L2:** Aggiunge alla funzione di loss un termine proporzionale alla somma dei quadrati dei pesi:

$$L_{\text{reg}} = \lambda \sum_i w_i^2$$

Anche in questo caso,  $\lambda$  controlla l'importanza della regolarizzazione. La regolarizzazione L2 tende a produrre modelli con pesi più piccoli e distribuiti, riducendo la complessità del modello senza eliminare completamente nessun peso.

### 11.6.2 Dropout

Un'altra tecnica di regolarizzazione molto efficace è il **dropout**. Durante l'addestramento, il dropout consiste nel "spegnere" casualmente una frazione dei neuroni in ogni layer della rete. Questo impedisce alla rete di dipendere troppo da singoli neuroni, costringendola a imparare rappresentazioni più robuste e generalizzabili. Durante la fase di test, invece, tutti i neuroni sono attivi, ma i pesi vengono scalati per tenere conto del fatto che durante l'addestramento alcuni neuroni erano stati spenti.

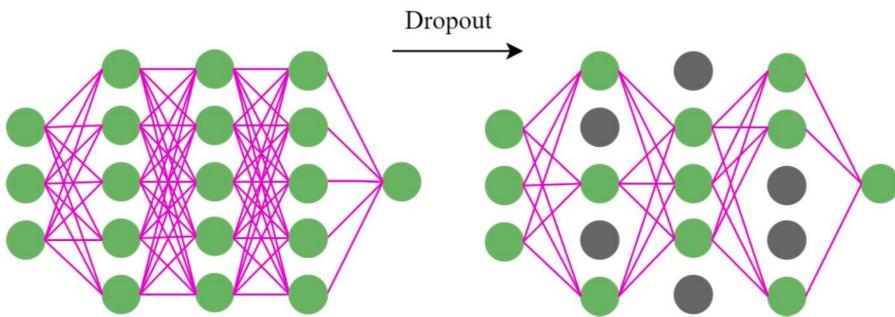


Figura 11.12: Esempio di applicazione del dropout in una rete neurale: durante l'addestramento, alcuni neuroni (in grigio) vengono "spenti" casualmente, costringendo la rete a non dipendere troppo da singoli neuroni e a imparare rappresentazioni più robuste. Durante il test, tutti i neuroni sono attivi, ma i pesi vengono scalati per tenere conto del dropout.

### 11.6.3 Early Stopping

Un problema che può sopraggiungere durante il training è trovare un minimo locale per la funzione di loss che non generalizza bene sui dati di test. Una tecnica semplice ma efficace per evitare questo problema è l'**early stopping**. Durante l'addestramento, si monitora la performance della rete su un set di validazione separato dal training set. Se la performance sul set di validazione inizia a peggiorare (ad esempio, se la loss aumenta o l'accuracy diminuisce), si interrompe l'addestramento, anche se la performance sul training set continua a migliorare. In questo modo, si evita di addestrare troppo a lungo la rete, prevenendo l'overfitting. Questo può essere effettuato salvando i pesi della rete ogni volta che la performance sul set di validazione migliora, e ripristinando i pesi migliori alla fine dell'addestramento.

Il numero di epoch senza miglioramento  $\alpha$  prima di fermare l'addestramento è un iperparametro che può essere regolato in base al problema specifico.

### 11.6.4 Aumento del training set

Altri metodi per ridurre l'overfitting includono l'**aumento del training set** (data augmentation), che consiste nel generare nuovi dati di training a partire dai dati esistenti tramite trasformazioni come rotazioni, traslazioni, zoom, ecc. Questo aiuta la rete a imparare a riconoscere pattern più generali e a non dipendere troppo da caratteristiche specifiche dei dati originali.

Per esempio, se una rete neurale è addestrata per classificare immagini si potrebbe incrementare il training set ruotando le immagini o distorcendole leggermente, in modo che la rete impari a riconoscere gli oggetti indipendentemente dalla loro posizione o orientamento.

## 11.7 Tipi di reti neurali

Generalmente le reti neurali possono essere classificate in base alla loro architettura e al tipo di dati che elaborano. Infatti, una specifica architettura può essere più adatta per un certo tipo di problema rispetto ad un'altra.

### 11.7.1 Feed-Forward Networks

Le **Feed-Forward Networks** (FFN) sono il tipo più semplice di reti neurali, in cui l'informazione fluisce in una sola direzione, dall'input all'output, senza cicli o connessioni ricorrenti. Queste reti sono composte da strati di neuroni, dove ogni neurone in uno strato è connesso a tutti i neuroni dello strato successivo. Le FFN sono comunemente utilizzate per problemi di classificazione e regressione.

### 11.7.2 Reti Neurali Convoluzionali

Le **Reti Neurali Convoluzionali** (Convolutional Neural Networks, CNN) sono progettate per elaborare dati con una struttura a griglia, come le immagini. Le CNN utilizzano operazioni di convoluzione per estrarre caratteristiche locali dai dati, sfruttando la correlazione spaziale tra i pixel. Queste reti sono particolarmente efficaci per compiti di visione artificiale, come il riconoscimento di immagini e la classificazione.

La loro architettura tipica alterna layer **convoluzionali**, che applicano filtri per estrarre caratteristiche, e layer di **pooling**, che riducono la dimensionalità dei dati mantenendo le informazioni più rilevanti.

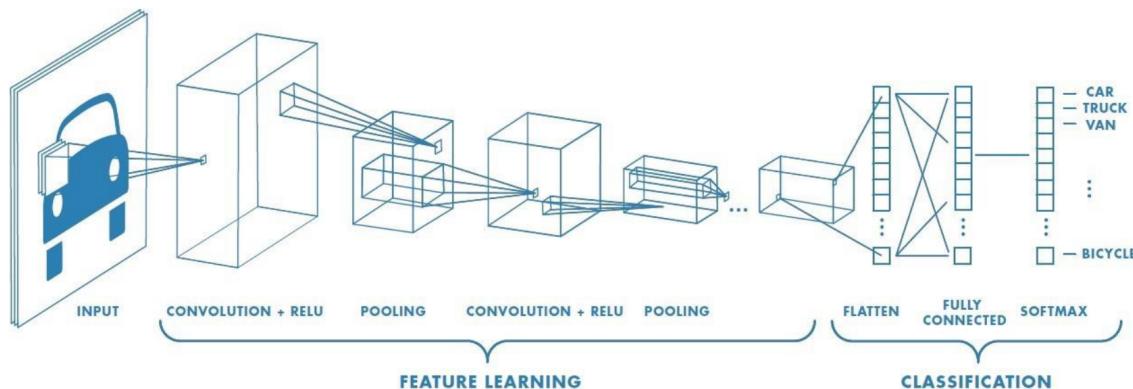


Figura 11.13: Schema di una rete neurale convoluzionale per la classificazione di immagini: dall'input grezzo si susseguono layer di convoluzione + ReLU e pooling per l'estrazione gerarchica delle feature, seguiti da livelli fully connected e da un output softmax che produce le probabilità per ciascuna classe (es. car, truck, van, bicycle).

**Fotorecettori.** Il primo layer coglie le informazioni che rappresentano i pixel essenziali delle immagini (come nei fotorecettori nell'occhio umano), ovvero i **contorni**. Il riconoscimento dei contorni non dipende dal punto di osservazione, quindi i filtri convoluzionali sono in grado di catturare queste caratteristiche indipendentemente dalla loro posizione nell'immagine.

**Convolutional Layer.** I **convolutional layer** sono composti da un insieme di filtri (o kernel) che vengono applicati all'immagine di input tramite l'operazione di convoluzione. Ogni filtro è una piccola matrice di pesi che scorre sull'immagine, calcolando prodotti scalari tra i pesi del filtro e i pixel dell'immagine. Questo processo produce delle mappe di attivazione che evidenziano la presenza di specifiche caratteristiche nell'immagine, come bordi, angoli o texture.

In particolare, lavorano su un quadrato  $k \times k$  di nodi nella griglia del layer precedente (dove  $k$  è la dimensione del filtro). Per esempio, prendendo un pixel  $x_{ij}$  il filtro viene applicato al quadrato in cui il pixel in alto a sinistra è  $x_{ij}$ :

$$z_{ij} = \sum_{i=0}^k \sum_{j=0}^k w_{ij} \cdot x_{i+j, j+i} + b$$

dove  $w_{ij}$  sono i pesi del filtro e  $b$  è il bias associato al filtro.

**Stride.** Per controllare la dimensione dell'output, si può utilizzare il parametro **stride**, che indica di quanti pixel spostarsi orizzontalmente e verticalmente dopo aver applicato il filtro. Un valore di stride maggiore di 1 riduce la dimensione dell'output, mentre uno stride di 1 mantiene la stessa dimensione (a meno di padding).

Ipotizziamo di avere un'immagine di dimensione  $W \times H$  e un filtro di dimensione  $k \times k$  con uno stride di  $s$ . La dimensione dell'output dopo l'applicazione del filtro sarà:

$$W_{out} = \frac{W - k}{s} + 1, \quad H_{out} = \frac{H - k}{s} + 1$$

**Zero Padding.** Per evitare che le dimensioni dell'output si riducano troppo rapidamente con l'applicazione di più layer convoluzionali, si può utilizzare il **zero padding**, che consiste nell'aggiungere un bordo di zeri attorno all'immagine di input. Questo permette di mantenere le dimensioni dell'output più vicine a quelle dell'input.

**Pooling Layer.** I **pooling layer** sono utilizzati per ridurre la dimensionalità delle mappe di attivazione prodotte dai convolutional layer, mantenendo le informazioni più rilevanti. Scorre in larghezza e in lunghezza (di uno step stride) una finestra di dimensione  $p \times p$  e ne calcola un valore rappresentativo, come il massimo (max pooling) o la media (average pooling) dei valori all'interno della finestra. Questo processo aiuta a ridurre il numero di parametri e a prevenire l'overfitting.

**CNN su immagini a colori.** Nel caso di un’immagine a colori, l’input è costituito da una matrice  $W \times H \times 3$ , dove i tre canali rappresentano i valori di rosso, verde e blu (RGB) per ciascun pixel. I filtri convoluzionali in questo caso hanno una profondità di 3, in modo da poter operare su tutti e tre i canali contemporaneamente. Durante la convoluzione, ogni filtro produce una mappa di attivazione che combina le informazioni provenienti dai tre canali, permettendo alla rete di apprendere caratteristiche complesse che coinvolgono tutte le componenti cromatiche dell’immagine.

Tutti i nodi di uno stesso layer convoluzionale condividono gli stessi pesi, in modo da rilevare la stessa caratteristica in diverse posizioni dell’immagine. Questo riduce significativamente il numero di parametri della rete rispetto a una rete fully connected, rendendo l’addestramento più efficiente e meno soggetto a overfitting.

### 11.7.3 Rete neurali di grafi

Le reti neurali di grafi (Graph Neural Network, GNN) hanno un funzionamento simile alle CNN ma sono progettate per lavorare su dati rappresentabili tramite grafi. La GNN produce per ogni nodo un vettore che rappresenta le informazioni principali e tali rappresentazioni possono essere poi sfruttate per risolvere i problemi di classificazioni di nodi e/o grafi oppure predizione di archi.

Le GNN si basano sul concetto di **messaggistica** (message passing), in cui i nodi comunicano tra loro per aggiornare le loro rappresentazioni in base alle informazioni dei nodi vicini. Questo processo avviene in più passaggi, in cui ogni nodo aggrega le informazioni dai suoi vicini e aggiorna la sua rappresentazione. La struttura del grafo e le connessioni tra i nodi influenzano profondamente il modo in cui le informazioni vengono propagate e aggregate.

Solitamente le GNN sono composte da 3 tipi di layer:

**Permutation Equivariant** - Questi layer sono progettati per essere invarianti alla permutazione dei nodi, ovvero l’ordine in cui i nodi vengono presentati non influenza sul risultato finale. Lavorano aggregando le informazioni dai nodi vicini e aggiornando le rappresentazioni in modo che siano indipendenti dall’ordine dei nodi.

**Local Aggregation** - Questi layer aggregano le informazioni dai nodi vicini in modo locale, considerando solo i nodi direttamente connessi. Utilizzano funzioni di aggregazione come la somma, la media o il massimo per combinare le rappresentazioni dei nodi vicini.

**Global Pooling** - Questi layer aggregano le informazioni da tutti i nodi del grafo per produrre una rappresentazione globale. Possono essere utilizzati per compiti di classificazione del grafo o per estrarre caratteristiche globali. Le funzioni di pooling comuni includono la somma, la media o il massimo delle rappresentazioni dei nodi.

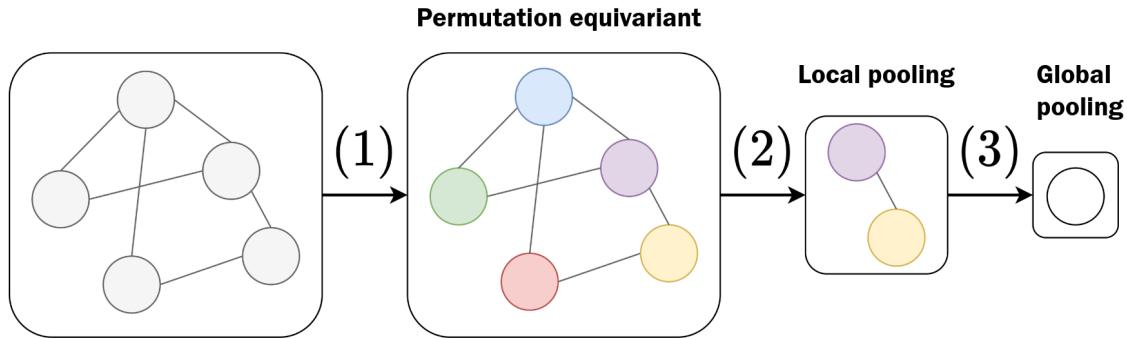


Figura 11.14: Schema generale di una rete neurale su grafi. (1) Un blocco *permutation equivariant* propaga le informazioni sul grafo e aggiorna le rappresentazioni dei nodi; (2) un'operazione di *local pooling* aggrega le caratteristiche su un sottografo rilevante; (3) un *global pooling* produce un'unica rappresentazione vettoriale dell'intero grafo.

#### 11.7.4 Reti Neurali Ricorrenti

Le **Reti Neurali Ricorrenti** (Recurrent Neural Networks, RNN) sono progettate per elaborare dati sequenziali, come serie temporali o testo. Le RNN mantengono uno stato interno che viene aggiornato ad ogni passo della sequenza, permettendo alla rete di "ricordare" informazioni precedenti e di catturare dipendenze temporali nei dati.

Si parla di reti neurali *ricorrenti* perché i neuroni in un layer possono avere connessioni che ritornano a se stessi o ad altri neuroni nello stesso layer, creando cicli nel grafo computazionale, chiamati **layer ricorrenti**.

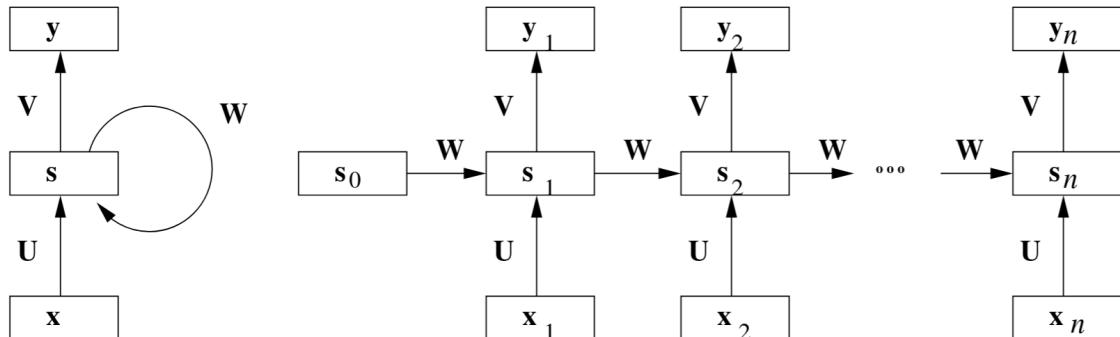


Figura 11.15: A sinistra, l'unità base di una rete neurale ricorrente (RNN), in cui lo stato nascosto  $s$  viene aggiornato combinando l'input corrente  $x$  con lo stato precedente tramite le matrici di pesi  $\mathbf{U}$ ,  $\mathbf{W}$  e genera l'output  $y$  tramite  $\mathbf{V}$ . A destra, la stessa RNN "unrolled" per  $n$  passi temporali, con la catena di stati  $s_1, \dots, s_n$  e dei corrispondenti input  $x_1, \dots, x_n$  e output  $y_1, \dots, y_n$ .

**Input di una RNN.** L'input di una RNN è una sequenza di vettori:

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

dove ogni vettore  $x_t$  rappresenta l'input al tempo  $t$ , consegnando una sequenza di output di vettori:

$$y = (y_1, y_2, \dots, y_T)$$

I pesi della rete sono condivisi da tutti i nodi del layer ricorrente e sono rappresentati da tre matrici:

- **U**: matrice dei pesi che collega l'input  $\mathbf{x}_t$  allo stato nascosto  $\mathbf{s}_t$ .
- **W**: matrice dei pesi che collega lo stato nascosto precedente  $\mathbf{s}_{t-1}$  allo stato nascosto corrente  $\mathbf{s}_t$ .
- **V**: matrice dei pesi che collega lo stato nascosto corrente  $\mathbf{s}_t$  all'output  $\mathbf{y}_t$ .

Chiamiamo  $\mathbf{s}_t$  lo stato nascosto della RNN al tempo  $t$ . Lo stato nascosto viene aggiornato combinando l'input corrente  $\mathbf{x}_t$  con lo stato precedente  $\mathbf{s}_{t-1}$ :

$$\mathbf{s}_t = f(\mathbf{U} \mathbf{x}_t + \mathbf{W} \mathbf{s}_{t-1} + \mathbf{b})$$

dove  $f$  è una funzione di attivazione non lineare (ad esempio *tanh* o ReLU) e  $\mathbf{b}$  è il bias associato allo stato nascosto.

L'output al tempo  $t$  si ottiene applicando la matrice **V** allo stato nascosto:

$$\mathbf{y}_t = \mathbf{V} \mathbf{s}_t.$$

Uno dei problemi principali di queste RNN sono le **sequenze di lunghezza variabile**. Per gestirlo, si possono utilizzare due tecniche:

**Zero Padding** - Come nel caso delle CNN, si possono aggiungere zeri alla fine delle sequenze più corte per uniformare la lunghezza di tutte le sequenze nel batch.

**Bucketing** - Si possono raggruppare le sequenze in "bucket" in base alla loro lunghezza, in modo che tutte le sequenze in un bucket abbiano la stessa lunghezza. In questo modo, si può addestrare la RNN su ciascun bucket separatamente, evitando di dover gestire sequenze di lunghezza variabile all'interno dello stesso batch.

Si possono anche **combinare** le due tecniche: si costruiscono diversi bucket che *gestiscono* sequenze di lunghezza simile, e all'interno di ogni bucket si applica lo zero padding per *uniformare* la lunghezza delle sequenze.

Un limite delle RNN, però, è l'apprendimento di relazioni "distanti", ovvero la difficoltà di catturare dipendenze a lungo termine nelle sequenze. Questo causa a livello di calcolo differenziale il problema del **vanishing gradient**, in cui i gradienti calcolati durante la backpropagation diventano molto piccoli, rendendo difficile l'aggiornamento efficace dei pesi associati a queste dipendenze lontane.

### 11.7.5 LSTM: Long Short-Term Memory

Per superare il problema del vanishing gradient nelle RNN, sono state sviluppate le **Long Short-Term Memory** (LSTM), un tipo speciale di RNN progettato per catturare

dipendenze a lungo termine nelle sequenze. Le LSTM introducono una struttura di memoria chiamata *cell state* che permette di mantenere informazioni rilevanti per periodi di tempo più lunghi. Le LSTM utilizzano tre *gates* (porte) principali per controllare il flusso di informazioni nella cella di memoria:

**Forget Gate** - Decide quali informazioni dalla cella di memoria devono essere dimenticate.

Utilizza una funzione sigmoide per produrre un vettore di valori tra 0 e 1, dove 0 significa "dimentica completamente" e 1 significa "mantieni completamente".

**Input Gate** - Determina quali nuove informazioni devono essere aggiunte alla cella di memoria. Anche questa gate utilizza una funzione sigmoide per selezionare le informazioni rilevanti.

**Output Gate** - Decide quali informazioni dalla cella di memoria devono essere utilizzate per calcolare l'output della LSTM. Utilizza una funzione sigmoide per selezionare le informazioni rilevanti.

La struttura di una LSTM è definita da:

- Uno stato nascosto  $\mathbf{H}_t$  che rappresenta la memoria corrente della rete al tempo  $t$ .
- Una cella di memoria  $\mathbf{C}_t$  che immagazzina informazioni a lungo termine.
- Le tre gates  $I_t, F_t, O_t$  (input, forget, output) che regolano il flusso di informazioni nella cella di memoria al tempo  $t$ .

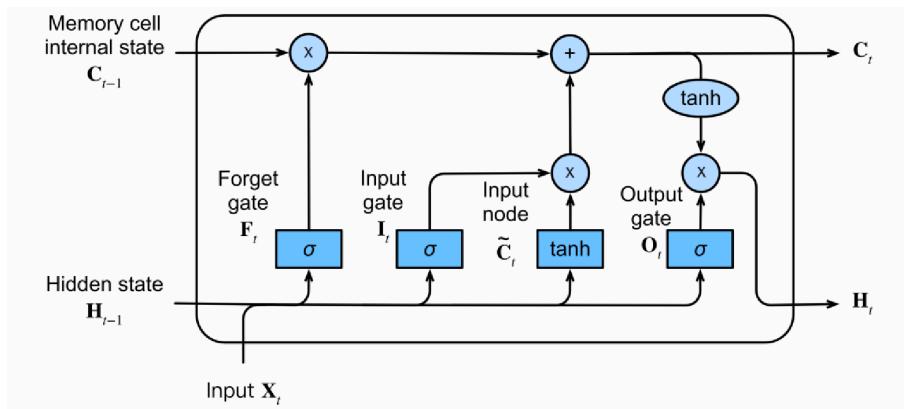


Figura 11.16: Struttura interna di una cella LSTM. La cella utilizza tre gate principali — *forget gate*, *input gate* e *output gate* — per controllare in modo dinamico il flusso di informazione nello stato interno  $C_t$  e nello stato nascosto  $H_t$ . Le attivazioni sigmoide ( $\sigma$ ) e tangente iperbolica ( $\tanh$ ) regolano rispettivamente la selezione e la trasformazione dei valori.

**Aggiornamento della cella di memoria.** I pesi di una LSTM sono rappresentati da diverse matrici:

- $\mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o$ : matrici dei pesi che collegano l'input  $\mathbf{x}_t$  alle rispettive gates (input, forget, output).

- $\mathbf{U}_i, \mathbf{U}_f, \mathbf{U}_o$ : matrici dei pesi che collegano lo stato nascosto precedente  $\mathbf{H}_{t-1}$  alle rispettive gates.
- $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o$ : bias associati alle rispettive gates.

Per aggiornare la cella di memoria e lo stato nascosto al tempo  $t$ , si seguono questi passaggi:

1. **Calcolo della forget gate.** Per prima cosa la rete decide quali informazioni della cella precedente  $\mathbf{C}_{t-1}$  devono essere *dimenticate*. Questo è il ruolo della *forget gate*  $\mathbf{F}_t$  (blocco a sinistra in figura 11.16):

$$\mathbf{F}_t = \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{H}_{t-1} + \mathbf{b}_f).$$

Ogni componente di  $\mathbf{F}_t$  è un valore tra 0 e 1: valori vicini a 0 annullano la corrispondente componente di memoria precedente, valori vicini a 1 la lasciano passare quasi inalterata.

2. **Calcolo della input gate.** In parallelo, la *input gate*  $\mathbf{I}_t$  (secondo blocco  $\sigma$  in figura 11.16) stabilisce quanto della nuova informazione candidata verrà effettivamente scritta nella cella:

$$\mathbf{I}_t = \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{H}_{t-1} + \mathbf{b}_i).$$

3. **Calcolo della nuova informazione candidata.** Il nodo interno di input  $\tilde{\mathbf{C}}_t$  (blocco  $\tanh$  al centro della figura 11.16) produce un vettore di possibili nuovi contenuti di memoria:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{U}_c \mathbf{H}_{t-1} + \mathbf{b}_c),$$

dove  $\mathbf{W}_c, \mathbf{U}_c, \mathbf{b}_c$  sono i pesi specifici del nodo di input. Qui  $\tanh$  comprime ogni componente in  $[-1, 1]$ , producendo valori “firmati” che possono sia aumentare che ridurre la memoria.

4. **Aggiornamento dello stato di cella.** A questo punto combiniamo la memoria precedente con la nuova informazione candidata. Come si vede dal ramo superiore della figura 11.16, la cella viene aggiornata come:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t,$$

dove  $\odot$  indica il prodotto elemento per elemento (prodotto di Hadamard). Il termine  $\mathbf{F}_t \odot \mathbf{C}_{t-1}$  implementa la dimenticanza controllata (una sorta di “cancella/riduci”), mentre  $\mathbf{I}_t \odot \tilde{\mathbf{C}}_t$  rappresenta la parte di nuova informazione che viene effettivamente memorizzata.

5. **Calcolo della output gate.** Non tutta la memoria aggiornata  $\mathbf{C}_t$  viene resa visibile all'esterno. La *output gate*  $\mathbf{O}_t$  (blocco  $\sigma$  a destra in figura 11.16) decide quali componenti della cella parteciperanno allo stato nascosto:

$$\mathbf{O}_t = \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{H}_{t-1} + \mathbf{b}_o).$$

**6. Aggiornamento dello stato nascosto.** Infine, lo stato nascosto  $\mathbf{H}_t$  (ramo inferiore della figura 11.16) viene ottenuto filtrando la memoria aggiornata attraverso una non linearità tanh e modulandola con la output gate:

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

ù

### 11.7.6 Autoencoder

Gli **autoencoder** sono un tipo di rete neurale utilizzata principalmente per l'apprendimento non supervisionato di rappresentazioni compatte dei dati. Un autoencoder è composto da due parti principali: un *encoder* e un *decoder*. L'encoder comprime l'input in una rappresentazione a bassa dimensionalità (chiamata *codice* o *embedding*), mentre il decoder cerca di ricostruire l'input originale a partire da questa rappresentazione compressa.

La loss function da minimizzare si basa sulla differenza tra l'input originale e la sua ricostruzione, spesso utilizzando la Mean Squared Error (MSE) o la Binary Cross-Entropy (BCE) a seconda del tipo di dati. Uno dei problemi di questa loss function è la dimensione dello **spazio latente**<sup>6</sup>  $F$ :

- Se  $F$  è troppo grande, l'autoencoder potrebbe semplicemente imparare a copiare l'input all'output senza estrarre caratteristiche significative, portando a una scarsa generalizzazione.
- Se  $F$  è troppo piccolo, l'autoencoder potrebbe non essere in grado di catturare tutte le informazioni rilevanti dell'input, risultando in una ricostruzione di bassa qualità.

**Varianti di autoencoder.** Oltre all'architettura di base, esistono diverse varianti di autoencoder progettate per affrontare specifici problemi o migliorare le prestazioni:

- **Denoising Autoencoder:** Progettato per imparare rappresentazioni robuste, questo tipo di autoencoder viene addestrato a ricostruire l'input originale a partire da una versione rumorosa dello stesso. Questo aiuta la rete a imparare caratteristiche più significative e a ridurre il rumore nei dati.
- **Sparse Autoencoder:** Introduce una penalizzazione sulla sparsità delle attivazioni nello spazio latente, incoraggiando la rete a utilizzare solo un piccolo numero di neuroni per rappresentare l'input. Questo può portare a rappresentazioni più interpretabili e a una migliore generalizzazione.

### 11.7.7 Variational Autoencoder

Una variante avanza degli autoencoder è il **Variational Autoencoder** (VAE): utilizzano la stessa struttura degli autoencoder tradizionali, ma ricostruiscono campioni da una distribuzione probabilistica nello spazio latente invece di un singolo punto. Questo consente

---

<sup>6</sup>Per spazio latente, si intende lo spazio delle rappresentazioni compresse generate dall'encoder, che dovrebbe essere di dimensione inferiore rispetto all'input originale per garantire una compressione efficace.

di generare nuovi dati simili a quelli di training, rendendo i VAE utili per compiti di generazione di dati. L'assunzione fatta è che i dati nello spazio latente siano **distribuiti** secondo una distribuzione gaussiana multivariata con media  $\mu$  e deviazione standarda  $\sigma$  e sono parametri appresi dalla rete in fase di training sulla base degli input forniti.

**Loss function dei VAE.** Nella definizione della funzione di loss dei VAE si considera la perdita di informazione nella ricostruzione dell'input e una regolarizzazione che forza la distribuzione appresa nello spazio latente a essere vicina a una distribuzione gaussiana standard (con media 0 e deviazione standard 1).

**Applicazione di VAE.** I VAE sono utilizzati in vari campi, tra cui la generazione di immagini, la modellazione del linguaggio e la sintesi di dati. La loro capacità di generare nuovi campioni simili a quelli di training li rende particolarmente utili per applicazioni creative e di simulazione.

## Riferimenti

I riferimenti di questo capitolo includono:

- Materiale visto a lezione.
- Capitolo 13 del libro [8].



## **Capitolo 12**

# **Introduzione a Transformer e LLM**



# **Parte III**

## **Approfondimenti**



# Bibliografia

- [1] Albert-László Barabási e Márton Pósfai. *Network Science*. Cambridge: Cambridge University Press, 2016. ISBN: 978-1-107-07626-6. URL: [https://assets.cambridge.org/97811070/76266/frontmatter/9781107076266\\_frontmatter.pdf](https://assets.cambridge.org/97811070/76266/frontmatter/9781107076266_frontmatter.pdf).
- [2] Vincenzo Bonnici et al. «A Subgraph Isomorphism Algorithm and its Application to Biochemical Data». In: *BMC Bioinformatics* 14.Supp1 7 (2013), S13. DOI: 10.1186/1471-2105-14-S7-S13.
- [3] Diane J. Cook e Lawrence B. Holder, cur. *Mining Graph Data*. Online ISBN: 978-0-470-07304-9. Hoboken, NJ: Wiley-Interscience, 2006, p. 512. ISBN: 978-0-471-73190-0. DOI: 10.1002/0470073047. URL: <https://onlinelibrary.wiley.com/doi/book/10.1002/0470073047>.
- [4] Luigi P. Cordella et al. «A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (2004), pp. 1367–1372. DOI: 10.1109/TPAMI.2004.75.
- [5] Luigi P. Cordella et al. «Performance Evaluation of the VF Graph Matching Algorithm». In: *Proceedings of the 10th International Conference on Image Analysis and Processing (ICIAP)*. Venice, Italy, 1999, pp. 1172–1177. DOI: 10.1109/ICIAP.1999.797762.
- [6] Trevor Hastie, Robert Tibshirani e Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second. New York, NY: Springer, 2009. ISBN: 978-0-387-84857-0. DOI: 10.1007/978-0-387-84858-7.
- [7] Gareth James et al. *An Introduction to Statistical Learning: with Applications in R*. Second. New York, NY: Springer, 2021. ISBN: 978-1-0716-1417-4. DOI: 10.1007/978-1-0716-1418-1. URL: <https://www.statlearning.com/>.
- [8] Jure Leskovec, Anand Rajaraman e Jeffrey D. Ullman. *Mining of Massive Datasets*. 2<sup>a</sup> ed. Cambridge University Press, 2014.