IML Semplice (per davvero)

Rovesti Gabriel

Attenzione

Il file non ha alcuna pretesa di correttezza; di fatto, è una riscrittura attenta di appunti, slide, materiale sparso in rete, approfondimenti personali dettagliati al meglio delle mie capacità. Credo comunque che, per scopo didattico e di piacere di imparare (sì, io studio per quello e non solo per l'esame) questo file possa essere utile. Semplice si pone, per davvero ci prova.
Thank me sometimes, it won't kill you that much.

Gabriel

Sommario

Introduzione a IML: *Machine* learning, Paradigmi di Machine Learning e fondamentali

"Si dice che un programma per computer impara dall'esperienza E rispetto a una certa classe di compiti T ed una misura delle prestazioni P, se le sue prestazioni nei compiti in T, misurate da P, migliorano con l'esperienza E".
Un esempio che si può citare:

- Example: playing checkers
  - Experience *E* = the experience of playing many games of checkers
  - Task *T* = the task of playing checkers
  - Performance measure *P* = percent of games won against opponents

Si ha quindi che un *learning algorithm*, sia un algoritmo in grado di apprendere partendo da un insieme di dati. Gli ingredienti principali sono 3:

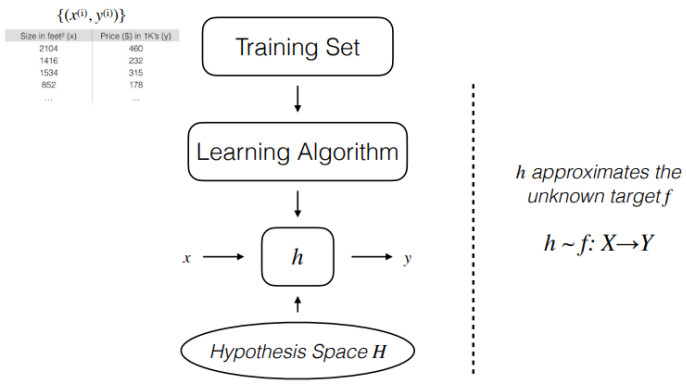
1) Il *task*, definito dal problema che vogliamo affrontare e dall'output desiderato (ad es. la classificazione del tipo di frutto);
2) La *performance measure*, intendendo quanto buono è il sistema di machine learning e la misurazione dipende dal *task*

   - Example (classification): *accuracy* is the proportion of examples for which the model gives the correct output

3) L'*experience*, data dai dati disponibili (in base a quali tipi di dati, come li otteniamo e come possono essere utilizzati)

I principali paradigmi di apprendimento (*main learning paradigms)* sono:

1) *Supervised learning*
   Obiettivo: dare la "risposta giusta" per ogni esempio nei dati
   Dati gli esempi $\{x^{(i)}, y^{(i)}\}$ imparare una funzione (descrizione) che cattura il contenuto informativo degli esempi. In sostanza, cerchiamo una funzione $h(\cdot)$ che sia in grado di mappare in modo predittivo gli $x^{(i)}$ agli $y^{(i)}$, cioè $h: X \rightarrow Y$
   Un esperto (o un insegnante) fornisce la supervisione (cioè i valori di $h(\cdot)$ corrispondenti alle istanze $x^{(i)}$)
   Output: Classificazione (a valore discreto) vs Regressione (output a valore reale)



2) *Unsupervised learning*, trovando regolarità/pattern nei dati, dunque dati esempi $\{x^{(i)}\}$, scoprire regolarità su tutto il dominio di input. Non c'è nessun esperto e nessuna supervisione.
3) *Reinforcement learning*, dove si ha un agente che potrebbe essere
    a. essere in uno stato *s*
    b. eseguire una azione *a* (tra quelle ammissibili per lo stato *s*)

    Esso opera su un ambiente *e*, che in risposta all'azione *a* nello stato *s* ritorna:
        1) il successivo stato e un premio *r* (che può essere positivo, negativo o neutrale)

    L'obiettivo dell'agente è massimizzare la funzione dei premi.

    Altre strategie possibili sono (esempi citati a scopo didattico):
    - *Weak-supervised learning* è una branca dell'apprendimento automatico in cui vengono utilizzati dati non organizzati o imprecisi per fornire indicazioni per etichettare una grande quantità di dati non supervisionati in modo che una grande quantità di dati possa essere utilizzata nell'apprendimento automatico o nell'apprendimento supervisionato.
    - *Self-supervised learning* consente ai sistemi di intelligenza artificiale di apprendere da ordini di grandezza maggiori di dati, il che è importante per riconoscere e comprendere modelli di rappresentazioni del mondo più sottili e meno comuni.
    - *Federated learning* (noto anche come apprendimento collaborativo) è una tecnica di apprendimento automatico che addestra un algoritmo su più dispositivi edge decentralizzati o server che conservano campioni di dati locali, senza scambiarli.
    - Il *Deep Learning*, la cui traduzione letterale significa *apprendimento profondo*, è una sottocategoria del Machine Learning (che letteralmente viene tradotto come apprendimento automatico) e indica quella branca dell'intelligenza artificiale che fa riferimento agli algoritmi ispirati alla struttura e alla funzione del cervello chiamate reti neurali artificiali.

Il focus principale sarà il Supervised Learning, partendo dalla classificazione (discreta) rispetto alla regressione (output reale). Si parte da:
- Un insieme di dati *D* (estratto dallo spazio di istanza *X*)
- Spazio delle ipotesi *H*, cioè l'insieme delle funzioni che possono essere implementate dal sistema di apprendimento automatico. Assumiamo che la funzione da apprendere *f* possa essere rappresentata/approssimata dall'ipotesi $h \in H$
- Un algoritmo che apprenda, visto come algoritmo di ricerca in *H*

L'esperienza da sola potrebbe non permetterci di trarre conclusioni su un'istanza di dati non visti.
Bias induttivo/inductive bias (o di apprendimento): sulla rappresentazione (H) e/o sulla ricerca (algoritmo di apprendimento), tutte le ipotesi sulla "natura" della "natura" della funzione target e della sua selezione.
Ci sono due tipi di bias:
1) Restrizione: limitare lo spazio delle ipotesi.
2) Preferenza: impone un ordine allo spazio delle ipotesi.

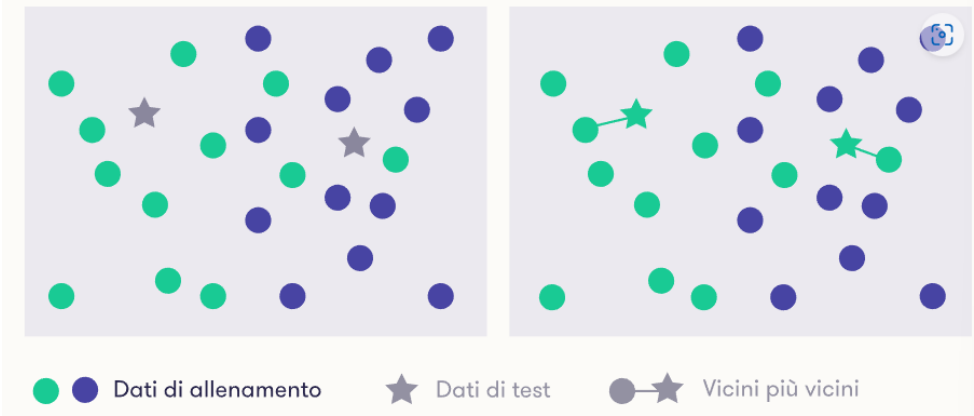
Esempi di bias induttivo:
1) Linear regression:
    L'analisi di regressione lineare viene utilizzata per prevedere il valore di una variabile in base al valore di un'altra variabile. La variabile che si vuole prevedere è chiamata variabile dipendente. La variabile che si utilizza per prevedere il valore dell'altra variabile è detta variabile indipendente. Questa forma di analisi stima i coefficienti dell'equazione lineare che coinvolge una o più variabili indipendenti che meglio predicono il valore della variabile dipendente. La regressione lineare si adatta a una linea o superficie retta che minimizza le discrepanze tra i valori previsti e quelli effettivi. Esistono semplici calcolatori di regressione lineare che utilizzano il metodo dei "minimi quadrati" per individuare la retta più adatta a una serie di dati accoppiati.

Prima di tentare di eseguire la regressione lineare, è necessario assicurarsi che i dati possano essere analizzati con questa procedura. I dati devono passare attraverso alcuni presupposti.

Ecco come verificare questi presupposti:

1) Le variabili devono essere misurate a livello continuo. Esempi di variabili continue sono il tempo, le vendite, il peso e i punteggi dei test.
2) Utilizzate un diagramma di dispersione per scoprire rapidamente se esiste una relazione lineare tra queste due variabili.
3) Le osservazioni devono essere indipendenti l'una dall'altra (cioè non devono esserci dipendenze).
4) I dati non devono presentare outlier significativi (osservazioni anomale in statistica
5) Verificate l'omoschedasticità, un concetto statistico in base al quale le varianze lungo la linea di regressione lineare migliore rimangono simili lungo tutta la linea.
6) residui (errori) della retta di regressione best-fit seguono una distribuzione normale.

2) Nearest neighbors:
Il classificatore Nearest Neighbor ("vicino più prossimo") è uno dei classificatori più semplici. Quando riceve un elemento da classificare, trova l'elemento dei dati di allenamento che assomiglia maggiormente al nuovo elemento ed emette la propria etichetta. Un esempio è fornito nel diagramma seguente.



Nel diagramma che precede mostriamo una raccolta di elementi di dati di allenamento, alcuni dei quali appartengono a una classe (verde) e altri a un'altra classe (blu). Sono inoltre presenti due elementi di dati di test, le due stelle, che classificheremo usando il metodo Nearest Neighbor.

I due elementi di test sono classificati entrambi nella classe "verde" perché i loro vicini più prossimi sono tutti e due verdi (si veda il diagramma b sopra).

La posizione dei punti nel grafico rappresenta in qualche modo le proprietà degli elementi. Poiché il diagramma è tracciato su una superficie bidimensionale piana dove è possibile muoversi in due direzioni indipendenti (su/giù o sinistra/destra), gli elementi hanno due proprietà che possiamo usare a scopo di confronto. Immaginiamo per esempio di rappresentare i pazienti di una clinica in termini di età e di livello di glucosio nel sangue. Il diagramma che precede deve comunque essere considerato soltanto come uno strumento visivo per illustrare l'idea generale, che è correlare i valori delle classi alla somiglianza o alla prossimità (vicinanza).

L'idea generale non è in alcun modo limitata alle due dimensioni e il classificatore Nearest Neighbor può essere facilmente applicato a elementi che sono caratterizzati da molte più proprietà e non solo due. L'esempio utile sono i sistemi di raccomandazione, che usano proprio questo algoritmo.

Linear regression: assume that the output or dependent variable is related to independent variable linearly (in the weights).

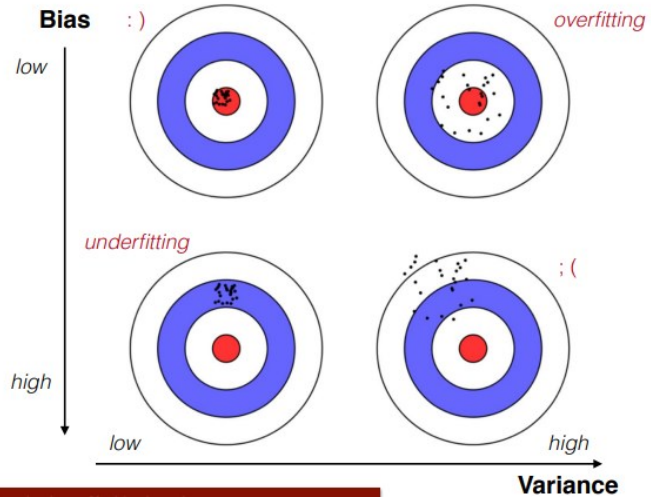
Nearest neighbors: assume that most of the cases in a small neighborhood in feature space belong to the same class.

Ci può essere anche una sorta di *algorithmic bias*, che descrive errori sistematici e ripetibili in un sistema che crea risultati ingiusti, come ad esempio privilegiare un gruppo arbitrario di utenti rispetto ad altri (bias discusso anche negli ultimi framework del GDPR).

In statistica e nell'apprendimento automatico, il bias-variance tradeoff è la proprietà di un modello secondo cui la varianza del parametro stimato tra i campioni può essere ridotta aumentando il bias dei parametri stimati. Il *dilemma bias-varianza o problema bias-varianza* è il conflitto nel tentativo di minimizzare simultaneamente queste due fonti di errore che impediscono agli algoritmi di apprendimento supervisionato di generalizzare al di là del loro set di addestramento:

- L'errore di bias è un errore dovuto a presupposti errati nell'algoritmo di apprendimento. Un bias elevato può far sì che un algoritmo non colga le relazioni rilevanti tra le caratteristiche e gli output target (underfitting).
- La varianza è un errore dovuto alla sensibilità a piccole fluttuazioni nel set di addestramento. Un'elevata varianza può derivare da un algoritmo che modella il rumore casuale dei dati di addestramento (overfitting).

• Dartboard metaphor illustrating bias and variance:



L'overfitting e l'underfitting sono due problemi tipici del machine learning in cui il modello raggiunge scarse performance nella classificazione dopo l'addestramento ma per motivi diversi.

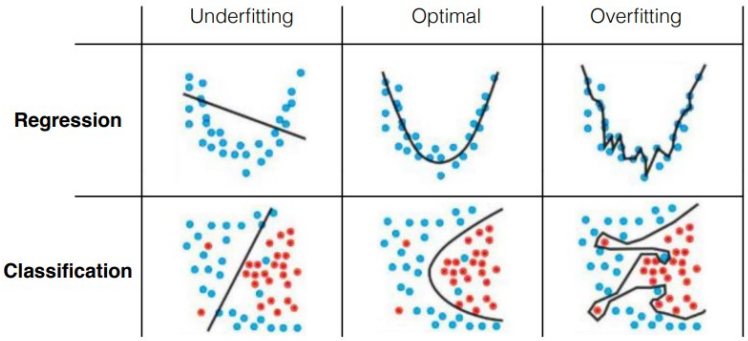
- Nell'**overfitting** ci sono troppi parametri nel modello e un'elevata variabilità della classificazione. Il modello è troppo complesso e sensibile ai dati di training (high variance).
- Nell'**underfitting** ci sono pochi parametri nel modello e un'elevata discrepanza nella classificazione (high bias). Il processo di apprendimento è troppo semplice.

Per capire se un modello ha problemi di underfitting o overfitting, suddivido il dataset in training set e test set. Creo il modello sui dati di training. Poi estrapolo le predizioni e verifico l'accuratezza dei risultati sia sui dati di test (test set) che sui dati di addestramento (training set).

Scritto da Gabriel

- Se l'errore sui dati di training è elevato, c'è sicuramente un problema di underfitting. Il modello ha generalizzato troppo.
- Se l'errore sui dati di training è accettabile ma l'errore sui dati di test è elevato, c'è un problema di overfitting. Il modello non ha generalizzato abbastanza.

Bias-Variance Tradeoff



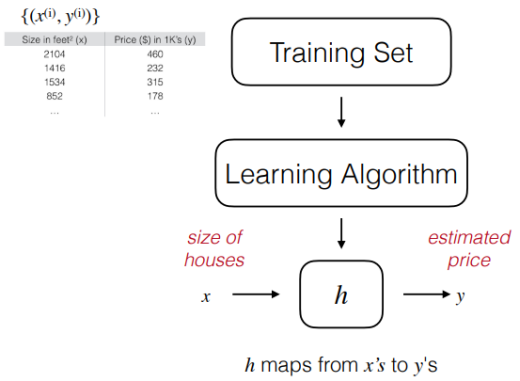
Linear Regression e Gradient Descent

- **Example:** housing prices in Portland (OR)
  - Notation:
    - m = Number of training examples
    - x's = "input" variable / features
    - y's = "output" variable / target variable

Un esempio di Linear Regression, sono i prezzi delle case a Portland, considerando il numero di esempi da valutare, le variabili di input-caratteristiche, le variabili di output-target.

Size in feet² (x)	Price ($) in 1K's (y)
2104	460
1416	232
1534	315
852	178
…	…

Ora: come rappresentiamo *h*? Abbiamo un modello univariato (ad una variabile), rispetto al training set ed all'algoritmo. Quindi, considerato il generico training set, come scegliamo i parametri θ?



How do we represent *h* ?

$h_\theta(x) = \theta_0 + \theta_1 x$

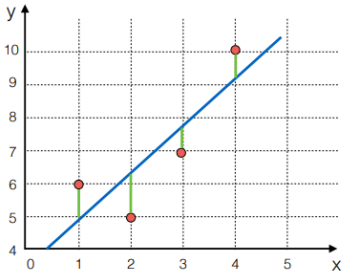
Univariate linear regression model *(i.e. one variable)*

Scegliamo, ad esempio, θ_0 e θ_1 affinché $h_0(x)$ sia vicino ad *y* per i nostri esempi $\{(x^{(i)}, y^{(i)})\}$, con la tecnica dei residui tramite i minimi quadrati.
Da questa intuizione, trovando i residui, la funzione di costo viene data dalla serie armonica dei residui al quadrato (minimi quadrati), dove andiamo a minimizzare sui punti θ. In seguito, si nota che questa coincide con una sorta di valor medio.

m training examples

x	y
1	6
2	5
3	7
4	10

$$h_\theta(x) = \theta_0 + \theta_1 x$$

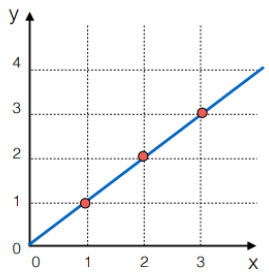


- Residuals: $\{h_\theta(x^{(i)}) - y^{(i)}\}$
- Cost function: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$
- Objective: $\underset{\theta_0,\theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

Come scegliamo, dunque, i parametri θ? Partendo dalla nostra funzione, ad esempio, si considera la funzione di ipotesi *h* e si sostituisce il parametro θ, per esempio ponendo $\theta_1 = 1$.

$h_\theta(x) = \theta_1 x$

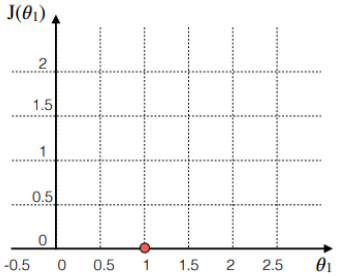
(for fixed θ_1 this is a function of x)



$\theta_1=1$

$J(\theta_1)$

(function of the parameter θ_1)



$$J(\theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m} \sum_{i=1}^{m} (\theta_1 x^{(i)} - y^{(i)})^2 \overset{\theta_1=1}{=} \frac{1}{2m} (0+0+0)^2$$

Similmente, scegliamo una serie di altri valori come segue:

$$J(0) = \frac{1}{2m} [(0-1)^2 + (0-2)^2 + (0-3)^2] = \frac{1}{6} [1+4+9] \approx 2.33$$

$$J(-0.5) = \frac{1}{2m} [(-0.5-1)^2+(-1-2)^2+(-1.5-3)^2] \approx 5.25$$

E otteniamo che, per ciascun valore di θ_1 corrisponde ad una differente ipotesi (quindi, ad esempio, una diversa retta lungo i dati). Infatti, la *cost function*/funzione di costo misura le prestazioni di un modello di apprendimento automatico.
La funzione di costo è il calcolo dell'errore tra i valori previsti e quelli effettivi, rappresentato come un singolo numero reale.
Il quadrato permette di ottenere valori non negativi.

$$J = \frac{1}{n} \sum_{i=1}^{n} (pred_i - y_i)^2$$

Avremo quindi che cerchiamo, in ogni caso con i vari grafici, di avvicinarci a questi dati, ma vorremmo un algoritmo efficiente per trovare automaticamente i valori di θ_0 e θ_1, minimizzando in modo efficiente la funzione di costo *J*.

Si inizia quindi con qualche configurazione dei parametri θ e, continuando a cambiare i parametri di θ_i riduciamo la funzione di costo *J* finché non troviamo un minimo.

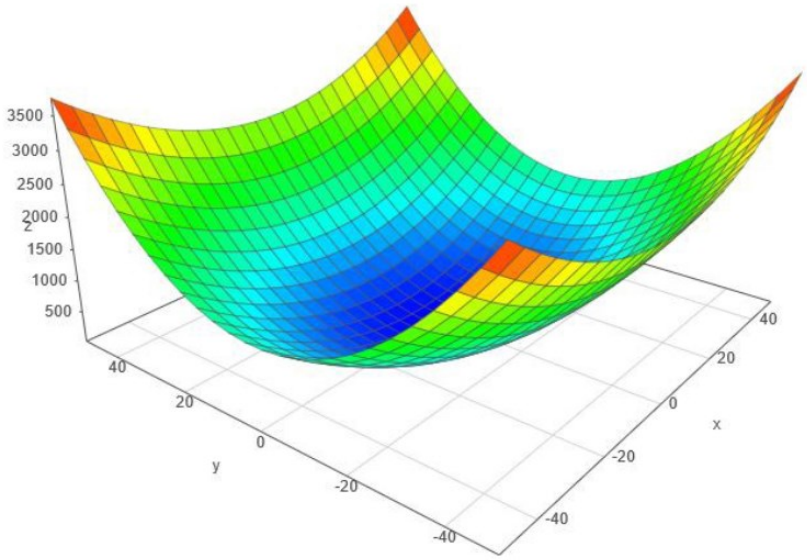
L'algoritmo di discesa del gradiente non funziona per tutte le funzioni. Esistono due requisiti specifici. Una funzione deve essere
- differenziabile
- convessa (quindi, ciò significa che il segmento di retta che collega due punti della funzione giace sulla sua curva o sopra di essa (non la attraversa). In caso contrario, significa che la funzione ha un minimo locale che non è globale)

Quindi, significa che:

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$$

dove λ indica la posizione di un punto sulla linea di sezione e il suo valore deve essere compreso tra 0 (punto a sinistra) e 1 (punto a destra), ad esempio λ=0,5 indica una posizione al centro.
(Possono, comunque, essere usate anche funzione quasi-convesse con i cosiddetti punti di sella, dove la funzione diventa concava/convessa).

Nel caso di una funzione univariata/ad una dimensione, è semplicemente la derivata prima in un punto selezionato. Nel caso di una funzione multivariata, è un vettore di derivate in ogni direzione principale (lungo gli assi variabili). Poiché siamo interessati solo alla pendenza lungo un asse e non ci interessano gli altri, queste derivate sono chiamate derivate parziali. Un gradiente multidimensionale corrisponde a nabla, come:



Il *gradient descent algorithm* calcola iterativamente il punto successivo utilizzando il gradiente nella posizione corrente, lo scala (in base a un tasso di apprendimento) e sottrae il valore ottenuto dalla posizione corrente (fa un passo). Sottrae il valore perché vogliamo minimizzare la funzione (massimizzarla sarebbe un'aggiunta). Questo processo può essere scritto come:

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

Esiste un parametro importante, η, che scala il gradiente e quindi controlla la dimensione del passo. In machine learning, è chiamato tasso di apprendimento e ha una forte influenza sulle prestazioni.

- Più piccolo è il tasso di apprendimento, più a lungo la GD converge, o può raggiungere la massima iterazione prima di raggiungere il punto ottimale.
- Se il tasso di apprendimento fosse troppo alto, l'algoritmo potrebbe non convergere verso il punto ottimale (saltellare) o addirittura divergere completamente.

In sintesi, le fasi del metodo Gradient Descent sono:

1) scegliere un punto di partenza (inizializzazione)
2) calcolare il gradiente in questo punto
3) fare un passo scalato nella direzione opposta al gradiente (obiettivo: minimizzare)
4) ripetere i punti 2 e 3 finché non viene soddisfatto uno dei criteri:
   - numero massimo di iterazioni raggiunto
   - la dimensione del passo è inferiore alla tolleranza (a causa del ridimensionamento o di un gradiente ridotto).

Dato $J(\theta_0, \dots, \theta_n)$ prendiamo un piccolo passo nella direzione del gradiente negativo affinché:

$$\boldsymbol{\theta}_{j+1} = \boldsymbol{\theta}_j - \eta \, \nabla J(\boldsymbol{\theta}_j) \quad \text{where} \quad \boldsymbol{\theta}=\{\theta_0, \dots, \theta_n\}$$

ricalcolando il gradiente ad ogni passo e ripetendo finché non converge, aggiornando simultaneamente θ_0 e θ_1).

- The parameter $\eta > 0$ is known as the *learning rate*
- After each update, the gradient is re-evaluated for the new weight vector $\boldsymbol{\theta}_{j+1}$, and the process repeated
- Note: the cost function is computed on the entire training set in order to evaluate ∇J (i.e. *batch* method)

Se il calcolo della derivata prima di $J(\theta)$ è >= 0, si sta diminuendo nella giusta direzione.
In merito, invece a η:

- se è troppo piccolo, il gradient descent è piccolo
- se è troppo grande, può superare il minimo

Se θ_1 è già il minimo locale, la derivata è pari a 0 e un passo del gradient descent non cambia θ_1.
Approcciando un minimo locale, il gradient descent può convergere ad un minimo locale ad una quota fissa η e prendendo naturalmente passi più piccoli.

Riprendendo il modello della regressione lineare, ripetendo il calcolo sui punti θ_0 e θ_1 aggiornando simultaneamente e cercando progressivamente fino alla convergenza.

Linear Regression Model

- Hypothesis:
  $h_\theta(x) = \theta_0 + \theta_1 x$
- Cost Function:
  $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$
- Objective: $\underset{\theta_0, \theta_1}{\text{minimize}} \; J(\theta_0, \theta_1)$

Gradient Descent

repeat until convergence{

$\theta_j := \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$

(for $j = 0$ and $j = 1$)

}

Si cerca di capire qual è il termine della derivata, partendo da un'inizializzazione casuale dei parametri.
Applicando l'algoritmo del gradient descent, l'algoritmo che trova la linea migliore (best fit) per un dato set di dati di allenamento in un numero minore di iterazioni.
Per una certa combinazione dei fattori θ_0 e θ_1, si ha l'errore minimo.

Partendo infatti dalla loss function, cioè:
La perdita è l'errore nel valore previsto di m e c. Il nostro obiettivo è ridurre al minimo questo errore per ottenere il valore più preciso di m e c.
Per calcolare la perdita utilizzeremo la funzione Errore quadratico medio. Questa funzione prevede tre fasi:

1) Trovare la differenza tra il valore effettivo di y e quello previsto (y = mx + c), per una data x.
2) Elevare al quadrato questa differenza.
3) Trovare la media dei quadrati per ogni valore in X

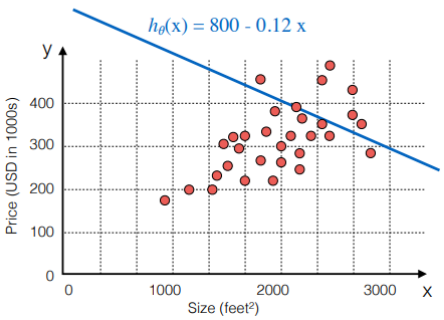
$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)^2 \qquad E = \frac{1}{n}\sum_{i=0}^{n}(y_i - (mx_i + c))^2$$

Proviamo ad applicare il gradient descent a "m" e "c" e ad affrontarla passo dopo passo:

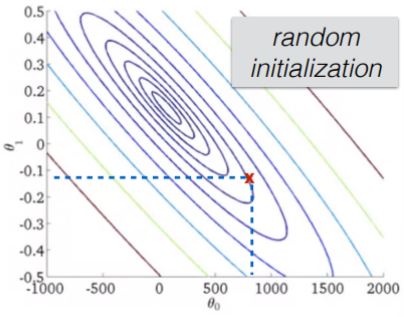
1) Inizialmente lasciamo *m = 0 e c = 0*. L è il nostro tasso di apprendimento. Questo controlla quanto cambia il valore di m a ogni passo. L può essere un valore piccolo come 0,0001 per ottenere una buona precisione.
2) Calcoliamo la derivata parziale della funzione di perdita rispetto a m e inseriamo i valori attuali di *x, y, m e c* per ottenere il valore della derivata D.
   D_m è il valore della derivata parziale rispetto a m. Allo stesso modo troviamo la derivata parziale rispetto a *c*
3) Ora aggiorniamo il valore attuale di "m" e "c"
4) Ripetiamo questo processo finché la nostra funzione di perdita non raggiunge un valore molto piccolo o idealmente 0 (che significa 0 errori o 100% di precisione). I valori di m e c che ci rimangono saranno i valori ottimali.

Questo spiega perché si ottiene questo:

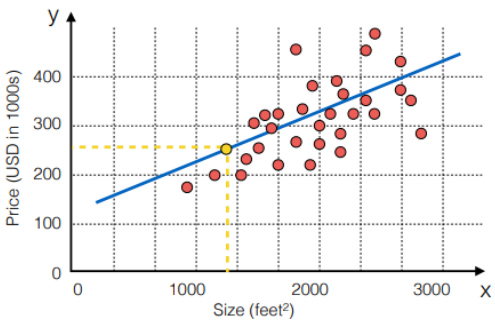
$h_\theta(x) = \theta_0 + \theta_1 x$ (for fixed θ_0,θ_1 this is a function of x)



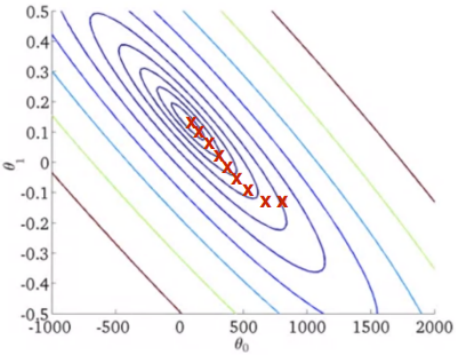
$J(\theta_0,\theta_1)$ (function of the parameters θ_0,θ_1)



$h_\theta(x) = \theta_0 + \theta_1 x$ (for fixed θ_0,θ_1 this is a function of x)



$J(\theta_0,\theta_1)$ (function of the parameters θ_0,θ_1)



Linear Classification, Logistic Regression

Esiste un altro modo per ottimizzare lungo il training set.
Stochastic Gradient Descent: aggiorna i parametri per ogni training case, in base al proprio gradiente. Come suggerisce il nome, muoviamo casualmente gli esempi nel training set, essendo indipendenti e identicamente distribuiti (i.i.d).

La classificazione/classification cerca di capire a quale categoria discreta appartenga uno specifico esempio (ad esempio, un numero, una e-mail se spam o non spam, transazioni online se vere o fraudolente, tumore se maligni o benigni, ecc.).
Le categorie di output sono etichette/labels/classes e, in particolare:

- *binary classification,* due possibili etichette
- *multi-class classification,* varie etichette possibili.

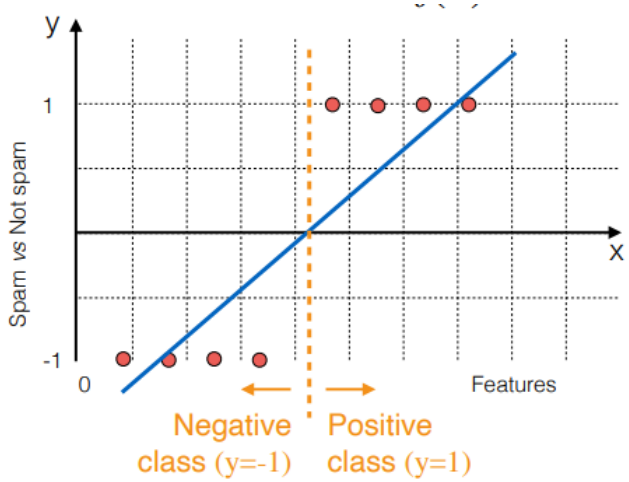
Per esempio, fino ad ora, avevamo un semplice classificatore di soglia e vogliamo capire se si può fare *binary classification*:

$$h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$$

Threshold classifier:

- If $h_\theta(\mathbf{x}) \geq 0.5$, predict $y = 1$
- If $h_\theta(\mathbf{x}) < 0.5$, predict $y = 0$

Cambiamo leggermente notazione sul piano:



Decision rule *(mathematically)*:

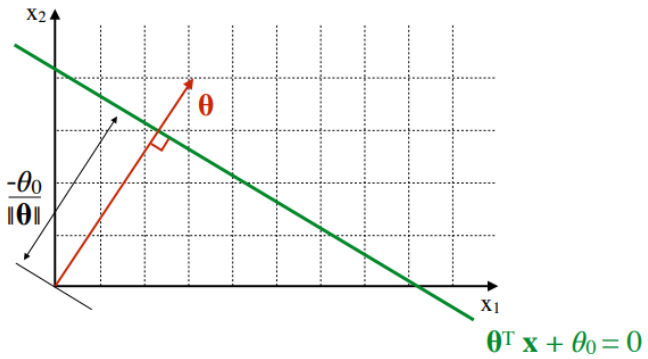
- $y = \text{sign}(h_\theta(\mathbf{x}))$

Questo specifica un *linear classifiier*, con un confine lineare (iperpiano) che separa lo spazio, tra positivo e negativo (in due metà, ad una dimensione è una semplice soglia/threshold).
Applicare la linear regression a compiti di classificazione non è una grande idea, dato che non fitta bene i dati. A due dimensioni è una retta, a tre dimensione è un piano.

Per dimensioni maggiori, assumiamo una retta, uno shift di un certo termine (bias term):

$\boldsymbol{\theta}^T \mathbf{x} = 0$ a line passing through the origin and orthogonal to $\boldsymbol{\theta}$

$\boldsymbol{\theta}^T \mathbf{x} + \theta_0 = 0$ shifts it by θ_0 ← *Note: this is usually referred as to the "bias term"*



Assumendo di usare la notazione:

A bit more about the notation

We are using this trick/assumption:

$$h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}^T \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$$

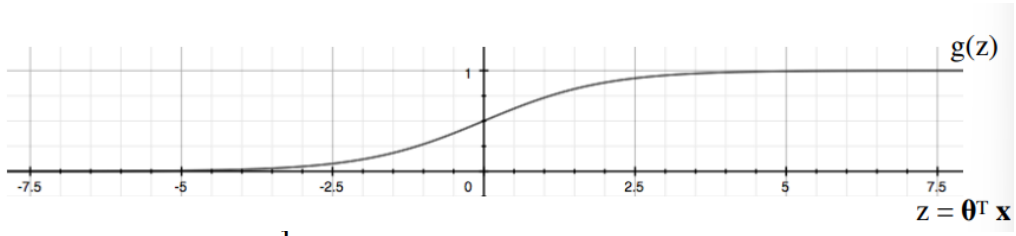
Si stima quindi un buon confine di decisione, decidendo la direzione (θ) e la posizione (θ_0) del limite e abbiamo bisogno di un criterio di selezione dei parametri.
Per esempio, le funzioni di perdita come segue:

- Zero/One: $J_{01}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \{0 \text{ if } h_\theta(x^{(i)})=y^{(i)}, 1 \text{ otherwise}\}$
- Absolute: $J_{abs}(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} |h_\theta(x^{(i)}) - y^{(i)}|$
- Squared: $J_{sqr}(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

- Loss function: $J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} cost(h_\theta(x^{(i)}), y^{(i)})$

Un approccio migliore è usare la logistic regression, che è un algoritmo di classificazione che presenta come proprietà: $0 \le h_\theta(x) \le 1$

La rappresentazione grafica è un sigmoide:



$$h_\theta(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T x}}$$

where $g(z) = \frac{1}{1 + e^{-z}}$ (*Sigmoid* or *Logistic* function)

Si ha che l'interpretazione dell'output di ipotesi è come segue, in pratica una certa probabilità:

- $h_\theta(\mathbf{x})$ = estimated probability that y=1 on input x
- More formally: $h_\theta(\mathbf{x}) = P(y=1 \mid x; \theta)$

Avendo due classi, avremo una proprietà di marginalizzazione, che è un metodo che richiede la somma dei possibili valori di una variabile per determinare il contributo marginale di un'altra.

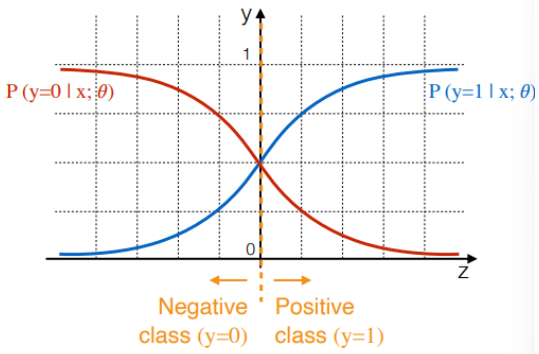
(Esempio utile:
Supponiamo di avere l'attrezzatura e la definizione necessarie per misurare la felicità di una persona e di registrare anche il tempo di una persona in Inghilterra e di un'altra in Scozia. Ora è possibile che le persone in Scozia siano generalmente più felici di quelle in Inghilterra. Il problema è che le persone hanno sempre una nazionalità, quindi non posso eliminarla nella misurazione. Quindi quello che stiamo misurando è P(felicità, paese | tempo), cioè stiamo guardando la felicità e il paese allo stesso tempo.)

La marginalizzazione ci dice di sommare alcune probabilità per ottenere la quantità probabilistica desiderata. Una volta calcolata la risposta (che può essere un singolo valore o una distribuzione), possiamo ottenere le proprietà che vogliamo (inferenza).

Qual è il limite di decisione per la logistic regression? Esso è un limite linearmente convergente.

$$h_\theta(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x})$$

where $g(z) = \frac{1}{1 + e^{-z}}$



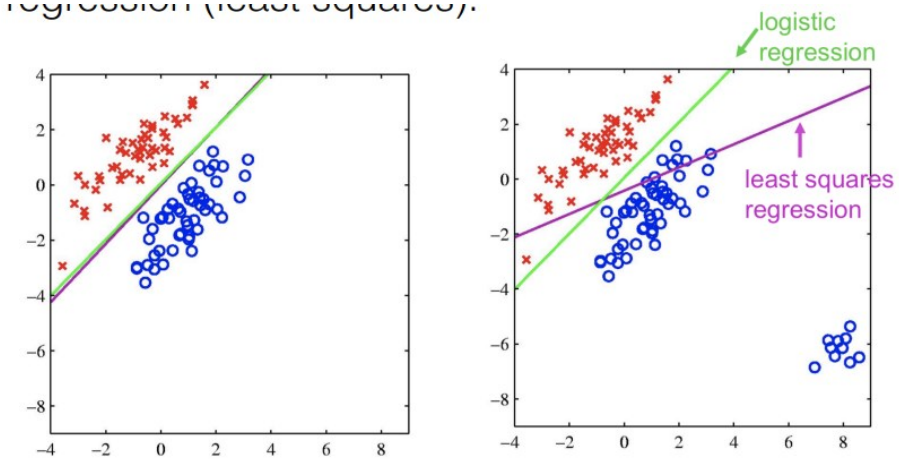
$h_\theta(x) = P(y=1 \mid x; \theta)$

Suppose predict y=1 if $h_\theta(x) \geq 0.5$

predict y=0 if $h_\theta(x) < 0.5$

Logistic Regression has a linear decision boundary

Confrontano la logistic regression con la linear regression:

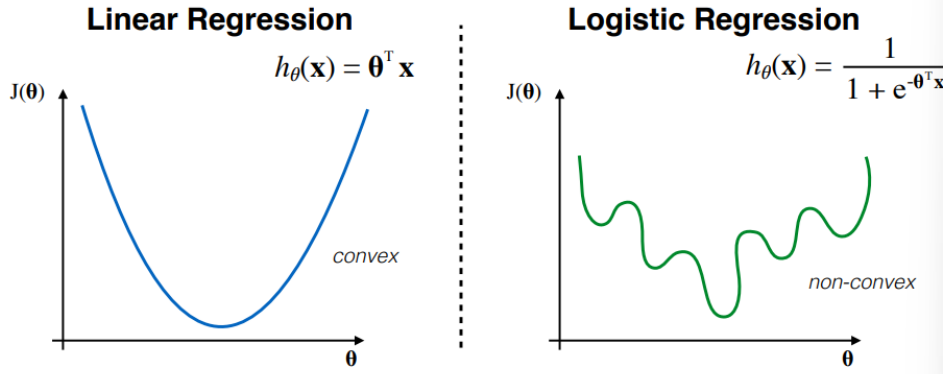


(Se la risposta giusta è 1 e il modello dice 1,5, perde, quindi cambia il confine per evitare di essere "troppo corretto" (si inclina lontano dai valori anomali).

La funzione di perdita, dunque:

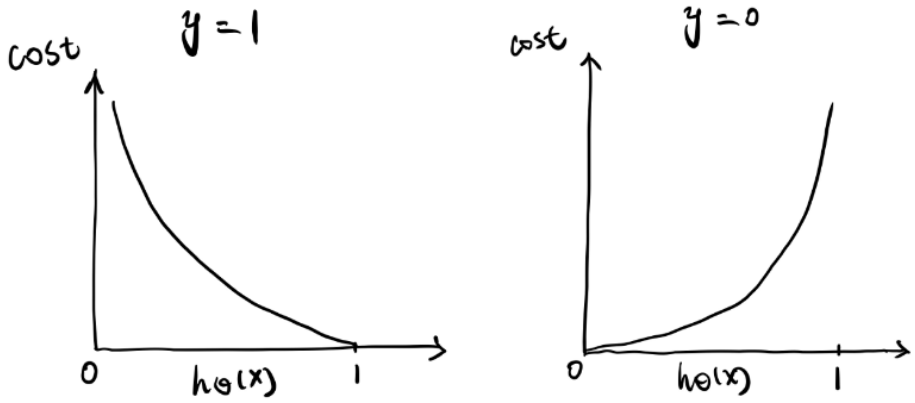
- Loss function: $J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} cost(h_\theta(x^{(i)}), y^{(i)})$

$$where \quad cost(h_\theta(x^{(i)}), y^{(i)}) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$



Intuitivamente, vogliamo assegnare una punizione maggiore quando si prevede 1 mentre l'effettivo è 0 e quando si prevede 0 mentre l'effettivo è 1. La funzione di perdita della regressione logistica fa esattamente questo, ed è chiamata Logistic Loss. Si veda quanto segue. Se y = 1, osservando il grafico qui sotto a sinistra, quando la previsione = 1, il costo = 0, quando la previsione = 0, l'algoritmo di apprendimento viene punito con un costo molto elevato. Allo stesso modo, se y = 0, il grafico a destra mostra che la previsione di 0 non viene punita, ma la previsione di 1 ha un costo elevato.

$$where \quad cost(h_\theta(x^{(i)}), y^{(i)}) = \begin{cases} -\log(h_\theta(x^{(i)})) & \text{if } y^{(i)} = 1 \\ -\log(1 - h_\theta(x^{(i)})) & \text{if } y^{(i)} = 0 \end{cases}$$



Un altro vantaggio di questa funzione di perdita è che, anche se la consideriamo separatamente per y = 1 e y = 0, può essere scritta come un'unica formula, il che comporta una maggiore comodità di calcolo:

$$Cost(h_\theta(x), y) = -ylog(h_\theta(x)) - (1 - y)log(h_\theta(x))$$

Quindi la funzione di costo del modello è la somma di tutti i campioni di dati di addestramento:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$$

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

$$m = number\ of\ samples$$

Con il gradient descent, possiamo apprendere i nostri parametri ricercando il minimo e cercando di applicare l'idea di trovare l'errore medio come segue (definita come *Cross-Entropy Loss Function*, chiamata anche *perdita logaritmica, log loss o logistic loss*. Ogni probabilità di classe prevista viene confrontata con l'effettivo risultato desiderato di classe 0 o 1 e viene calcolato un punteggio/perdita che penalizza la probabilità in base alla distanza dal valore effettivo previsto. La penalizzazione è di tipo logaritmico e dà luogo a un punteggio elevato per le grandi differenze vicine a 1 e a un punteggio ridotto per le piccole differenze che tendono a 0.

La perdita di entropia incrociata viene utilizzata per regolare i pesi del modello durante l'addestramento. L'obiettivo è quello di minimizzare la perdita, vale a dire che quanto più piccola è la perdita tanto migliore è il modello. Un modello perfetto ha una perdita di entropia incrociata pari a 0.)

```
repeat until convergence{
```

$$\theta_j := \theta_j - \eta \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \theta_j - \frac{\eta}{m}\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})\, x_j^{(i)}$$

(simultaneously update all θ_j)

```
}
```

La regola di aggiornamento (gradient descent) è esattamente la stessa sia per la regressione lineare che per quella logistica. Perché?
Diamo un'occhiata alla derivata della funzione di costo per la logistic regression.
Vogliamo trovare la derivata parziale della funzione di costo **J**:

$$\text{Cost function} \quad J(\boldsymbol{\theta}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \cdot \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - h_\theta(x^{(i)}))\right]$$

$$\text{where} \quad h_\theta(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) \quad \text{and} \quad g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

- Let's start by computing the derivative of $\sigma(z)$

$$\frac{d\,\sigma(z)}{dz} = \frac{d}{dz}\frac{f(z) = 1}{g(z) = 1 + e^{-z}}$$

Quotient rule

$$\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{f'g - f\,g'}{g^2}$$

A.Y. 2021/22: Introduction to Machine Learning

$$\frac{d\,\sigma(z)}{dz} = \frac{0 \cdot (1 + e^{-z}) - (1) \cdot (e^{-z} \cdot (-1))}{(1 + e^{-z})^2} = \frac{(e^{-z})}{(1 + e^{-z})^2} = \frac{1-1+(e^{-z})}{(1 + e^{-z})^2} =$$

$$= \frac{1+(e^{-z})}{(1 + e^{-z})^2} - \frac{1}{(1 + e^{-z})^2} = \frac{1}{(1 + e^{-z})} \cdot \left(1 - \frac{1}{(1 + e^{-z})}\right) = \sigma(z) \cdot (1 - \sigma(z))$$

$$\text{Cost function} \quad J(\boldsymbol{\theta}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \cdot \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - h_\theta(x^{(i)}))\right]$$

$$\text{where} \quad h_\theta(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) \quad \text{and} \quad g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

- Writing now in terms of partial derivatives:

$$\frac{\partial}{\partial\theta_j} J(\boldsymbol{\theta}) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \cdot \frac{1}{h_\theta(x^{(i)})} \cdot \frac{\partial}{\partial\theta_j} h_\theta(x^{(i)}) + \right.$$

$$\left. + (1 - y^{(i)}) \cdot \frac{1}{(1 - h_\theta(x^{(i)}))} \cdot \frac{\partial}{\partial\theta_j} (1 - h_\theta(x^{(i)}))\right]$$

Mettendo i risultati precedenti e usando il pattern di derivazione delle sigmoidi', si ha poi una semplificazione dei prodotti, ottenendo che la funzione di costo, alla fine, sarà:

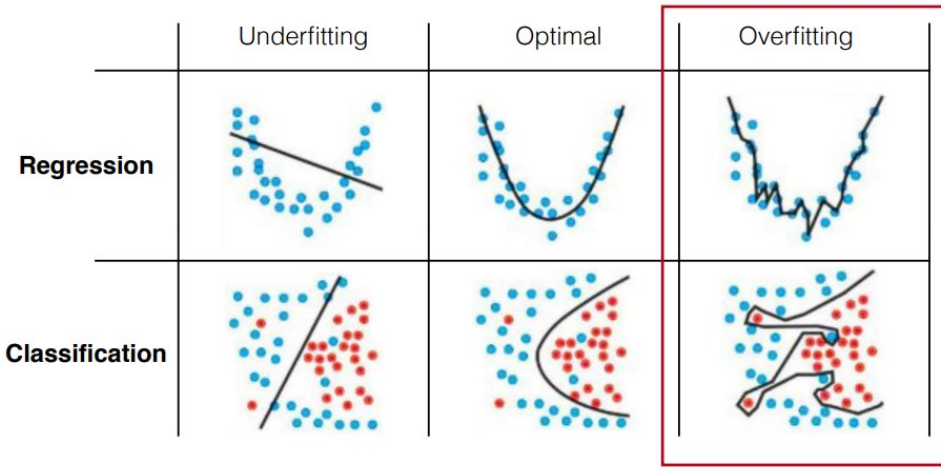
$$\frac{\partial}{\partial\theta_j} J(\boldsymbol{\theta}) = -\frac{1}{m}\left[\sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) \cdot x_j^{(i)}\right]$$

Regularization, Model Selection and Evaluation

Quando si parla di previsione dei modelli, è importante comprendere gli errori di previsione (bias e varianza). Esiste un compromesso tra la capacità di un modello di minimizzare il bias e la varianza. Una corretta comprensione di questi errori ci aiuterebbe non solo a costruire modelli accurati, ma anche a evitare gli errori di overfitting e underfitting.
Iniziamo quindi con le nozioni di base e vediamo come fanno la differenza nei nostri modelli di apprendimento automatico (concetti già spiegati, ma utili a concettualizzare qui).

- Il bias è la differenza tra la previsione media del nostro modello e il valore corretto che stiamo cercando di prevedere. Un modello con un bias elevato presta poca attenzione ai dati di addestramento e semplifica eccessivamente il modello. Questo porta sempre a un errore elevato sui dati di addestramento e di test.
- La varianza è la variabilità della previsione del modello per un dato punto di dati o un valore che ci dice la diffusione dei nostri dati. Un modello con una varianza elevata presta molta attenzione ai dati di addestramento e non generalizza sui dati che non ha mai visto prima. Di conseguenza, tali modelli ottengono ottime prestazioni sui dati di addestramento, ma presentano alti tassi di errore sui dati di test.

Consideriamo la variabile che stiamo cercando di prevedere come Y e le altre covariate come X. Assumiamo che esista una relazione tra le due variabili tale che $Y = f(X) + e$
dove *e* è il termine di errore e si distribuisce normalmente con una media pari a 0.
Creeremo un modello f^(X) di f(X) utilizzando la regressione lineare o qualsiasi altra tecnica di modellazione.



Nel diagramma precedente, al centro del bersaglio si trova un modello che predice perfettamente i valori corretti. Man mano che ci allontaniamo dal bersaglio, le nostre previsioni diventano sempre peggiori. Possiamo ripetere il processo di costruzione del modello per ottenere risultati diversi sul bersaglio.

Nel supervised learning, l'underfitting si verifica quando un modello non è in grado di catturare il modello sottostante dei dati. Questi modelli hanno solitamente un bias elevato e una bassa varianza. Succede quando abbiamo una quantità di dati molto bassa per costruire un modello accurato o quando cerchiamo di costruire un modello lineare con dati non lineari. Inoltre, questo tipo di modelli è molto semplice per catturare i modelli complessi nei dati, come la regressione lineare e logistica.

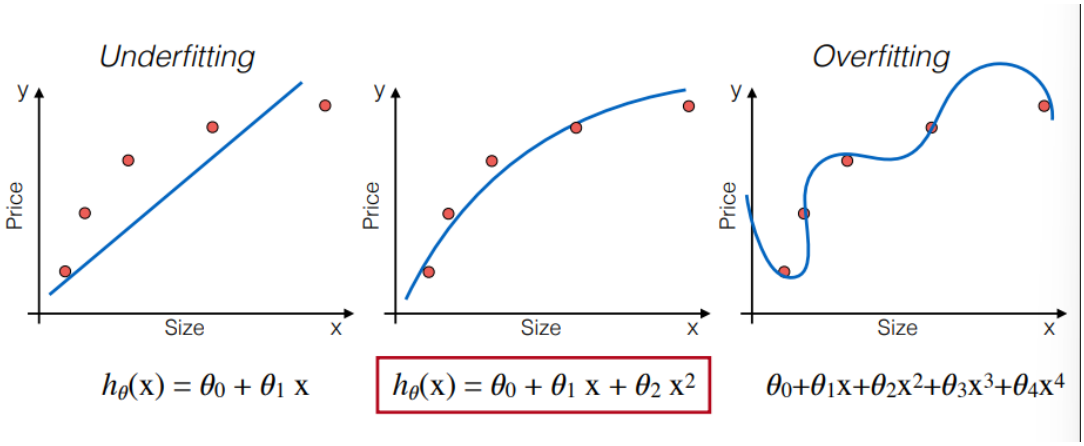
Nell'apprendimento supervisionato, l'overfitting si verifica quando il nostro modello cattura il rumore insieme al modello sottostante nei dati. Ciò accade quando addestriamo il nostro modello su un insieme di dati rumorosi. Questi modelli hanno un basso bias e un'alta varianza. Questi modelli sono molto complessi, come gli alberi decisionali, che sono inclini all'overfitting.

Scritto da Gabriel

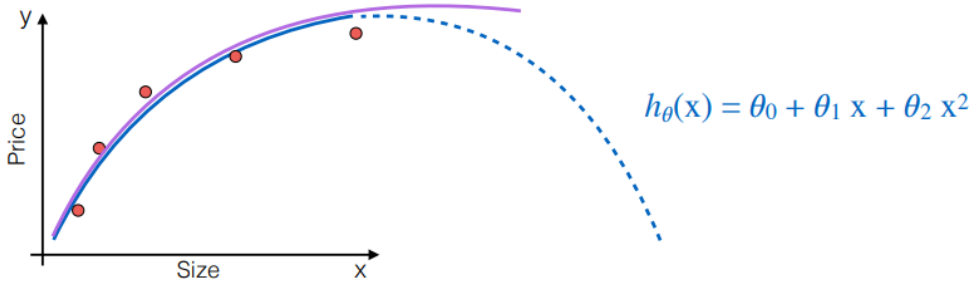
Perché c'è un tradeoff tra bias e varianza?

Se il nostro modello è troppo semplice e ha pochi parametri, può avere un bias elevato e una varianza bassa. D'altra parte, se il nostro modello ha un gran numero di parametri, avrà un'alta varianza e un basso bias. Dobbiamo quindi trovare il giusto/buono equilibrio, senza overfitting e underfitting sui dati.
Questo compromesso pesa sulla complessità, rispetto al bias o alla varianza, variando la complessità.

Prendiamo una generalizzazione, con:



Per prevenire l'underfitting, dobbiamo aumentare la complessità del modello e introdurre l'equazione Y come scritto qui. Questo è ancora considerato un modello lineare, poiché i coefficienti/pesi associati alle caratteristiche sono ancora lineari. x^2 è solo una caratteristica. Tuttavia, la curva che stiamo adattando è di natura quadratica.
Come si vede, è decisamente migliore:



Con l'overfitting, invece, abbiamo troppe caratteristiche e $h_0(x)$ potrebbe fallire nel generare nuovi campionamenti. In generale, il modello cattura anche il rumore dei dati. Questo è un esempio di over-fitting. Anche se questo modello passa attraverso la maggior parte dei dati, non riuscirà a generalizzare sui dati non visti.
Per evitare l'over-fitting, si possono aggiungere altri campioni di addestramento, in modo che l'algoritmo non apprenda il rumore del sistema e diventi più generalizzato.
In generale, comunque, si hanno due opzioni:

- Ridurre il numero di caratteristiche attraverso la selezione del modello/model selection
- Regolarizzazione (mantenere tutte le caratteristiche, ma ridurre l'influenza dei parametri θ_j
  Questo funziona bene se abbiamo molte caratteristiche, ciascuna che contribuisce un po' a prevedere *y*.

Cerchiamo, dunque, di capire la regolarizzazione/regularization.
Si tratta di una forma di regressione che vincola/regolarizza o restringe le stime dei coefficienti verso lo zero. In altre parole, questa tecnica scoraggia l'apprendimento di un modello più complesso o flessibile, per evitare il rischio di overfitting.
Una semplice relazione per la regressione lineare si presenta come segue. Qui Y rappresenta la relazione appresa e β rappresenta le stime dei coefficienti per le diverse variabili o predittori (X).

La procedura di adattamento prevede una funzione di perdita, nota come somma dei quadrati residui o RSS. I coefficienti vengono scelti in modo da minimizzare questa funzione di perdita.

$$\text{RSS} = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 .$$

In questo modo, i coefficienti verranno regolati in base ai dati di addestramento. Se nei dati di addestramento c'è del rumore, i coefficienti stimati non si generalizzeranno bene ai dati futuri. È qui che entra in gioco la regolarizzazione, che riduce o regolarizza le stime apprese verso lo zero.

A livello pratico, si considera di rendere θ_3 e θ_4 molto piccoli:

$$\min_{\theta} J(\theta) = \min_{\theta} \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + 1000\,\theta_3^2 + 1000\,\theta_4^2$$

E in generale, si introduce un termine che rimpicciolisce ogni parametro:

$$\textit{Objective} \quad \min_{\theta} J(\theta) = \min_{\theta} \frac{1}{2m} \left[\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2 \right]$$

This extra regularization term will shrink every single parameter

L'immagine sopra mostra la regressione, in cui la RSS viene modificata aggiungendo la quantità di contrazione. Ora, i coefficienti vengono stimati minimizzando questa funzione. Qui, λ è il parametro di regolazione che decide quanto vogliamo penalizzare la flessibilità del nostro modello. L'aumento della flessibilità di un modello è rappresentato dall'aumento dei suoi coefficienti e, se vogliamo minimizzare la funzione di cui sopra, questi coefficienti devono essere piccoli. È così che la tecnica di regressione impedisce che i coefficienti aumentino troppo.

Quando λ = 0, il termine di penalizzazione non ha alcun effetto e le stime prodotte dalla regressione saranno uguali ai minimi quadrati. Tuttavia, con l'aumentare di $\lambda \to \infty$, l'impatto della penalizzazione cresce e le stime del coefficiente della regressione ridge si avvicinano a zero. Come si può notare, la scelta di un buon valore di λ è fondamentale. La convalida incrociata è utile a questo scopo. Le stime dei coefficienti prodotte da questo metodo sono note anche come norma L^2.

I coefficienti prodotti dal metodo dei minimi quadrati standard sono equivarianti di scala, vale a dire che se moltiplichiamo ogni input per c, i coefficienti corrispondenti vengono scalati di un fattore 1/c. Pertanto, indipendentemente dalla scala del predittore, la moltiplicazione del predittore e del coefficiente (Xjβj) rimane la stessa. Tuttavia, questo non è il caso della regressione e quindi è necessario standardizzare i predittori o portare i predittori alla stessa scala prima di eseguire la regressione.

Ritrovando i parametri con il gradient descent, si ricava che:

repeat until convergence{

$$\theta_0 := \theta_0 - \eta \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})\, x_0^{(i)}$$

(simultaneously update all θ_j)

$$\theta_j := \theta_j - \eta \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})\, x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \quad (j = 1, \ldots, n)$$

}

Here there's something pretty interesting

---

e soprattutto:

repeat until convergence{

$$\theta_0 := \theta_0 - \eta \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})\, x_0^{(i)}$$

(simultaneously update all θ_j)

$$\theta_j := \theta_j (1 - \eta \frac{\lambda}{m}) - \eta \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})\, x_j^{(i)} \quad (j = 1, \ldots, n)$$

}

This is going to be <1 (≈0.99)

This is exactly the same term we had before introducing regularization

Possiamo regolarizzare la logistic regression in modo simile alla linear regression:

- Recall that our cost function was:

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^{m} y^{(i)} \cdot \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - h_\theta(x^{(i)})) \right]$$

- We can regularize this equation by adding a term:

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^{m} y^{(i)} \cdot \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

Si ritorna allo stesso algoritmo usato per la linear regression:

$$\theta_0 := \theta_0 - \eta \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})\, x_0^{(i)}$$

(simultaneously update all θ_j)

$$\theta_j := \theta_j - \eta \left[\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})\, x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \quad (j = 1, \ldots, n)$$

}

This is exactly the same algorithm we use for linear regression!

Introduciamo ora la empirical risk minimization, definisce una famiglia di algoritmi di apprendimento e viene utilizzato per dare dei limiti teorici sulle loro prestazioni
Non sappiamo quanto bene un algoritmo di apprendimento funzionerà in pratica ("rischio vero") perché la vera distribuzione dei dati. Ma possiamo misurare le sue prestazioni sul nostro set di dati di di addestramento ("rischio empirico").

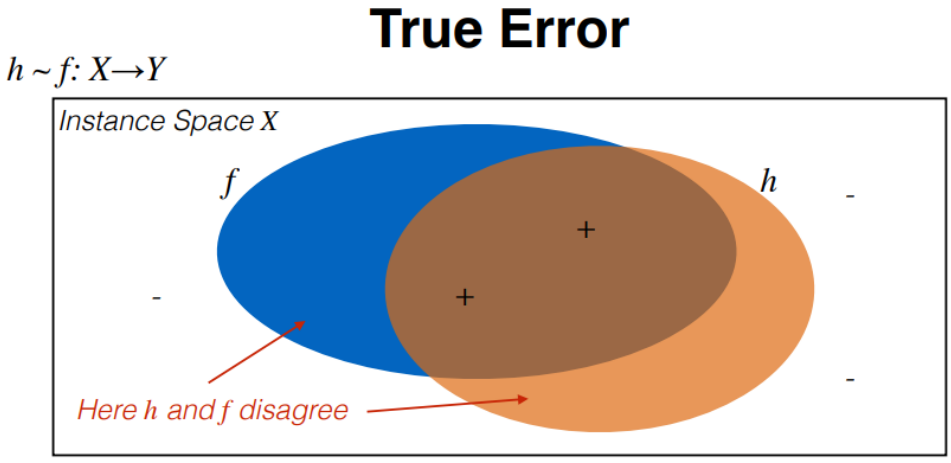
Partiamo da un semplice problema di classificazione dell'apprendimento supervisionato. Supponiamo di voler classificare le e-mail di spam. Ogni e-mail ha un'etichetta 0 o 1, spam o non spam. Denota lo spazio del dominio con X e lo spazio delle etichette con Y. Abbiamo anche bisogno di una funzione per mappare lo spazio degli insiemi del dominio nello spazio degli insiemi delle etichette, $f: X -> Y$, questa è solo una definizione formale di un compito di apprendimento.

Ora che abbiamo la definizione formale del problema, abbiamo bisogno di un modello che faccia le nostre previsioni: spam o non spam. Per coincidenza, il sinonimo di modello è l'ipotesi h, il che può creare un po' di confusione. L'ipotesi, in questo caso, non è altro che una funzione che prende in input il nostro dominio X e produce un'etichetta 0 o 1, cioè una funzione $h: X -> Y$.

Alla fine, vogliamo trovare l'ipotesi che minimizza il nostro errore, giusto? Per questo motivo si parla di minimizzazione empirica del rischio. Il termine empirico implica che minimizziamo il nostro errore sulla base di un insieme di campioni S dall'insieme di domini X. Guardando la cosa da una prospettiva probabilistica, diciamo che campioniamo S dall'insieme di domini X, con D che è la distribuzione su X. Quindi, quando campioniamo dal dominio, esprimiamo la probabilità che un sottoinsieme del dominio sia campionato dal dominio X con D(S).
Nell'equazione che segue, possiamo definire l'errore vero, che è basato sull'intero dominio X:

$$L_{\mathcal{D},f}(h) \overset{\text{def}}{=} \mathbb{P}_{x\sim\mathcal{D}}[h(x) \neq f(x)] \overset{\text{def}}{=} \mathcal{D}(\{x : h(x) \neq f(x)\}).$$

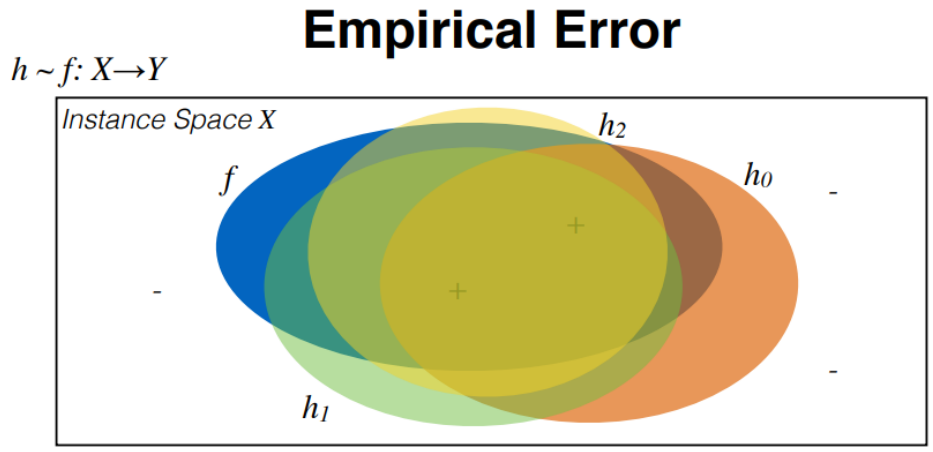
Graficamente:



Errore vero/True error dell'ipotesi *h* rispetto all'obiettivo *f* e alla distribuzione *D* (per osservare un'istanza di input x appartenente ad X) è la probabilità che *h* classifichi erroneamente un'istanza estratta a caso secondo *D*.

$$error_D(h) \equiv \underset{x \in D}{\text{Prob}}\{ f(x) \neq h(x) \}$$

Poiché abbiamo accesso solo a S, un sottoinsieme del dominio di input, impariamo sulla base di quel campione di esempi di addestramento. Non abbiamo accesso all'errore vero, ma all'errore empirico/empirical error dell'ipotesi *h* rispetto a un insieme di addestramento *T* è il numero di esempi che *h* classificherà in modo errato:

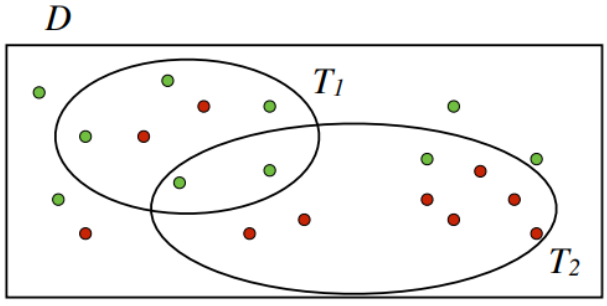


$$error_T(h) \equiv \#\{ (x, f(x)) \in T \mid f(x) \neq h(x) \}$$

L'errore empirico è talvolta chiamato anche *errore di generalizzazione*. Il motivo è che in realtà, nella maggior parte dei problemi, non abbiamo accesso all'intero dominio X di input, ma solo al nostro sottoinsieme S di addestramento. Vogliamo generalizzare sulla base di S, detto anche apprendimento induttivo. Questo errore è chiamato anche rischio, da cui il termine rischio nella minimizzazione empirica del rischio.

Riprendendo le slide, diamo un esempio come segue. Non sappiamo mai generalmente l'errore vero, vediamo solo l'errore empirico. Quanto bene l'errore empirico stima l'errore vero?

- A simple example:
  - Correctly classified: $f(x) = h(x)$
  - Misclassified: $f(x) \neq h(x)$



$error_D(h) = 0.5$

$error_{T_1}(h) = 0.29$

$error_{T_2}(h) = 0.7$

Guardando all'esempio citato bias e varianza:

- Per il bias, l'errore $error_{Ti}(h)$ è ottimisticamente di parte/biased
- Per la varianza, anche senza bias l'errore $error_{Ti}(h)$ potrebbe comunque variare da $error_D(h)$.

Ora possiamo parlare del problema dell'overfitting. In altre parole, poiché disponiamo solo di un sottocampione dei dati, può accadere che minimizziamo l'errore empirico ma in realtà aumentiamo l'errore reale. Questo risultato può essere osservato in un semplice problema di adattamento della curva.

Overfitting: given $h \in H$, h overfits T if $\exists h' \in H$ such that $error_T(h) < error_T(h')$ but $error_D(h) > error_D(h')$

Hold-out: manteniamo un sottoinsieme di *v* campioni dall'insieme di formazione (l'insieme di validazione) per valutare il nostro modello.

- Un classificatore/regressore viene addestrato su m-v campioni.
- I parametri vengono ottimizzati sugli insiemi di addestramento e validazione: poi si valutano le prestazioni sul set di test
- La dimensione (cardinalità) degli insiemi di addestramento+validazione dovrebbe essere maggiore dell'insieme di test, ad esempio 70%,15%,15%

L'hold-out è quando si divide il set di dati in un set di "addestramento" e in un set di "test". L'insieme di addestramento è quello su cui viene addestrato il modello, mentre l'insieme di test viene utilizzato per verificare le prestazioni del modello su dati non visti. Una suddivisione comune quando si utilizza il metodo hold-out consiste nell'utilizzare l'80% dei dati per l'addestramento e il restante 20% dei dati per il test.

K-fold cross validation: *k* classificatori/regressori $h_1 \ldots h_k$ sono addestrati a partizionare il training set *T* in *k* set disgiunti (pieghe) $V_1, \ldots V_k$ e poi applicando in modo iterativo l'approccio di hold-out. Le prestazioni complessive vengono misurate calcolando la media dei risultati
ottenuti sulle diverse pieghe.

La convalida incrociata è una procedura di ricampionamento utilizzata per valutare i modelli di apprendimento automatico su un campione di dati limitato.
La procedura ha un unico parametro, chiamato k, che si riferisce al numero di gruppi in cui deve essere suddiviso un determinato campione di dati. Per questo motivo, la procedura viene spesso chiamata convalida incrociata k-fold. Quando si sceglie un valore specifico per k, questo può essere usato al posto di k nel riferimento al modello, come ad esempio k=10 che diventa una convalida incrociata a 10 volte.

La convalida incrociata è utilizzata principalmente nell'apprendimento automatico applicato per stimare l'abilità di un modello di apprendimento automatico su dati non visti. In altre parole, si utilizza un campione limitato per stimare le prestazioni generali del modello quando viene utilizzato per fare previsioni su dati non utilizzati durante l'addestramento del modello.
Si tratta di un metodo molto diffuso perché è semplice da capire e perché in genere produce una stima meno distorta o meno ottimistica dell'abilità del modello rispetto ad altri metodi, come la semplice divisione treno/test.

La procedura generale è la seguente:

1) Mescolare il dataset in modo casuale.
2) Dividere il set di dati in k gruppi
3) Per ogni gruppo unico:
    1) prendere il gruppo come set di dati di attesa o di test
    2) Prendere i gruppi rimanenti come set di dati di addestramento
    3) Adattare un modello sul set di addestramento e valutarlo sul set di test.
    4) Conservare il punteggio della valutazione e scartare il modello.
    5) Riassumere l'abilità del modello utilizzando il campione di punteggi di valutazione del modello.

È importante notare che ogni osservazione del campione di dati viene assegnata a un gruppo individuale e rimane in quel gruppo per tutta la durata della procedura. Ciò significa che ogni campione ha l'opportunità di essere usato nel set di hold out 1 volta e di essere usato per addestrare il modello k-1 volte.

Leave-one-out: È un caso speciale di k-fold cross validation in cui k è uguale alla cardinalità dell'insieme di addestramento.
La convalida incrociata leave-one-out utilizza il seguente approccio per valutare un modello:
1. Dividere un set di dati in un set di allenamento e in un set di test, utilizzando tutte le osservazioni tranne una come parte del set di allenamento.
Si noti che lasciamo solo un'osservazione "out" dall'insieme di addestramento. È da qui che il metodo prende il nome di convalida incrociata "leave-one-out".

Scritto da Gabriel

2. Costruire il modello utilizzando solo i dati dell'insieme di addestramento.
3. Utilizzate il modello per prevedere il valore della risposta dell'unica osservazione esclusa dal modello e si calcola l'errore ai minimi quadrati.
4. Ripetere il processo n volte.

Infine, ripetiamo questo processo n volte (dove n è il numero totale di osservazioni nel set di dati), escludendo ogni volta un'osservazione diversa dal set di allenamento.

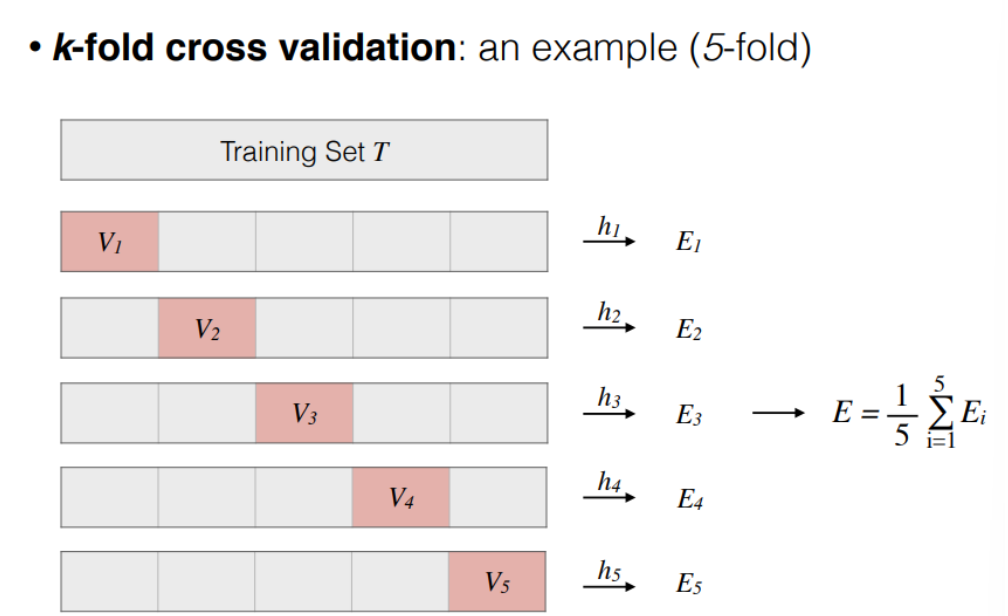
La convalida incrociata leave-one-out offre i seguenti vantaggi:

- Fornisce una misura molto meno distorta dell'MSE del test rispetto all'uso di un singolo set di test, perché adattiamo ripetutamente un modello a un set di dati che contiene n-1 osservazioni.
- Tende a non sovrastimare l'MSE del test rispetto all'utilizzo di un singolo set di test.

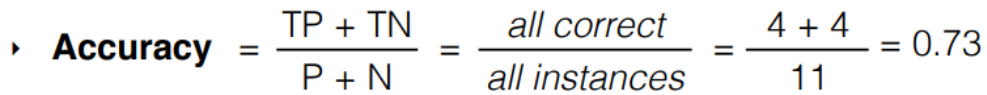
Tuttavia, la convalida incrociata leave-one-out presenta i seguenti svantaggi:

- Può essere un processo che richiede molto tempo quando n è grande.
- Può anche essere dispendioso se un modello è particolarmente complesso e richiede molto tempo per essere adattato a un set di dati.
- Può essere computazionalmente costoso.

Un esempio grafico di k-fold cross validation:



E la valutazione delle performance è come segue:



Si usa una struttura che sarebbe una *confusion matrix*.
Una matrice di confusione è un riepilogo dei risultati delle previsioni su un problema di classificazione. Il numero di previsioni corrette e scorrette viene riassunto con valori di conteggio e suddiviso per ogni classe. Questa è la chiave della matrice di confusione.
La matrice di confusione mostra i modi in cui il modello di classificazione di classificazione è confuso quando fa delle previsioni. Fornisce una visione non solo degli errori commessi dal classificatore, ma soprattutto dei tipi di errori commessi.
È questa suddivisione che supera il limite dell'utilizzo della sola accuratezza della classificazione.

- **Confusion Matrix**:

		Actual Class (groundtruth)	
		dog	no-dog
Predicted	dog	**TP:** True Positive	**FP:** False Positive *Type I error*
	no-dog	**FN:** False Negative *Type II error*	**TN:** True Negative
		P	**N**

La tabella precedente presenta i seguenti casi:

- Vero negativo: Il modello ha dato una previsione negativa e anche il valore reale o effettivo era negativo.
- Vero positivo: Il modello ha previsto sì e anche il valore reale era vero.
- Falso negativo: Il modello ha previsto un no, ma il valore reale era un sì
- Falso positivo: Il modello ha previsto sì, ma il valore effettivo era no.

Evaluation, Learning Curves & Babysitting

Come valutare le prestazioni? Esamineremo diverse metriche per i compiti di classificazione.
Si prende un esempio semplice; classificare se l'animale sia "cane" oppure "non sia cane".
In questo caso:

- i TP (True Positive) sono i cani
- i FP (False Positive) sono quelli che hanno affinità di specie con i cani e somigliano a cani, in questo caso i lupi
- i TN (True Negative), quelli che non assomigliano decisamente a cani essendo tutt'altro animali
- i FN (False Negative), quelli che sembrano tutt'altro e in realtà sono proprio cani.

La valutazione è come segue:

$$\textbf{Accuracy} = \frac{TP + TN}{P + N} = \frac{all\ correct}{all\ instances} = \frac{4 + 4}{11} = 0.73$$

Con il falso negativo, si avrebbe un *errore di tipo 2/type 2 error.*

Un errore di tipo I (falso-positivo) si verifica se lo sperimentatore rifiuta un'ipotesi nulla che è effettivamente vera nella popolazione; un errore di tipo II (falso-negativo) si verifica se lo sperimentatore non rifiuta un'ipotesi nulla che è effettivamente falsa nella popolazione.
La valutazione avviene secondo la differenza (Precisione/Precision vs. Richiamo/Recall):

- Precisione: la frazione di istanze recuperate che sono rilevanti

$$\text{Prec} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{\textit{all predicted}}$$

- Richiamo: la frazione di istanze rilevanti che vengono recuperate

$$\text{Rec} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\textit{all groundtruth instances}}$$

Intendendo con TPR la True Positive Rate, si ha anche questa uguaglianza: $\text{sensitivity} = \text{TPR} = \text{Rec}$

- F1-score: Media armonica tra Precisione e Richiamo

$$\text{F1} = 2 \cdot \frac{\text{Prec} \cdot \text{Rec}}{\text{Prec} + \text{Rec}}$$

Similmente, si ha anche la specificità (FPR/False Positive Rate):

$$\text{specificity} = \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} = 1 - \frac{\text{TN}}{\text{FP} + \text{TN}}$$

Per quale motivazione abbiamo bisogno di metriche differenti?
Sull'esempio di prima avremmo:

$$\text{Prec} = \frac{\text{TP}}{\text{TP} + \text{FP}} = 4/6 = 0.67 \qquad \text{Rec} = \frac{\text{TP}}{\text{TP} + \text{FN}} = 4/5 = 0.8$$

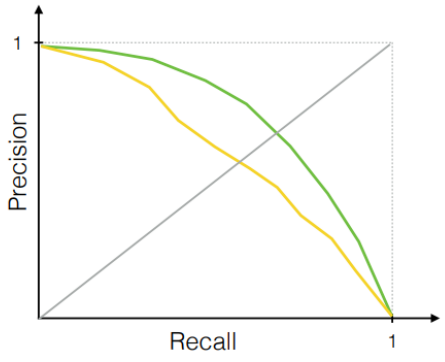
Se avessimo un altro modello sugli stessi dati, si avrebbero:

$$\text{Prec} = \frac{\text{TP}}{\text{TP} + \text{FP}} = 2/2 = 1 \qquad \text{Rec} = \frac{\text{TP}}{\text{TP} + \text{FN}} = 2/5 = 0.4$$

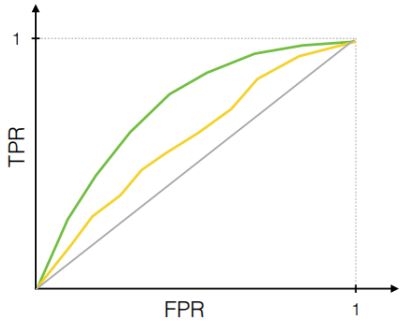
Serve un "giusto" compromesso tra Precisione e Richiamo. Dipende dall'applicazione/compito che si sta affrontando. Per esempio, in un'applicazione di videosorveglianza si potrebbe preferire un elevato richiamo (per non perdere nessun evento anomalo), ma è necessario un equilibrio! Se ci sono troppi allarmi, l'operatore umano non presterà attenzione agli allarmi.

Purtroppo, non è possibile avere sia la precisione che il richiamo elevati. Se si aumenta la precisione, si riduce il richiamo e viceversa. Si tratta del cosiddetto *compromesso precisione/richiamo – precision/recall tradeoff*.
La valutazione delle metriche procede con:

- *Precision-Recall Curves*:
  Le curve di precisione-richiamo riassumono il compromesso tra il tasso di veri positivi e il valore predittivo positivo per un modello predittivo che utilizza diverse soglie di probabilità. Le curve PR sono si ottengono calcolando precisione e richiamo in funzione degli iperparametri (come una soglia).



- *ROC curves* (simili alle curve PR ma con riferimento a FPR e TPR)
  Le curve ROC riassumono il compromesso tra il tasso di veri positivi e il tasso di falsi positivi per un modello predittivo utilizzando diverse soglie di probabilità. Sono appropriate quando le osservazioni sono bilanciate tra ciascuna classe, mentre le curve precision-recall sono appropriate per i dataset sbilanciati. Abbiamo anche l'Area Under Curve (AUC): un singolo punteggio calcolato dal ROC.
  Si noti che ROC significa "Receiver Operating Characteristic curve".



La coppia Precision-Recall rappresenta l'insieme di metriche più comuni per i compiti di di recupero delle informazioni. L'insieme rilevante (tutte le istanze) potrebbe essere enorme. Solitamente, come output, si ottiene un elenco classificato.
Spesso si calcolano Prec@K e Rec@K sui primi K risultati (top-K) → Caso Google.

La valutazione considera quindi la *Average Precision/AP*, una misura che combina la precisione ed il richiamo per risultati di ricerca classificati. AP considera anche la posizione nella classificazione e solitamente riportiamo AP (*mAP*) per un insieme di query.

$$AP = \frac{\sum_{k=1}^{n} P(k) \cdot rel(k)}{all\ groundtruth\ instances}$$

$$mAP = \frac{1}{Q} \cdot \sum_{q=1}^{Q} AP(q)$$

- P(k) è la precisione al cut-off/soglia/treshold k nell'elenco (quindi, il cutoff è una classificazione binaria che la predizione sia vera)
- rel(k) è una funzione indicatrice uguale ad 1 se l'oggetto di rango *k* è rilevante, 0 altrimenti
- Q è il numero di query

Un esempio grafico:

Relevant	Precision	Recall
1	1	0.2
1	1	0.4
0	0.67	0.4
1	0.75	0.6
0	0.6	0.6
1	0.67	0.8
0	0.57	0.8
0	0.5	0.8
0	0.44	0.8
1	0.5	1

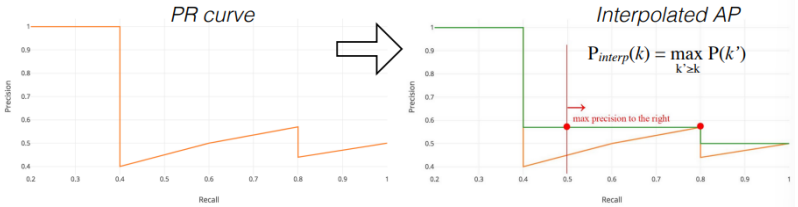
Query: "dog"

$$AP = \frac{1}{5} \cdot (1 + 1 + 0.75 + + 0.67 + 0.5) = 0.78$$

La definizione generale di precisione media (AP) è la seguente :
"trovare l'area sotto la curva precisione-recall di cui sopra:

- Il richiamo aumenta man mano che si scende nella classifica delle previsioni.
- La precisione ha un andamento a "zig-zag" (scende con i falsi positivi e risale con i veri positivi).

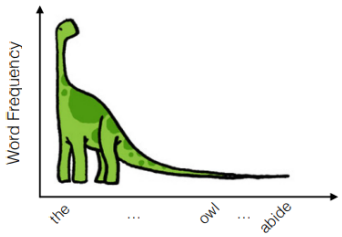
Si esegue quindi una interpolazione per trovare la media come segue:



- Le metriche vengono selezionate su un set di dati/benchmark e misurano ciò che ci interessa (le prestazioni) In genere non possiamo ottimizzare direttamente per le metriche!
- La funzione di perdita (costo) dovrebbe rispecchiare il problema che stiamo risolvendo.
Possiamo sperare che produca modelli che vadano bene sul nostro set di dati.

Ora introduciamo il concetto di *umbalanced data/dati non bilanciati*. I dati sbilanciati si riferiscono a quei tipi di insiemi di dati in cui la classe di riferimento ha una distribuzione non uniforme delle osservazioni, cioè un'etichetta di classe ha un numero molto elevato di osservazioni e l'altra ha un numero molto basso di osservazioni. Avere dati sbilanciati è piuttosto comune.

Distribuzioni a coda lunga/long tail distributions: in quasi tutti gli scenari la distribuzione reale dei dati (etichette) è a coda lunga.



This could be the case of word frequencies in a document

Questo potrebbe essere il caso delle frequenze delle etichette (classe) in un set di dati come ImageNet (database di immagini organizzato a gerarchie di nodi con centinaia/migliaia di immagini utile per lo sviluppo della computer vision/deep learning) o, più in generale, sul web.

L'accuratezza è molto negativa se i dati sono sbilanciati.
Diamo un'altra occhiata al nostro primo esempio: cane vs. non-cane

Accuracy = (4+4)/11 = 0.73 Chance = 0.5

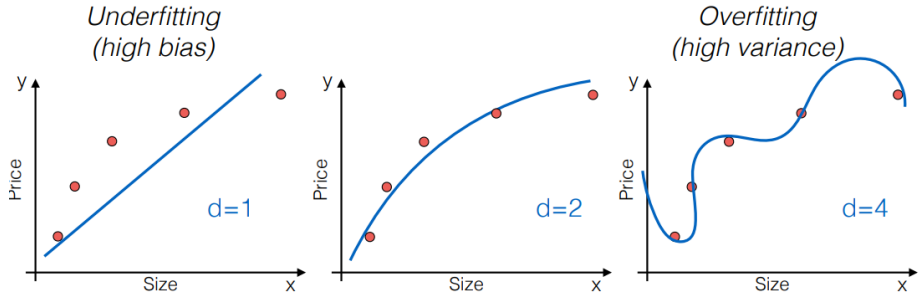
Proviamo con un set di dati diverso (non bilanciato):

$\text{Accuracy}_{alldogs}$ = (8+0)/11 = 0.73 Chance = 0.5

Il prof raccomanda di usare la precisione solo se le classi sono bilanciate
- Precisione-Recall e mAP sono solitamente più robusti
- È possibile calcolare l'accuratezza ponderata prendendo in considerazione tenendo conto del numero di istanze in ogni classe
- Più in generale, in caso di classi multiple è possibile calcolare sia le *micro* che le *macro*-medie cioè si possono calcolare le medie rispetto a ogni classe o a ogni assegnazione di una singola istanza

Diagnosing bias vs variance: Se il modello di apprendimento non funziona come previsto,
quasi sempre è perché si è verificato un problema di un problema di bias elevato o un problema di varianza elevata.
Occorre distinguere se il problema che contribuisce alle cattive previsioni è il bias o la varianza.
Un elevato bias è un underfitting e un'elevata varianza è un overfitting. Idealmente, dobbiamo trovare una media aurea tra questi due elementi.
L'errore di addestramento tenderà a diminuire all'aumentare del grado *d* del polinomio.
Allo stesso tempo, l'errore di convalida incrociata (cross validation error) tenderà a diminuire all'aumentare di d fino a un certo punto, per poi aumentare all'aumentare di *d*, formando una curva convessa.

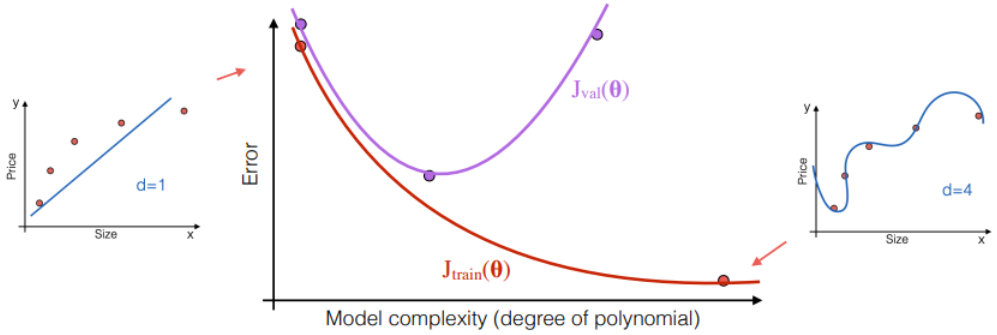
Come capire cosa sta succedendo (in pratica)?



Nell'underfitting, sia l'errore di training che l'errore di cross validation sono grandi e l'errore di cross validation è circa quello di training. Nell'overfitting, l'errore di training sarà basso e quello di cross validation sarà molto più grande di quello di training.

Possiamo ora esaminare di nuovo questo esempio tenendo conto dell'hold-out e del compromesso bias-varianza.

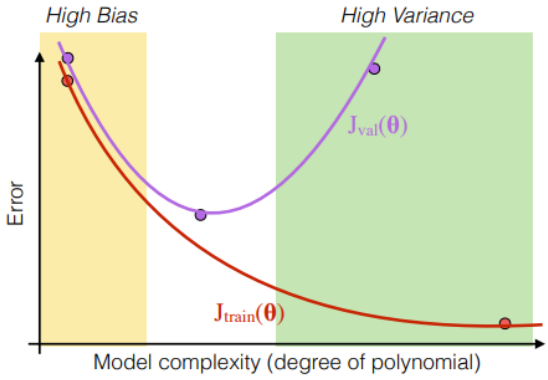
- "Measuring" bias vs variance:
  - Training Error: $J_{train}(\theta) = \frac{1}{2m_t} \sum_{i=1}^{m_t} (h_\theta(x^{(i)}) - y^{(i)})^2$
  - Validation Error: $J_{val}(\theta) = \frac{1}{2m_v} \sum_{i=1}^{m_v} (h_\theta(x^{(i)}) - y^{(i)})^2$



Si nota che si cerca di minimizzare ai minimi quadrati e l'errore di valutazione è compressivamente più alto di quello di training, sulla base del grado *d* del polinomiale.

Complessivamente, però, si hanno due problemi:

- Da una parte l'underfitting (alto bias)
- Dall'altra l'overfitting (alta varianza)



High bias (underfit):

$J_{train}(\theta)$ will be high

$J_{val}(\theta) \approx J_{train}(\theta)$

High variance (overfit):

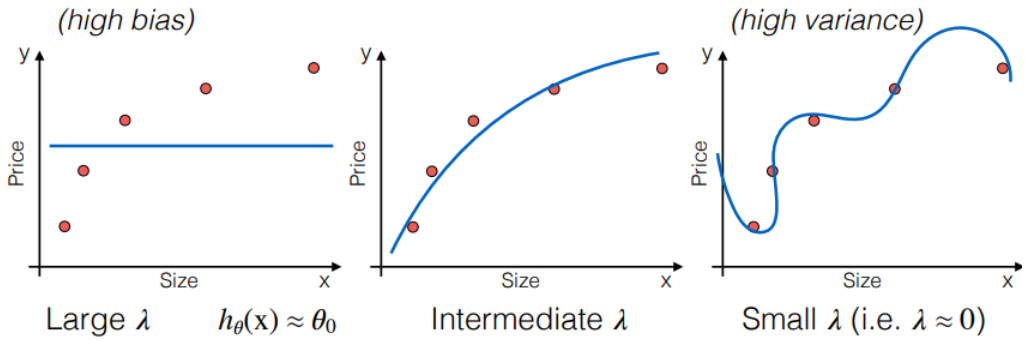
$J_{train}(\theta)$ will be low

$J_{val}(\theta) \gg J_{train}(\theta)$

Qual è il contributo della regolarizzazione?

$$\min_{\theta} \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

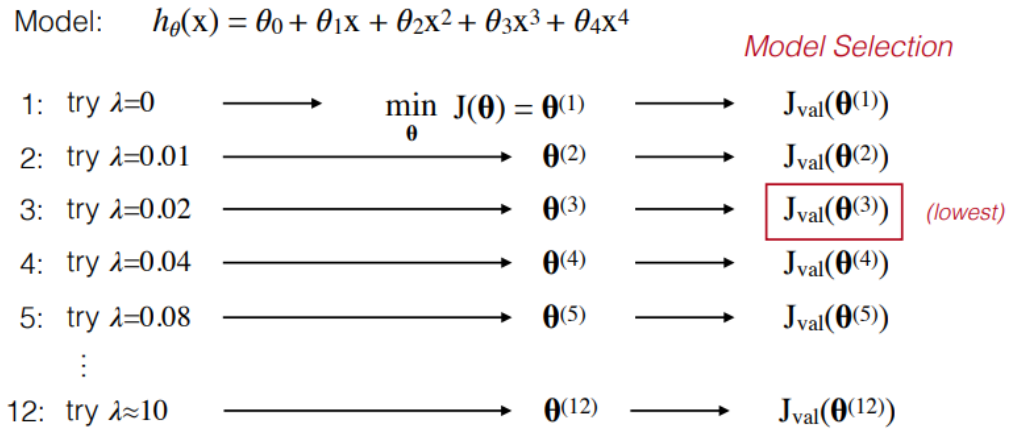
$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$



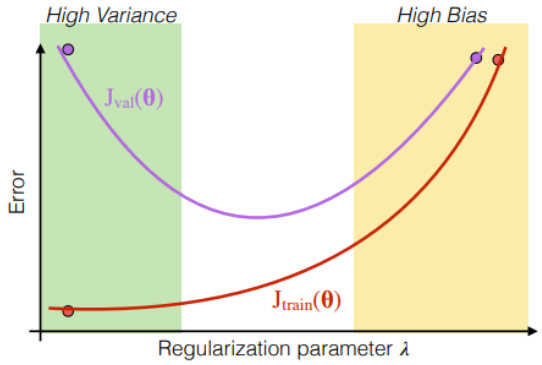
Si consideri la scelta del parametro λ:

- Se λ aumenta, il fit diventa più rigido
- Se λ tende a 0, avremo un overfitting

La nostra definizione degli errori non cambia.
Quello che sarà da fare, dunque è:
- Creare un elenco di lambda
- Creare un insieme di modelli con gradi diversi o altre varianti.
- Iterare i λ e per ogni λ passare in rassegna tutti i modelli per imparare alcuni Θ.
- Calcolare l'errore di validazione incrociata utilizzando i Θ appresi (calcolati con λ) sull'errore di cross validation senza regolarizzazione o λ = 0



La scelta della regolarizzazione evidenza questo andamento che segue:



$$J_{train}(\theta) = \frac{1}{2m_t}\sum_{i=1}^{m_t}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$J_{val}(\theta) = \frac{1}{2m_v}\sum_{i=1}^{m_v}(h_\theta(x^{(i)}) - y^{(i)})^2$$

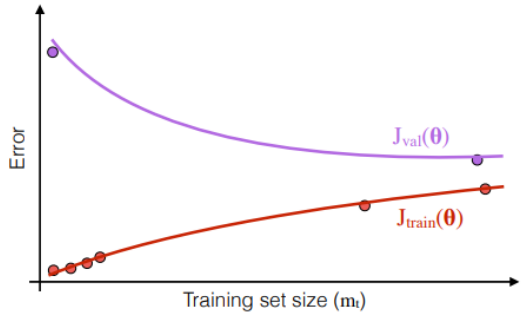
Scritto da Gabriel

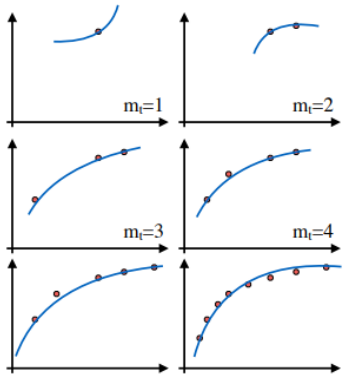
Prendiamo ora tutti gli approfondimenti che abbiamo fatto per costruire uno "strumento di diagnostica" per i sistemi di machine learning.
Si parla in particolare di *learning curves/curve di apprendimento*.

Le curve di apprendimento sono uno strumento diagnostico ampiamente utilizzato nell'apprendimento automatico per gli algoritmi che imparano da un set di dati di addestramento in modo incrementale. Il modello può essere valutato sul dataset di addestramento e su un dataset di convalida dopo ogni aggiornamento durante l'addestramento e i grafici delle prestazioni misurate possono essere creati per mostrare le curve di apprendimento.
L'esame delle curve di apprendimento dei modelli durante l'addestramento può essere utilizzato per diagnosticare problemi di apprendimento, come un modello underfit o overfit, e per verificare se i dataset di addestramento e di validazione sono adeguatamente rappresentativi.

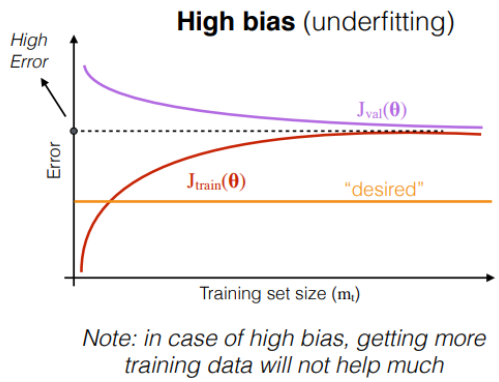
Le curve di apprendimento possono essere utilizzate per diagnosticare se un modello può soffrire di bias, varianza o un po' di entrambi. Quella che segue è l'intuizione generale:

$$J_{train}(\boldsymbol{\theta}) = \frac{1}{2m_t}\sum_{i=1}^{m_t}(h_\theta(x^{(i)}) - y^{(i)})^2 \qquad J_{val}(\boldsymbol{\theta}) = \frac{1}{2m_v}\sum_{i=1}^{m_v}(h_\theta(x^{(i)}) - y^{(i)})^2$$

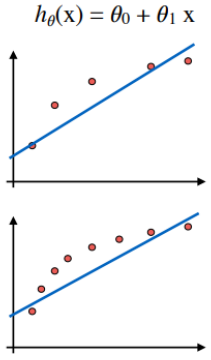




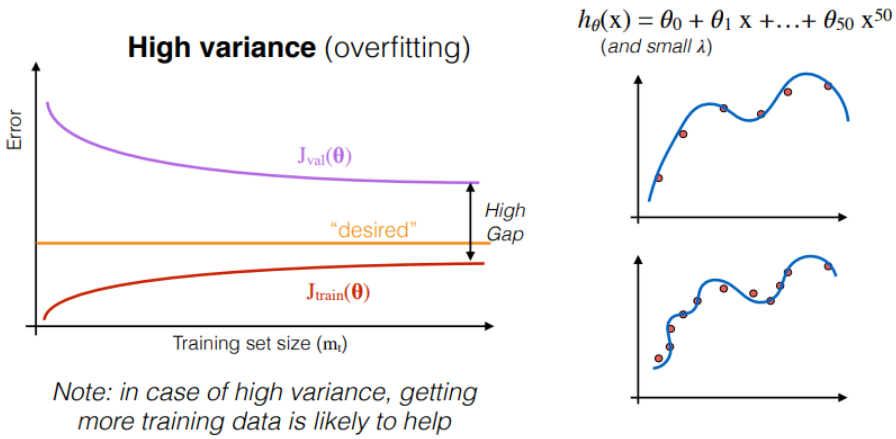
In merito invece ai problemi di bias e varianza:



Note: in case of high bias, getting more training data will not help much



In caso di alto bias, avere più dati di addestramento non aiuterà molto. Il bias si riferisce all'errore dovuto a ipotesi troppo semplicistiche o a presupposti errati nell'algoritmo di apprendimento. Il bias si traduce in un adattamento insufficiente ai dati. Un bias elevato significa che il nostro algoritmo di apprendimento non ha colto tendenze importanti tra le caratteristiche.

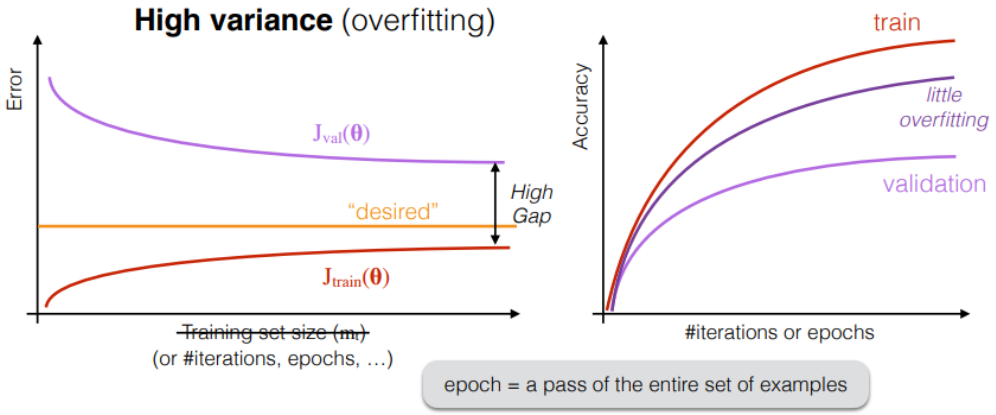


Note: in case of high variance, getting more training data is likely to help

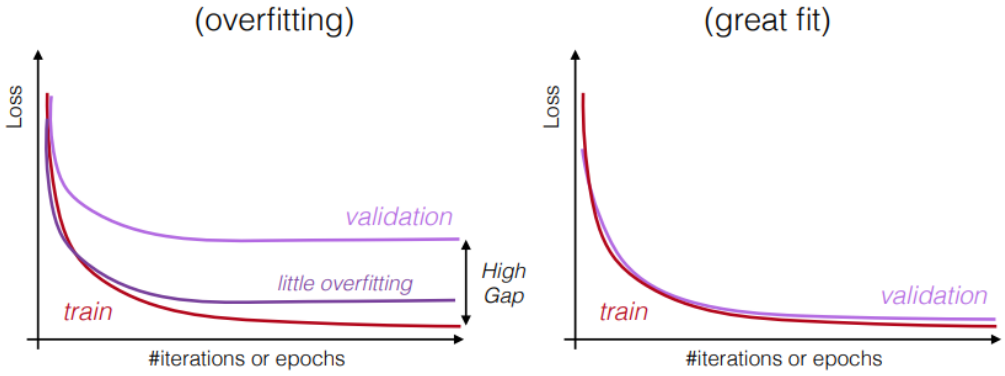
La varianza si riferisce all'errore dovuto a un modello eccessivamente complesso che cerca di adattarsi il più possibile ai dati di addestramento. Nei casi di varianza elevata, i valori previsti dal modello sono estremamente vicini ai valori reali del set di formazione. L'algoritmo di apprendimento copia le tendenze dei dati di addestramento e questo comporta una perdita di capacità di generalizzazione. L'alta varianza dà luogo a un over-fitting.

Un modello con un'elevata varianza, se testato su dati non visti, non darà risultati soddisfacenti. Ciò è dovuto al fatto che l'algoritmo è molto sensibile ad alti gradi di variazione nei dati di addestramento, con il risultato di trasportare troppo rumore dai dati di addestramento perché il modello possa essere utile per i dati di test.

È possibile computare differenti curve di apprendimento con diverse "dimensioni" (come misure di valutazione, numero di campionamenti, ecc. Si consideri: *epoch* → Insieme di esempi.



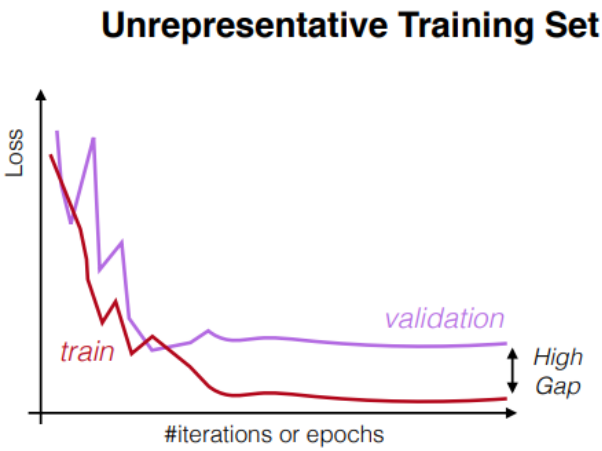
Altre curve di apprendimento sono disegnate con riferimento alla perdita, secondo la serie di iterazioni/epochs, avendo a seconda del caso un fit più o meno adatto e migliore.



Da qui, diventa utile debuggare e seguire un algoritmo di apprendimento (in gergo, *babysitting*). Supponiamo di aver implementato un modello di regressione lineare regolarizzata per prevedere i prezzi delle abitazioni. Non funziona su nuovi dati; cosa fare dopo?

- Si possono ottenere più dati di addestramento → Corregge l'elevata varianza
- Si prova un set più piccolo di features → Corregge l'alta varianza
- Si cerca di ottenere più caratteristiche → Corregge il bias alto
- Si cerca di aggiungere complessità al modello (ad es. caratteristiche polinomiali) → Corregge il bias alto
- Si cerca di diminuire λ → Corregge il bias alto
- Si cerca di aumentare λ → Corregge l'alta varianza.

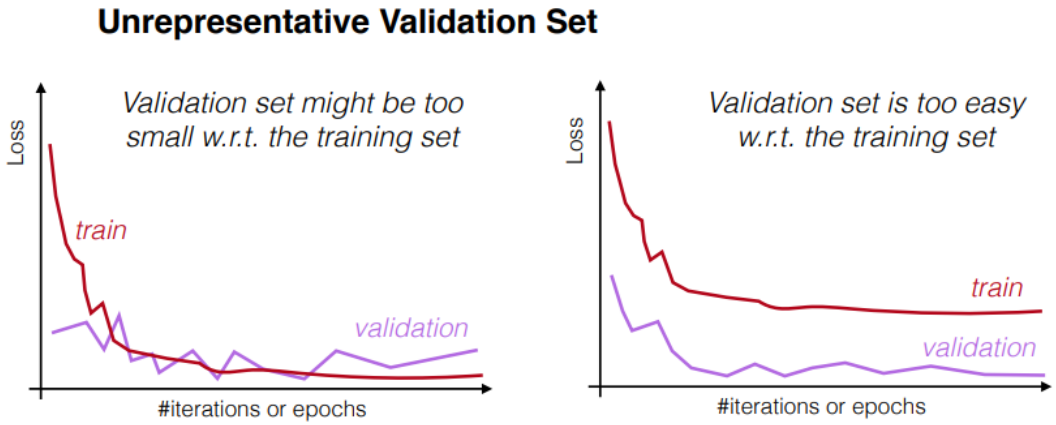
Le curve di apprendimento possono essere utilizzate anche per diagnosticare la qualità dei nostri set di addestramento/validazione.



- L'insieme di addestramento non fornisce informazioni sufficienti per apprendere il problema
- Può verificarsi se l'insieme di addestramento ha un numero troppo basso di esempi rispetto all'insieme di validazione

Un set di dati di addestramento non rappresentativo significa che il set di dati di addestramento non fornisce informazioni sufficienti per l'apprendimento del problema, rispetto al set di dati di validazione utilizzato per valutarlo. Questa situazione può essere identificata da una curva di apprendimento per la perdita di addestramento che mostra un miglioramento e, allo stesso modo, da una curva di apprendimento per la perdita di validazione che mostra un miglioramento, ma rimane un grande divario tra entrambe le curve. Ciò può verificarsi quando

Il dataset di addestramento ha un numero troppo basso di esempi rispetto al dataset di validazione.
Il dataset di addestramento contiene caratteristiche con una varianza minore rispetto al dataset di validazione.



Un dataset di validazione non rappresentativo significa che il dataset di validazione non fornisce informazioni sufficienti per valutare la capacità di generalizzazione del modello. Questo può accadere se il dataset di validazione ha un numero di esempi troppo basso rispetto al dataset di addestramento. Questo

caso può essere identificato da una curva di apprendimento per la perdita di addestramento che sembra un buon adattamento (o altri adattamenti) e una curva di apprendimento per la perdita di convalida che mostra movimenti rumorosi e miglioramenti minimi o nulli.

Artificial Neural Networks I

Le reti neurali sono un insieme di algoritmi, modellati sul modello del cervello umano, progettati per riconoscere gli schemi. Interpretano i dati sensoriali attraverso una sorta di percezione artificiale, etichettando o raggruppando gli input grezzi. I modelli che riconoscono sono numerici, contenuti in vettori, in cui devono essere tradotti tutti i dati del mondo reale, siano essi immagini, suoni, testi o serie temporali.

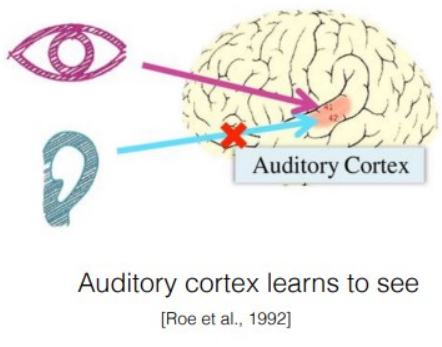
Le reti neurali ci aiutano a raggruppare e classificare. Possono essere considerate come un livello di raggruppamento e classificazione sopra i dati memorizzati e gestiti. Aiutano a raggruppare i dati non etichettati in base alle somiglianze tra gli esempi in ingresso e classificano i dati quando hanno un set di dati etichettati su cui allenarsi. (Le reti neurali possono anche estrarre caratteristiche che vengono poi fornite ad altri algoritmi per il raggruppamento e la classificazione; si può quindi pensare alle reti neurali profonde come componenti di applicazioni di apprendimento automatico più ampie che coinvolgono algoritmi di apprendimento per rinforzo, classificazione e regressione).

Concretamente, sono algoritmi modellati (vagamente) sul cervello.

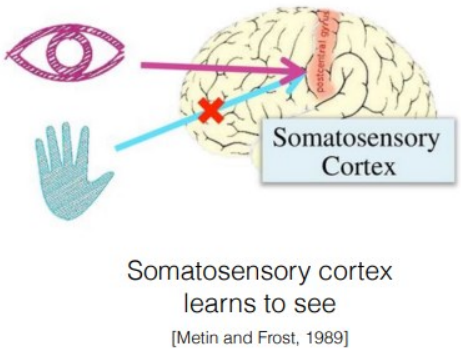
- Le reti neurali sono state introdotte alla fine degli anni '50.
- Sono state ampiamente utilizzate negli anni '80 e nei primi anni '90; la loro popolarità è diminuita alla fine degli anni '90.
- Recente "rinascita": fine anni 2000/inizio 2010 (rivoluzione del deep learning).
- Le reti neurali artificiali non sono così complesse o intricate come la struttura del cervello.

Si ha poi avuto un boom del machine learning come conseguenza di tutto ciò.

Si parte dalla cosiddetta *"one learning algorithm" hypothesis*, per cui è stato dimostrato che il cervello umano utilizza essenzialmente lo stesso algoritmo per comprendere diverse modalità di input.
Ad esempio, i vari sensori neurali, per sentire da un punto di vista di udito e di tatto, considerano la stessa modalità di apprendimento.



Auditory cortex learns to see
[Roe et al., 1992]

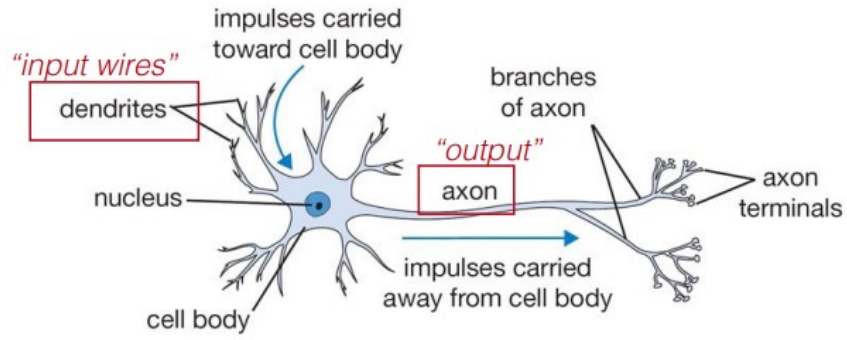


Somatosensory cortex learns to see
[Metin and Frost, 1989]

Esempio: Esperimenti sui furetti, in cui l'"input" per la visione è stato inserito nella parte uditiva del cervello, e la corteccia uditiva impara a "vedere".
[Roe et al., 1992].

A causa di questi esperimenti si ha la sensazione che se la stessa parte del cervello può elaborare la vista, il suono o il tatto, allora (forse) esiste un unico algoritmo di apprendimento. Pertanto, invece di progettare centinaia di algoritmi diversi, dobbiamo approssimare questo unico algoritmo.
Le applicazioni delle reti neurali e di questa intuizione sono presenti ovunque.

Cominciamo con l'osservare l'aspetto di un singolo neurone nel cervello.



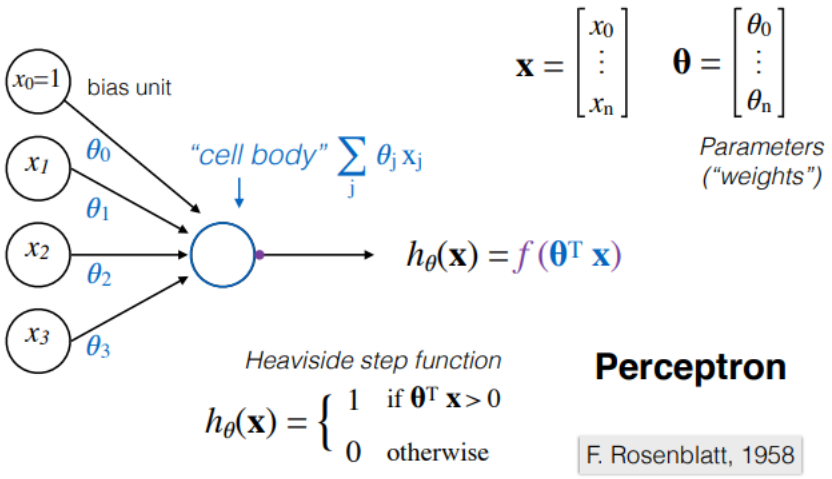
I neuroni possono essere visti solo al microscopio e possono essere suddivisi in tre parti:

1) Soma (corpo cellulare) - questa parte del neurone riceve le informazioni. Contiene il nucleo della cellula.
2) Dendriti - questi sottili filamenti trasportano le informazioni da altri neuroni al soma. Sono la parte di "ingresso" della cellula.
3) Assone - questa lunga proiezione trasporta le informazioni dal soma e le invia ad altre cellule. È la parte "in uscita" della cellula. Normalmente termina con una serie di sinapsi che si collegano ai dendriti di altri neuroni.

Sia i dendriti che gli assoni sono talvolta indicati come fibre nervose.
Una cosa interessante è che se un neurone riceve un gran numero di input da altri neuroni, questi segnali si sommano fino a superare una determinata soglia.
Una volta superata questa soglia, il neurone si attiva per inviare un impulso lungo il suo assone, chiamato potenziale d'azione.

Definiamo un modello semplice per un neurone artificiale (unità di calcolo), definita come *perceptron/percettrone.* Esso è composto da:

1) Valori di input o strato ad input singolo
2) Pesi e bias
3) Somma netta
4) Funzione di attivazione



In particolare, un perceptron funziona seguendo questi passi:
a. Tutti gli ingressi *x* vengono moltiplicati per i loro pesi.
b. Si sommano tutti i valori moltiplicati e vengono definiti come somma ponderata.
c. Applicare la somma ponderata alla funzione di attivazione corretta.

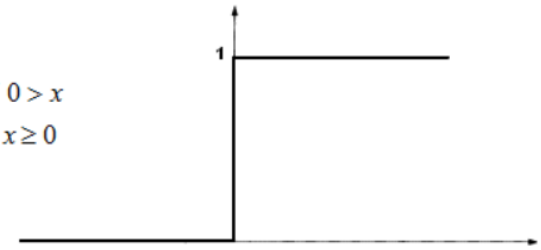
Un esempio di funzione di attivazione corretta: *Heaviside Step Function/Unit Step Activation Function*.

Heaviside step function

$$h_\theta(\mathbf{x}) = \begin{cases} 1 & \text{if } \boldsymbol{\theta}^T \mathbf{x} > 0 \\ 0 & \text{otherwise} \end{cases}$$

Unit step (threshold)

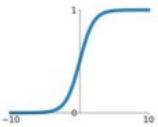
$$f(x) = \begin{cases} 0 \text{ if } 0 > x \\ 1 \text{ if } x \geq 0 \end{cases}$$



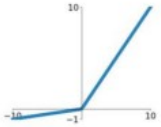
Come visto sopra, usiamo parametri e pesi. Esiste anche un valore di bias qui in mezzo
I pesi indicano la forza di un particolare nodo. Un valore di bias consente di spostare la curva della funzione di attivazione verso l'alto o verso il basso.

Quindi, il percettrone viene solitamente utilizzato per classificare i dati in due parti. Pertanto, è noto anche come *classificatore binario lineare*.
Al giorno d'oggi esistono diverse funzioni di attivazione che è possibile utilizzare:

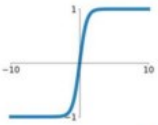
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

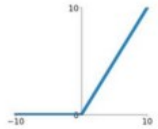


tanh
$\tanh(x)$

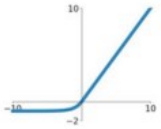


Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$



ELU
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Cercando di dare un attimo un contesto a ciascuna di queste.
Le funzioni di attivazione non lineari sono preferite in quanto consentono ai nodi di apprendere strutture più complesse nei dati. Tradizionalmente, due funzioni di attivazione non lineari molto utilizzate sono la *sigmoide* e la *tangente iperbolica (tanh)*.

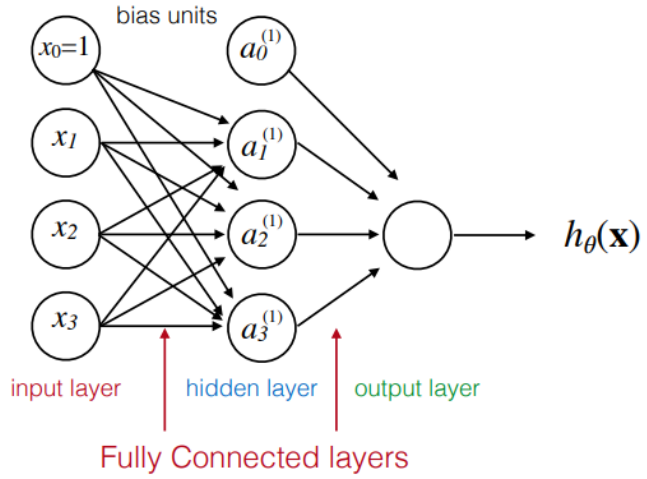
- *La funzione di attivazione sigmoide*, detta anche *funzione logistica*, è tradizionalmente una funzione di attivazione molto diffusa per le reti neurali. L'input della funzione viene trasformato in un valore compreso tra 0,0 e 1,0. Gli input molto più grandi di 1,0 vengono trasformati nel valore 1,0 e, allo stesso modo, i valori molto più piccoli di 0,0 vengono trasformati in 0,0. La forma della funzione per tutti i possibili ingressi è una forma a S da zero fino a 0,5 e 1,0. Per molto tempo, fino ai primi anni '90, è stata l'attivazione predefinita utilizzata nelle reti neurali.
- La *funzione tangente iperbolica*, o *tanh* in breve, è una funzione di attivazione non lineare di forma simile che produce valori compresi tra -1,0 e 1,0. Negli ultimi anni Novanta e negli anni Duemila, la funzione tanh è stata preferita alla funzione di attivazione sigmoide, in quanto i modelli che la utilizzavano erano più facili da addestrare e spesso avevano prestazioni predittive migliori.

Un problema generale delle funzioni sigmoide e tanh è la loro saturazione. Ciò significa che i valori grandi scattano a 1,0 e quelli piccoli a -1 o 0, rispettivamente per tanh e sigmoide. Inoltre, le funzioni sono veramente sensibili solo alle variazioni intorno al punto medio del loro ingresso, come 0,5 per la sigmoide e 0,0 per la tanh. La sensibilità limitata e la saturazione della funzione avvengono indipendentemente dal fatto che l'attivazione sommata del nodo fornito in ingresso contenga o meno informazioni utili. Una volta satura, diventa difficile per l'algoritmo di apprendimento continuare a adattare i pesi per migliorare le prestazioni del modello.

Scritto da Gabriel

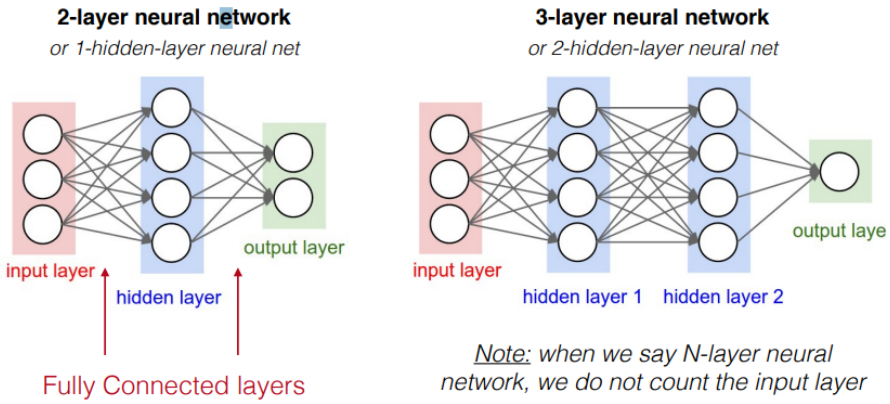
- La *funzione di attivazione lineare rettificata*, o in breve *ReLU*, è una funzione lineare a tratti che emette direttamente l'input se è positivo, altrimenti emette zero. È diventata la funzione di attivazione predefinita per molti tipi di reti neurali perché un modello che la utilizza è più facile da addestrare e spesso ottiene prestazioni migliori.
    - *La Leaky Rectified Linear Unit, o Leaky ReLU*, è un tipo di funzione di attivazione basata su una ReLU, ma con una piccola pendenza per i valori negativi invece di una pendenza piatta. Il coefficiente di pendenza viene determinato prima dell'addestramento, cioè non viene appreso durante l'addestramento. Questo tipo di funzione di attivazione è popolare nei compiti in cui si può soffrire di gradienti scarsi, ad esempio nell'addestramento di reti generative avversarie.
- L'unità Maxout è una generalizzazione delle funzioni ReLU e ReLU Leaky. Si tratta di una funzione lineare continua che restituisce il massimo degli ingressi, progettata per essere utilizzata insieme al dropout. Sia la ReLU che la ReLU con perdita sono casi speciali di Maxout.

Una rete neurale (artificiale) è solo un gruppo di questi neuroni diversi connessi insieme.



Gli *strati completamente connessi* in una rete neurale sono quelli in cui tutti gli ingressi di uno strato sono collegati a ogni unità di attivazione dello strato successivo. Nella maggior parte dei modelli di apprendimento automatico più diffusi, gli ultimi strati sono strati completamente connessi che compilano i dati estratti dagli strati precedenti per formare l'output finale. È il secondo strato che richiede più tempo.

Il termine *architettura* riferito alle reti neurali si riferisce a come i differenti neuroni sono collegati tra di loro, in particolare come segue.



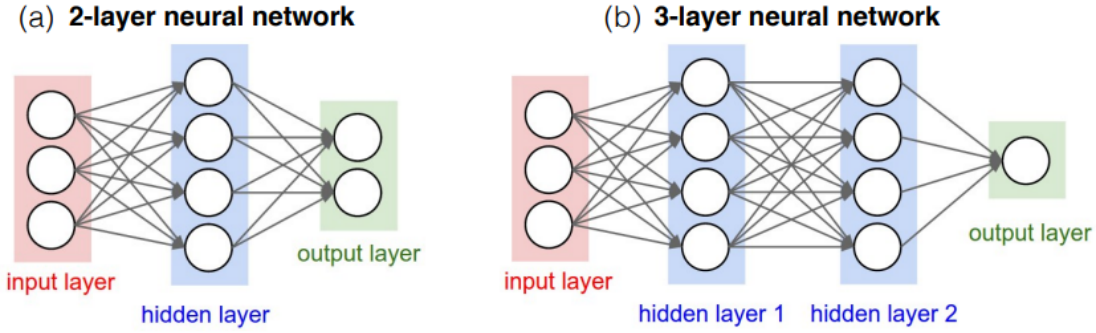
In particolare, si considera che ci sarà sempre:

- Uno strato di input. Come suggerisce il nome, accetta input in diversi formati forniti dal programmatore.
- Uno strato di output. L'input viene sottoposto a una serie di trasformazioni tramite lo strato nascosto, che alla fine si traduce in un output che viene trasmesso tramite questo strato.
  La rete neurale artificiale prende l'input e calcola la somma ponderata degli input, includendo un bias. Questo calcolo è rappresentato sotto forma di funzione di trasferimento.

- Degli strati intermedi (di numero crescente in base alla complessità della rete neurale), chiamati strati nascosti (hidden layers). Il livello nascosto si trova tra i livelli di ingresso e di uscita. Esegue tutti i calcoli per trovare caratteristiche e modelli nascosti.

Esempio concreto potrebbe provenire ad esempio da:
https://www.sciencedirect.com/topics/engineering/three-layered-neural-network
Settando i valori iniziali, si considerano gli input dei singoli neuroni e si calcolano gli errori generalizzati per ognuno. Più strati nascosti (hidden) ci sono e più la misura tende ad essere corretta dal calcolo dei pesi secondo il metodo del gradiente.

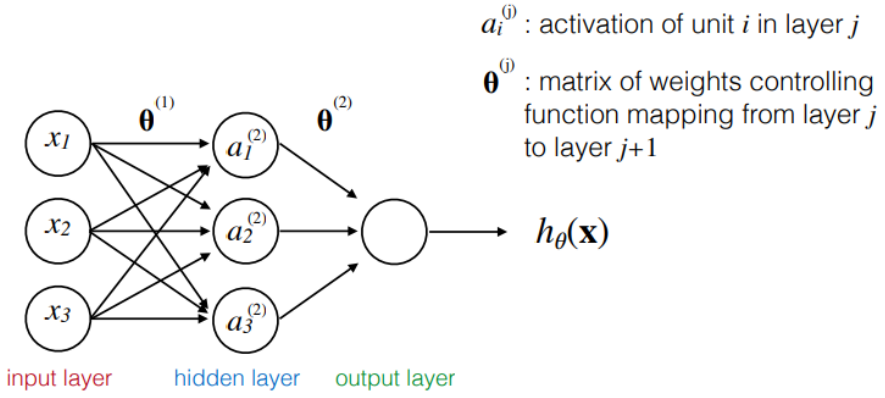
Altri esempi utili sono le:
Sizing neural networks/Dimensionamento delle reti neurali: qui, le due metriche comunemente utilizzate sono i #*neuroni* o (più spesso) i #*parametri*.



(b) has 4+4+1 neurons; how many (learnable) parameters?

(b) #parameters: 3x4 + 4x4 + 4x1 = 32 + 9 *(biases)* = 41

Una rete neurale (artificiale) è solo un gruppo di questi neuroni diversi forti insieme e, nel campo dell'intelligenza artificiale, cerca di imitare la rete di neuroni che compongono il cervello umano, in modo che i computer abbiano la possibilità di capire le cose e prendere decisioni in modo simile a quello umano. La rete neurale artificiale è progettata programmando i computer in modo che si comportino semplicemente come cellule cerebrali interconnesse.



Partendo dalla funzione di attivazione delle singole unità, si forma una matrice di pesi controllanti il mapping della funzione da strato a strato, per raggiungere l'output voluto.

- Le unità di ingresso sono impostate da una funzione esterna (può essere generata dai sensori) che provoca l'attivazione dei loro collegamenti di uscita al livello specificato.
- Procedendo attraverso la rete, queste uscite saranno l'input per il livello successivo.
Ciascuna uscita è solo la somma ponderata dell'attivazione dei collegamenti che alimentano un nodo.
- La funzione di attivazione trasforma questa combinazione lineare: in genere si tratta di una funzione non lineare (ad esempio una sigmoide). Questa funzione corrisponde alla "soglia" di quel nodo.

Ora si ha un esempio di *feedforward neural network*.

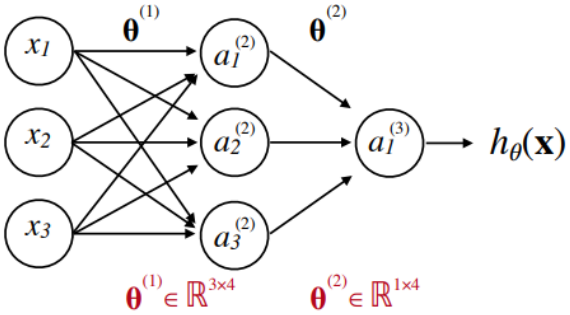
Una rete neurale *feedforward* è un tipo di rete neurale artificiale in cui le connessioni dei nodi non formano un ciclo. Spesso definite come una rete multistrato di neuroni, le reti neurali feedforward sono così chiamate perché tutte le informazioni fluiscono solo in avanti.
I dati entrano nei nodi di ingresso, attraversano gli strati nascosti e infine escono dai nodi di uscita. La rete è priva di collegamenti che permettano alle informazioni che escono dal nodo di uscita di essere inviate nuovamente alla rete.

Lo scopo delle reti neurali feedforward è quello di *approssimare funzioni*.
Ecco come funziona:

- Esiste un classificatore che utilizza la formula $y = f*(x)$.
- Questo assegna il valore dell'input *x* alla categoria *y*.
- La rete feedforward mapperà $y = f(x; \theta)$. Quindi memorizza il valore di θ che più si avvicina alla funzione.

Normalmente, una rete neurale feedforward funge da base per il rilevamento degli oggetti nelle foto.

Le reti neurali sono costituite da due elementi principali che eseguono operazioni matematiche. I neuroni calcolano somme ponderate utilizzando i dati di ingresso e i pesi sinaptici, poiché le reti neurali sono solo calcoli matematici basati sui collegamenti sinaptici.
In pratica, dato:



Le somme possono essere espresse come vettori (o somme di vettori):

$$a_1^{(2)} = f(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = f(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = f(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3)$$

$$h_\theta(\mathbf{x}) = a_1^{(3)} = f(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)})$$

Questo si può computare, se gli elementi hanno diverse dimensioni, attraverso matrici.
Ad esempio, se la rete ha u_j unità nello strato *j* e u_{j+1} unità nello strato *j+1*, allora θ sarà di dimensione u_{j+1} *x* $(u_j + 1)$.
Per mettere insieme i vari vettori, viene introdotta la notazione apposita per indicare la *combinazione lineare* tra gli stessi; quindi, un'espressione dove possono comparire scalari e vettori ed esistono somme e prodotti vettoriali.

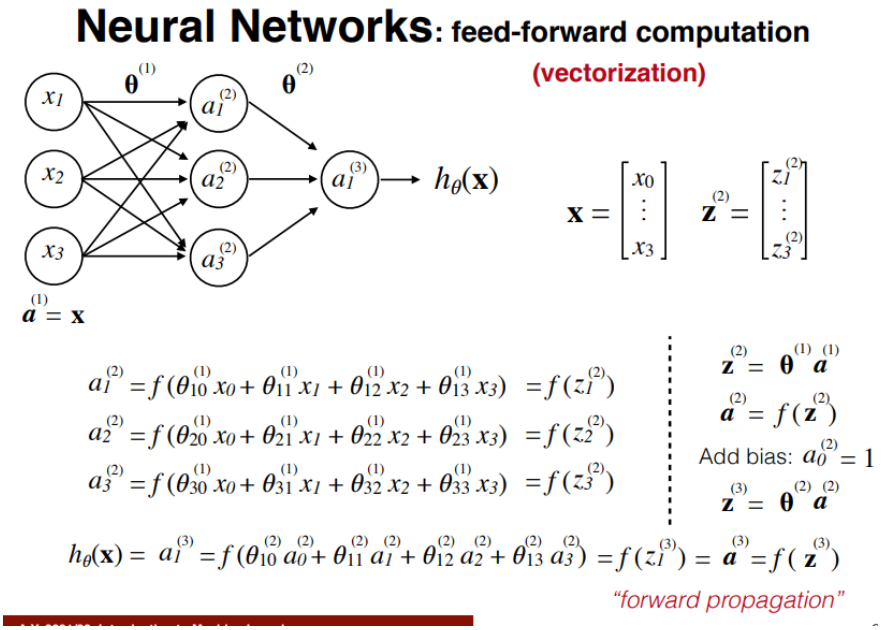
$z_i^{(j)}$: linear combination of the input nodes

$$a_1^{(2)} = f(\underbrace{\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3}_{z_1^{(2)}}) = f(z_1^{(2)})$$
$$a_2^{(2)} = f(\theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3) = f(z_2^{(2)})$$
$$a_3^{(2)} = f(\theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3) = f(z_3^{(2)})$$

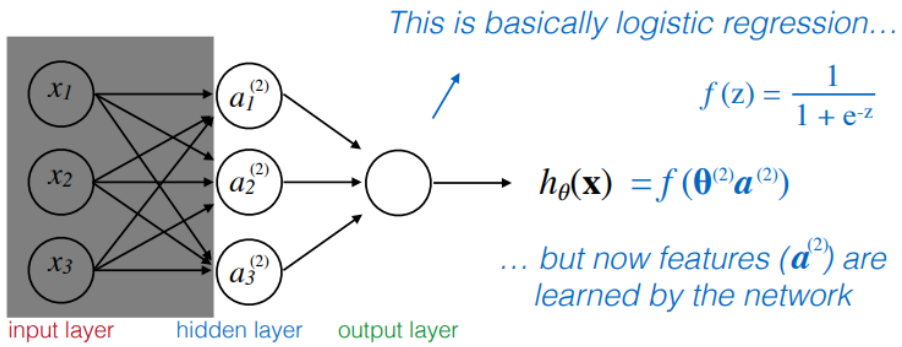
$$h_\theta(\mathbf{x}) = a_1^{(3)} = f(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)}) = f(z_1^{(3)})$$

Si parla, qui, di *forward propagation*, che si riferisce all'immagazzinamento e al calcolo dei dati di ingresso che vengono alimentati in avanti attraverso la rete per generare un'uscita. Gli strati nascosti della rete neurale accettano i dati dallo strato di ingresso, li elaborano sulla base della funzione di attivazione e li passano allo strato di uscita o agli strati successivi. I dati fluiscono in avanti, in modo da evitare un flusso

circolare di dati che non genera un output. La configurazione della rete che aiuta la propagazione in avanti è proprio la rete feed-forward.
La propagazione in avanti è il modo in cui le reti neurali fanno previsioni. I dati in ingresso vengono "propagati in avanti" attraverso la rete, strato per strato, fino allo strato finale che emette una previsione.
In tutto questo, si sottolinea nuovamente la vettorizzazione (cercare combinazioni di vettori) operata.



Questa visione di propagazione in avanti ci aiuta anche a comprendere ciò che le reti neurali potrebbero fare. Avendo uno strato che calcola e uno strato che dà in output qualcosa, si avvicina ad una visione vicina alla logistic regression, ma ora le caratteristiche/features $a^{(2)}$ sono apprese dalla rete.

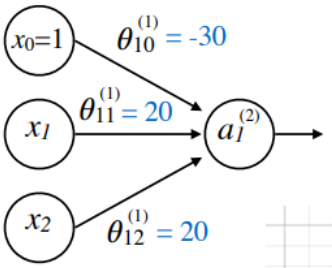


Ora, degli esempi concreti di computazione di funzioni logiche:

$x_1, x_2 \in \{0, 1\}$

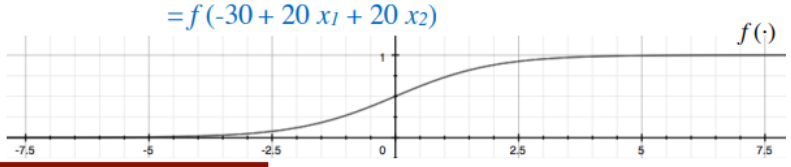
$y = x_1$ AND x_2

x_1	x_2	y	$h_\theta(\mathbf{x})$
0	0	0	$= f(-30) \approx 0$
0	1	0	$= f(-10) \approx 0$
1	0	0	$= f(-10) \approx 0$
1	1	1	$= f(10) \approx 1$



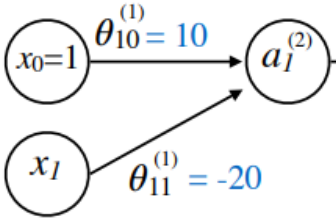
$h_\theta(\mathbf{x}) = a_1^{(2)} = f(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2)$

$= f(-30 + 20\, x_1 + 20\, x_2)$



Segue la rappresentazione di funzioni booleane. Ogni funzione booleana può essere rappresentata da una rete con un singolo strato nascosto; tuttavia, potrebbe richiedere un numero esponenziale (in numero di input) di unità nascoste.

$y = (\text{NOT } x_1)$

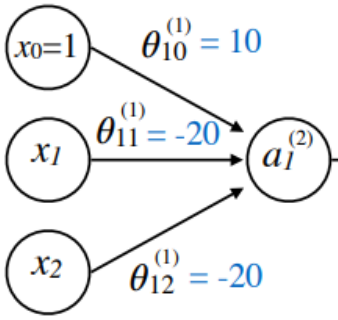


$h_\theta(\mathbf{x}) = f(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1)$

$= f(10 - 20\, x_1)$

x_1	y	$h_\theta(\mathbf{x})$
0	1	$= f(10) \approx 1$
1	0	$= f(-10) \approx 0$

$y = (\text{NOT } x_1)$ AND $(\text{NOT } x_2)$



$h_\theta(\mathbf{x}) = f(\theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2)$

$= f(10 - 20\, x_1 - 20\, x_2)$

x_1	x_2	y	$h_\theta(\mathbf{x})$
0	0	1	$= f(10) \approx 1$
0	1	0	$= f(-10) \approx 0$
1	0	0	$= f(-10) \approx 0$
1	1	0	$= f(-30) \approx 0$

Concludendo, è interessante osservare che nel 1969 Minsky e Seymour dimostrarono che era impossibile per i percettori apprendere una funzione XOR
Spesso viene riportato (erroneamente) che i due abbiano anche ipotizzato che un risultato simile sarebbe stato valido per i percettori multistrato.
Ciononostante, ha causato un significativo declino nell'interesse e nei finanziamenti della ricerca sulle reti neurali, ripreso dagli anni ’80 in avanti.

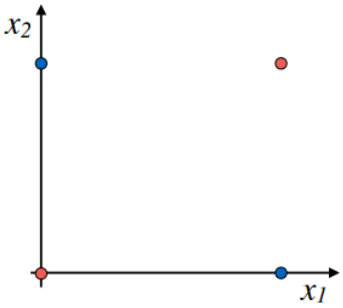
Artificial Neural Networks II

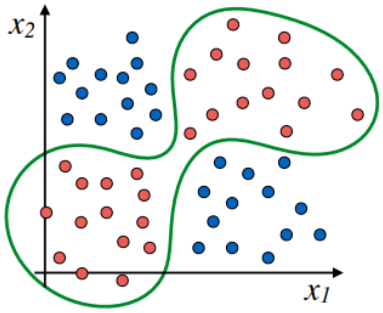
Tornando alle funzioni booleane, si può pensare a questo come a una versione semplificata di un problema di classificazione più complesso problema di classificazione (non lineare).
Nell'esempio che segue, delle funzioni XOR e XNOR rispetto ad una serie di input:

$x_1, x_2 \in \{0, 1\}$ (binary)

$y = x_1$ XOR x_2 / $y = x_1$ XNOR x_2

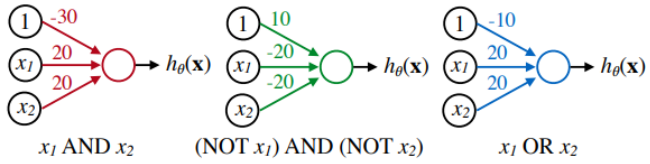
- Positive class
- Negative class





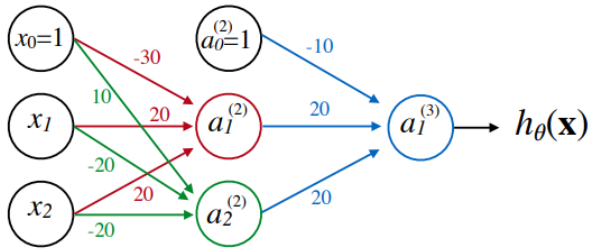
I dati possono essere combinati (a loro volta con altre serie di reti neurali) per rappresentare funzioni non lineari, come segue. Non è molto diverso da quanto detto sinora.

• Combining representations for non-linear functions



x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	y	$h_\theta(\mathbf{x})$
0	0	≈ 0	≈ 1	1	≈ 1
0	1	≈ 0	≈ 0	0	≈ 0
1	0	≈ 0	≈ 0	0	≈ 0
1	1	≈ 1	≈ 0	1	≈ 1

$y = x_1$ XNOR x_2

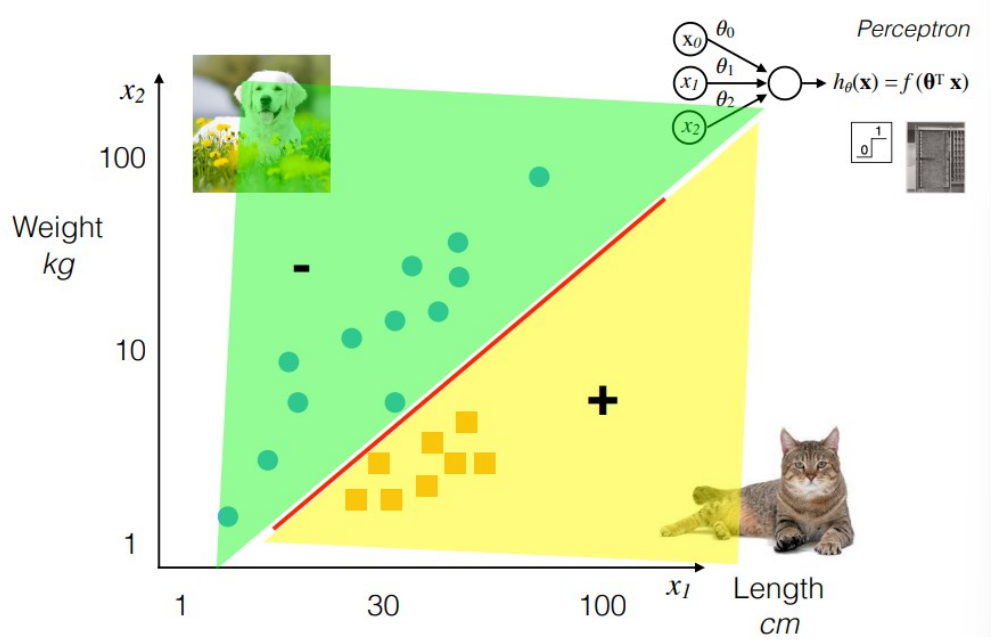


Introduciamo un classificatore binario per riconoscere gatti e cani (basato su due semplici caratteristiche, lunghezza e peso). Usando un percettrone, esso predice l'output come segue:

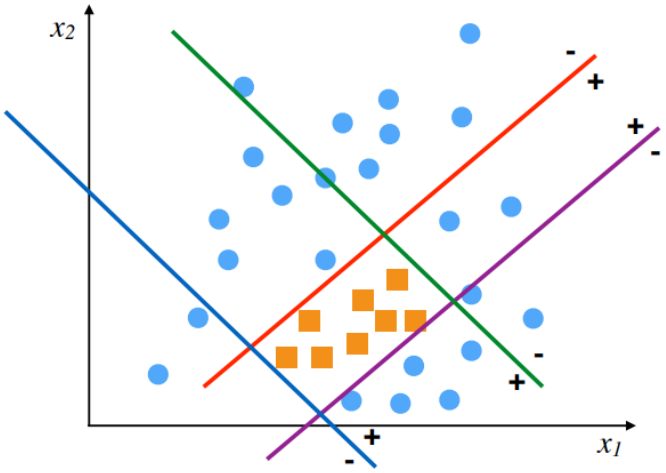
$$y_j = f[\mathbf{w}^T\mathbf{x}] = f[\vec{w} \cdot \vec{x}] = f[w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n]$$

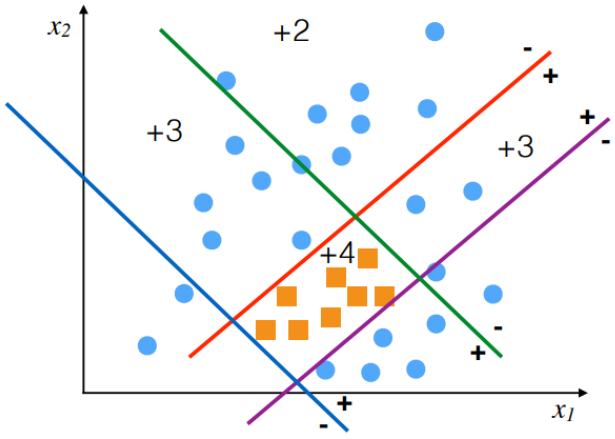
avendo che il peso *w* contiene un termine di bias w_0. In sostanza, per ogni input del training set, partendo dalle dimensioni degli elementi considerati, si hanno degli aggiustamenti per portarli ciascuno alla corretta dimensione e quindi dare un corretto aggiornamento del vettore peso per correggere l'errore tra output previsto e l'output effettivo.

Quanto descritto poco fa viene realizzato dal seguente esempio grafico:

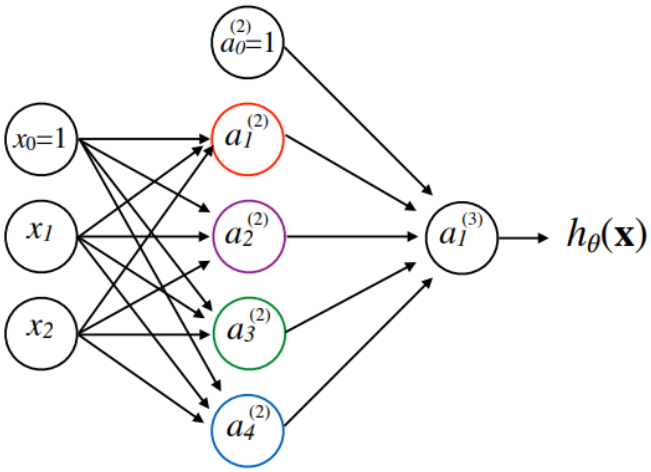


Ora generalizziamo l'esempio al caso "gatto" vs "non-gatto", considerando un esempio su una funzione e sulla combinazione delle rappresentazioni su funzioni non lineari.

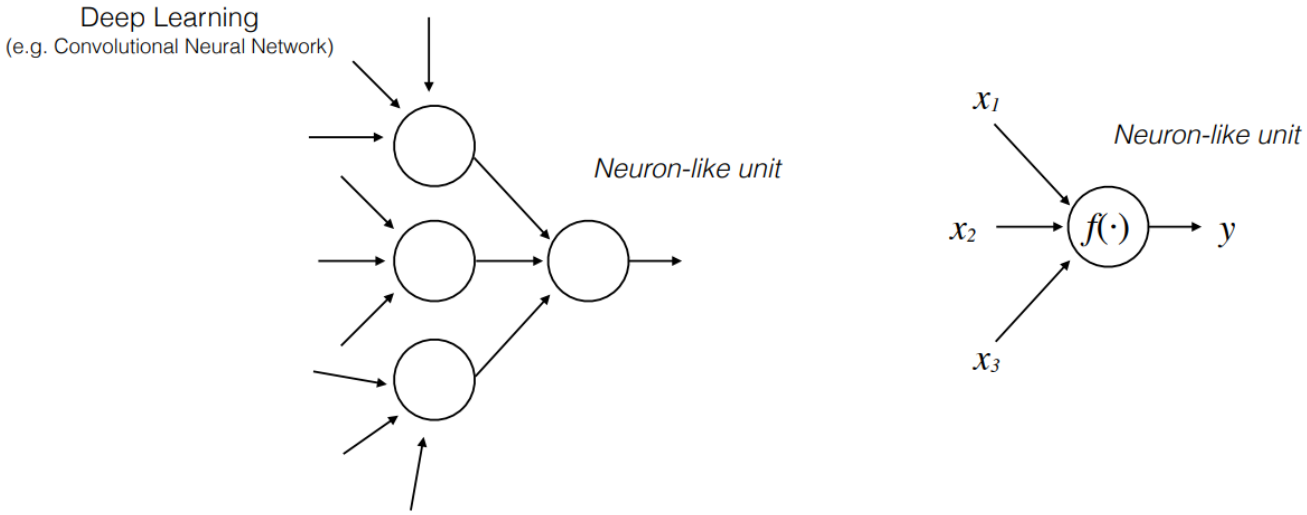




Come detto sopra, la rappresentazione per funzioni non lineari può essere anche descritta da una rete neurale il cui scopo è dare un output unico:



Prendiamo ora le *deep neural networks*, le quali considerano grandi quantità di dati, non si conoscono le particolari caratteristiche dei neuroni rappresentanti, pesanti da un punto di vista computazionale e si impiega molto tempo ad addestrarle.
Si impiega qui una *convolutional neural network*, (ConvNet/CNN) è un algoritmo di apprendimento profondo (deep learning) che è in grado di prendere in considerazione un'immagine in ingresso, assegnare un'importanza (pesi e bias apprendibili) a vari aspetti/oggetti dell'immagine ed essere in grado di differenziarli l'uno dall'altro. La preelaborazione richiesta in una ConvNet è molto inferiore rispetto ad altri algoritmi di classificazione. Mentre nei metodi primitivi i filtri vengono creati a mano, con un addestramento sufficiente le ConvNet hanno la capacità di apprendere questi filtri/caratteristiche.



L'architettura di una ConvNet è analoga a quella del modello di connettività dei neuroni nel cervello umano ed è stata ispirata dall'organizzazione della corteccia visiva. I singoli neuroni rispondono agli stimoli solo in una regione ristretta del campo visivo, nota come campo recettivo. Un insieme di tali campi si sovrappone per coprire l'intera area visiva.
Una ConvNet è in grado di *catturare con successo le dipendenze spaziali e temporali di un'immagine attraverso l'applicazione di filtri pertinenti*. L'architettura si adatta meglio al set di dati dell'immagine grazie alla riduzione del numero di parametri coinvolti e alla riutilizzabilità dei pesi. In altre parole, la rete può essere addestrata per comprendere meglio la complessità dell'immagine.
Ultimando questa analisi, sono in grado di *estrarre le caratteristiche ad alto livello* (come ad esempio i bordi, il colore, l'orientazione della luce, ecc), eventualmente riducendo gli errori di campionamento.

Infatti, l'assunzione che gli input siano immagini ci permette di codificare alcune proprietà locali nell'architettura. I risultati si sono raggiunti, poi, nel corso del tempo.
Esempi interessanti sono:

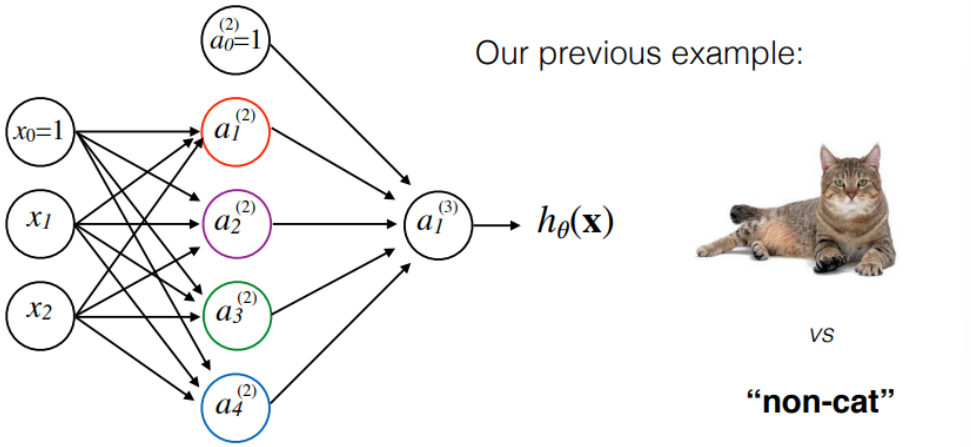
- AlexNet, architettura consistente di otto strati usando funzione ReLU non lineari, GPU multiple di elaborazione, raggruppare output senza sovrapposizione. Si ha avuto, in questo, un bel problema di overfitting
- LeNet 5, uno dei primi algoritmi, composto di 7 strati, in particolare di convoluzione e di subcampionamento, con un filtro in grado di prendere insiemi di dati, filtrarli e farli corrispondere a ciò che interessa.

Neural networks are cool again: ad esempio, nel riconoscimento delle immagini (usando sempre ImageNet per la classificazione degli errori nel corso degli anni), la media dell'errore è andata appiattendosi nel corso del tempo. Se gli errori sono diminuiti, gli strati/layers utilizzati sono aumentati sempre di più, oggetto anche delle vittore dei Turing Award degli ultimi anni.

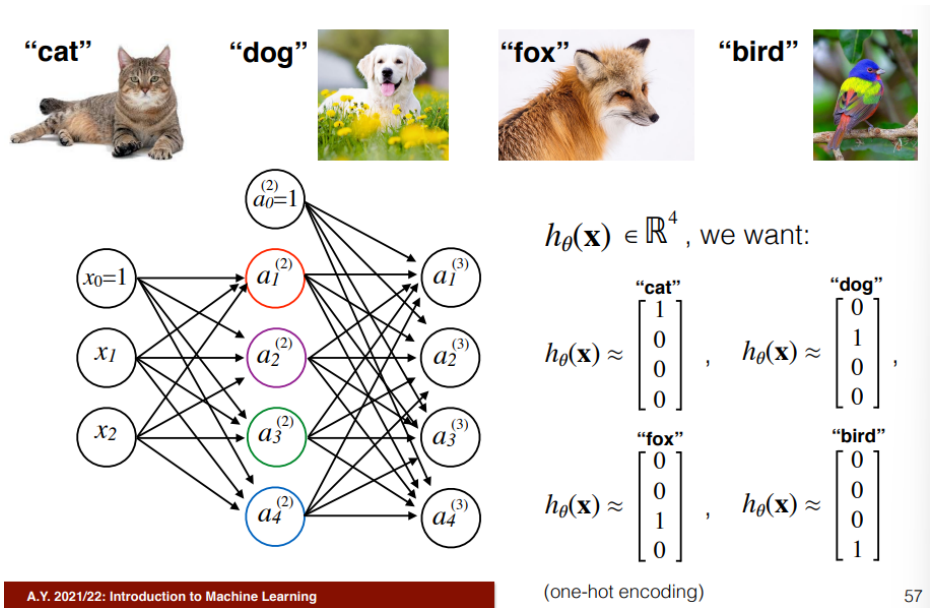
Ripartiamo dalle reti feed-forward, questa volta però usando due strategie per cercare di affrontare problemi di classificazione multiclasse K-way (quindi, in K modi):

- *One-vs-one/Uno-vs-uno*: addestriamo K·(K-1)/2 classificatori binari
  - Ogni classificatore binario viene addestrato per discriminare tra due classi; al momento della predizione applichiamo uno schema di voto
- *One-vs-all/Uno-vs-tutti:* addestriamo un classificatore per classe
  - Ciascun classificatore binario viene addestrato utilizzando i suoi campioni positivi e i campioni appartenenti a tutte le altre classi come negativi.
  - Questa strategia richiede che ogni classificatore produca un punteggio di confidenza con valore reale per la sua decisione; al momento della predizione selezioniamo il classificatore con il punteggio di confidenza più alto.

La classificazione multiclasse non è una classificazione multietichetta (multi-label classification).
Nei problemi di classificazione multietichetta, per ogni istanza devono essere previste più etichette. Il modo in cui effettuiamo la classificazione multiclasse nelle reti neurali è essenzialmente uno contro tutti.
Partendo dall'esempio precedente:



Considerando le classi multiple (intese come animali), avremo che secondo una rete neurale intendiamo classificare ogni classe come vettore come segue:



Approfondimento
(one-hot encoding: assegnare una rappresentazione dinamica, attraverso interi univoci, di nuove variabili binarie che fanno parte delle nostre classificazioni).

Scritto da Gabriel

Partendo dalla funzione di costo della logistic regression, creiamo una funzione di costo che rappresenta una generalizzazione su K-termini e, con essa, andiamo a creare la funzione di costo della rete neurale, che usa anche una funzione nota come *softmax function*, che calcola le probabilità relative per determinare quella finale e dà in output in un vettore le probabilità in distribuzione di una lista di potenziali risultati:

- Logistic Regression cost function:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \cdot \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot \log(1 - h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

Our cost function is going to be a generalization of this, where instead of having just one logistic regression output unit, we have K of them

- Where
  - $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ is the **regularization term**
  - λ is the **regularization factor**

- Neural Network cost function:

$$h_\theta(x) \in \mathbb{R}^K, \text{ where } (h_\theta(x))_k = k^{th} \text{ output}$$

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \cdot \log(h_\theta(x^{(i)})_k) + (1 - y_k^{(i)}) \cdot \log(1 - h_\theta(x^{(i)})_k)\right] +$$

softmax function

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}(\theta_{ji}^{(l)})^2$$

L : number of layers in the network
u_l : number of units in layer l

Qui il termine di sommatoria è generalizzare sulle K unità di uscita della rete neurale calcolando la funzione di costo e sommando tutte le unità di uscita della rete. Inoltre, seguendo la convenzione della regolarizzazione, il termine bias viene saltato dalla penalità di regolarizzazione nella definizione della funzione di costo. Anche se si includesse l'indice 0, in pratica non avrebbe alcun effetto sul processo.

Per apprendere i parametri utili, si può usare il gradient descent, trovando il minimo della funzione di costo appena presentata: $\min_{\theta} J(\theta)$

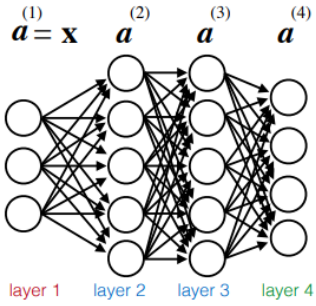
e quindi computando: $\frac{\partial}{\partial\theta_{ij}^{(l)}} J(\theta)$

(ricordando che θ_{ij} connette l'unità *j* allo strato *l* all'unità *i* nello strato *l+1*)
Si esegue dunque la computazione in avanti per calcolare, di volta in volta, le combinazioni lineari di vettori:

- Feed-forward computation:

"forward propagation"

$$z^{(2)} = \theta^{(1)} a^{(1)}$$
$$a^{(2)} = f(z^{(2)}) \quad \text{(add bias: } a_0^{(2)} = 1)$$
$$z^{(3)} = \theta^{(2)} a^{(2)}$$
$$a^{(3)} = f(z^{(3)}) \quad \text{(add bias: } a_0^{(3)} = 1)$$
$$z^{(4)} = \theta^{(3)} a^{(3)}$$
$$a^{(4)} = f(z^{(4)}) = h_\theta(x)$$

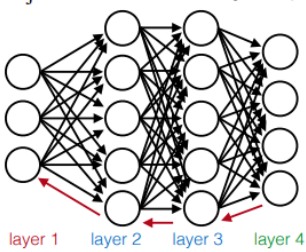


Next, in order to compute the partial derivatives, we are going to use an algorithm called "backpropagation"

Come scritto, per calcolare le derivate parziali derivate parziali, utilizzeremo un algoritmo chiamato "backpropagation". L'algoritmo di backpropagation si basa sull'applicazione ripetuta del calcolo dell'errore utilizzato per la discesa del gradiente, simile alle tecniche di regressione, e poiché viene applicato ripetutamente in ordine inverso partendo dallo strato di uscita e proseguendo verso lo strato di ingresso, viene definito *backpropagation*. Infatti:

• Gradient computation:

$a_j^{(l)}$: activation of unit j in layer l



$\delta^{(2)} \leftarrow \delta^{(3)} \leftarrow \delta^{(4)}$

"backpropagation"

Intuition: we shall compute $\delta_j^{(l)}$

("error" of unit j in layer l)

Therefore, for each output unit:

(e.g. l=4 in this example)

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

Why is that?

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

$$f'(z) = \sigma(z)\,(1 - \sigma(z))$$

e prev

$a^{(3)} .* (1 - a^{(3)})$

Then we compute the error for the previous layers:

$$\delta^{(3)} = (\theta^{(3)})^T \delta^{(4)} .* \boxed{f'(z^{(3)})} \longrightarrow a^{(3)} .* (1 - a^{(3)})$$

$$\delta^{(2)} = (\theta^{(2)})^T \delta^{(3)} .* f'(z^{(2)}) \quad (\text{no } \delta^{(1)})$$

$.*$ *is the element-wise multiplication*

I passi che la backpropagation compie sono quindi, sfruttando il gradiente regolarizzato per minimizzare la funzione di costo:

Training examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Initialization: set $\Delta_{ij}^{(l)} = 0$ (for all i, j, l) ⟶ *"Accumulators" used to compute* $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$

for k=1 to m

set $a^{(1)} = x^{(k)}$

for l=2 to L compute $a^{(l)}$ *"forward propagation"*

compute $\delta^{(L)} = a^{(L)} - y^{(k)}$

compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ *"backpropagation"*

The values $\delta^{(l)}$ are calculated by multiplying the delta values in the next layer with the theta matrix of layer l:

$$\delta^{(l)} = ((\theta^{(l)})^T \delta^{(l+1)}) .* \boxed{a^{(l)} .* (1 - a^{(l)})} \; f'(z^{(l)})$$

Nell'esempio che segue, si ha la matrice Δ, utilizzata come accumulatore per sommare i valori man mano che la retropropagazione procede e infine si ha il calcolo delle derivate parziali.

Training examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Initialization: set $\Delta_{ij}^{(l)} = 0$ (for all i, j, l) ⟶ "Accumulators" used to compute $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$

for k=1 to m

set $a^{(1)} = x^{(k)}$

for l=2 to L compute $a^{(l)}$ "forward propagation"

compute $\delta^{(L)} = a^{(L)} - y^{(k)}$

compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ "backpropagation"

compute gradients $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ ⟶ vectorization $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

compute $D_{ij}^{(l)} := \begin{cases} \frac{1}{m}(\Delta_{ij}^{(l)} + \lambda\, \theta_{ij}^{(l)}) & \text{if } j \neq 0 \\ \frac{1}{m}\Delta_{ij}^{(l)} & \text{if } j = 0 \end{cases}$ where $D_{ij}^{(l)} = \frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta)$

Concludendo questo, il gradient descent applicando la backpropagation.
Si hanno quindi iterazioni con inizializzazione casuale dei parametri, applicando delle propagazioni in avanti ed all'indietro, computazione di pesi e derivate parziali ed aggiornamento sugli stessi pesi:

Training examples $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Initialize all $\theta^{(l)}$ randomly (not to zero!)

Loop/iterate: *each iteration is called an epoch*

Backpropagation

Initialization: set $\Delta_{ij}^{(l)} = 0$ (for all i, j, l)

for k=1 to m

set $a^{(1)} = x^{(k)}$, for l=2 to L compute $a^{(l)}$ "forward propagation"

compute $\delta^{(L)} = a^{(L)} - y^{(k)}$

compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ "backpropagation"

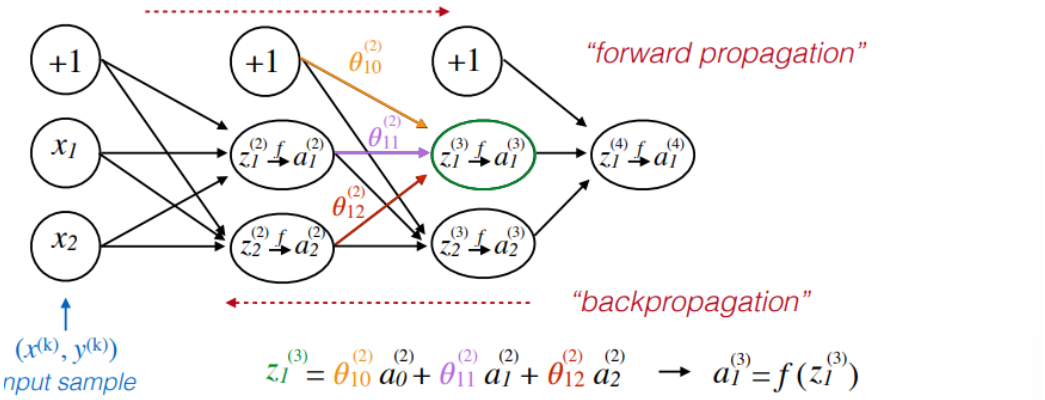
compute gradients $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

compute $D_{ij}^{(l)} := \{ \frac{1}{m}(\Delta_{ij}^{(l)} + \lambda\, \theta_{ij}^{(l)}) \text{ if } j \neq 0 \mid \frac{1}{m}\Delta_{ij}^{(l)} \text{ if } j = 0 \}$

Update weights: $\theta_{ij}^{(l)} = \theta_{ij}^{(l)} - \eta\, D_{ij}^{(l)}$

Ogni unità j è responsabile di una frazione dell'errore δ_j in ognuna delle unità di uscita a cui si connette
- δ_j è suddiviso in base alla forza della connessione tra le unità nascoste e quelle di uscita
- Poi, la "colpa" viene trasmessa a ritroso per fornire i valori di errore per lo strato nascosto

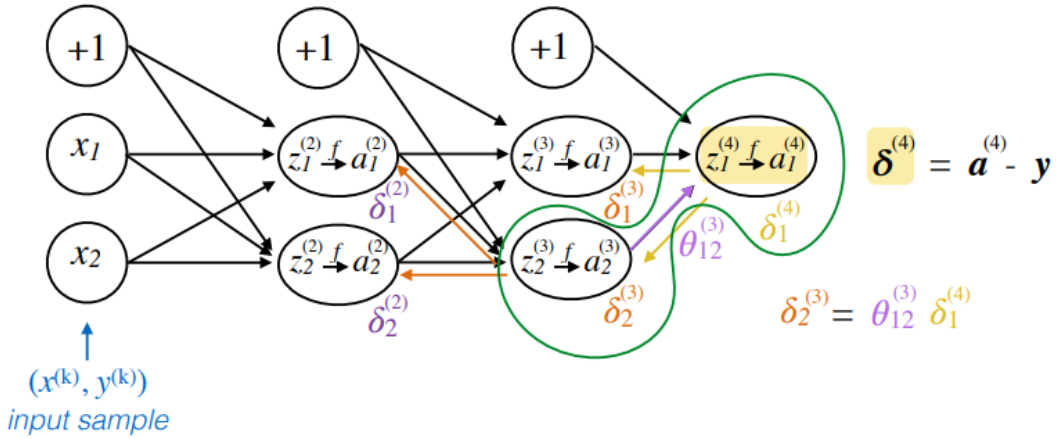
Si ha, però, un'intuizione (backpropagation intuition). Avendo un campione di allenamento da un insieme di esempi di allenamento da cui la rete neurale sta cercando di imparare. Se la funzione di costo viene applicata a questo singolo campione di allenamento impostando $\lambda = 0$ per semplicità, allora può essere ridotta al calcolo di δ_j (errore del costo a_j , ad esempio l'unità *j* nello strato *l*).

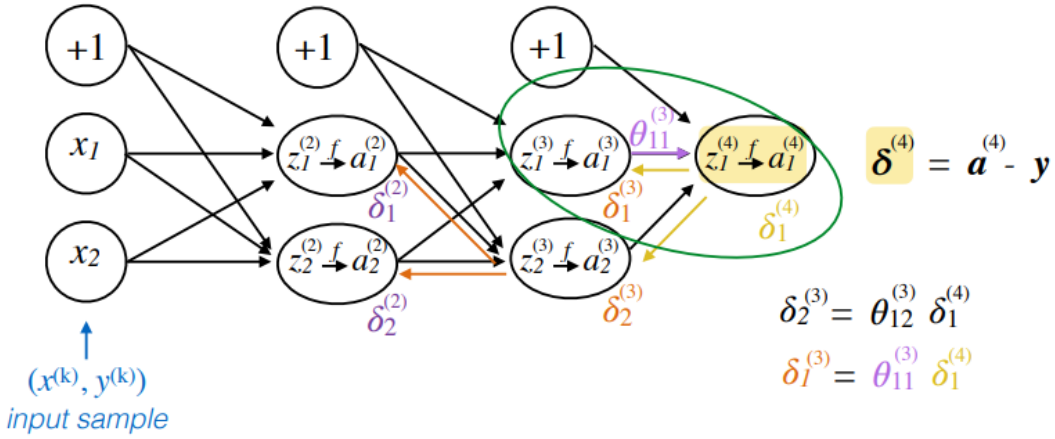


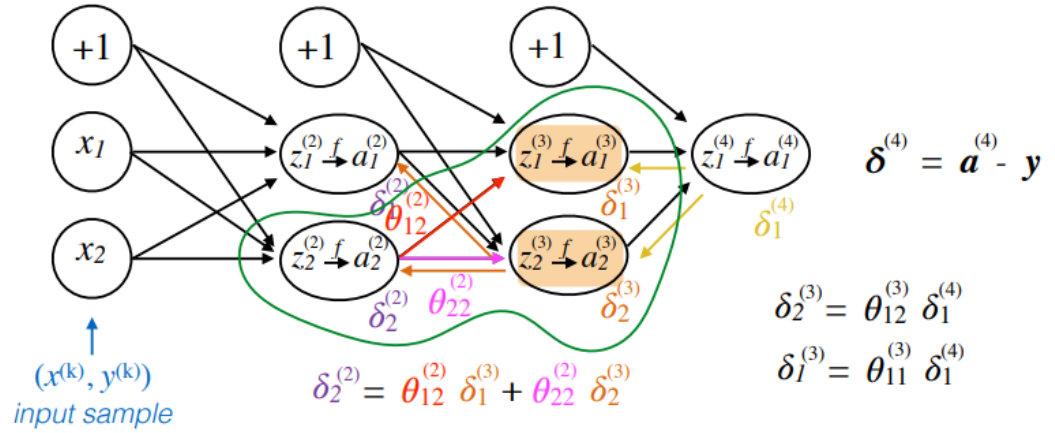
Backprop intuition: we shall compute $\delta_j^{(l)}$ (error of cost for $a_j^{(l)}$, i.e. unit j in layer *l*)

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost(k)}$, $\text{cost(k)} = y^{(k)} \log h_\theta(x^{(k)}) + (1-y^{(k)}) \log(1-h_\theta(x^{(k)}))$ (for j≥0)

Si ha che la rete migliora parecchio la previsione di un output anche per un singolo campione di allenamento. Formalmente, le derivate parziali rivelano matematicamente l'intenzione di modificare la funzione di costo (cambiando i parametri della rete), al fine di influenzare i valori intermedi calcolati, al fine di minimizzare l'output finale nella rete.
Quindi, si propaga il termine errore calcolato nello strato finale usando la propagazione fino agli stati precedenti. Concretamente, è quello che dovrebbero far intuire le slide nei successivi passaggi.







Artificial Neural Networks: implementation issues

Alcuni dettagli implementativi sul loro utilizzo:

- Inizializzazione casuale dei parametri
- Svolgimento (unrolling) dei parametri (se necessario), batch, attivazioni, ecc.
- Debugging: applicare il controllo del gradiente (gradient checking) per convalidare l'implementazione della backpropagation/backprop

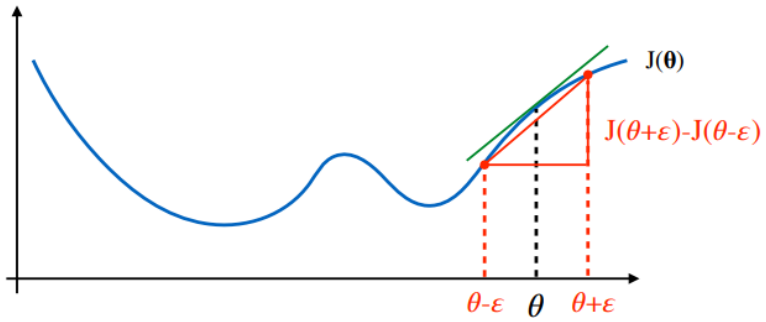
Eseguire il babysitting al processo di apprendimento:

- Controllare la perdita (curve di addestramento e di validazione).
- Tracciare anche l'accuratezza dell'addestramento e della validazione.
    - Attenzione: un enorme divario tra l'accuratezza di addestramento e quella di validazione indica che il modello è in overfitting

Si consideri *l'inizializzazione casuale.* L'inizializzazione di tutti i pesi a zero non funziona con le reti neurali Quando si esegue la backpropagation, tutti i nodi si aggiorneranno allo stesso valore ripetutamente.

- "Rottura della simmetria": Una inizializzazione casuale dei pesi risolve il problema.
  Ad esempio, inizializzare tutti i θ_{ij} ad un valore casuale tra [-ε, ε].
  In questo modo, ci si assicura che la backpropagation funzioni come voluto.

Avviene una stima numerica del gradiente (gradient checking):



$$\frac{d}{d\theta} J(\boldsymbol{\theta}) \approx \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon} \qquad \text{e.g. } \varepsilon=10^{-4}$$

- Perché abbiamo bisogno del controllo del gradiente?

Il backprop come algoritmo ha molti dettagli e può essere un po' complicato da implementare. Una proprietà spiacevole è che ci sono molti modi per avere bug sottili in backprop. In questo modo, se lo si esegue con la discesa del gradiente o con qualche altro algoritmo di ottimizzazione, potrebbe sembrare che stia funzionando. E la funzione di costo potrebbe finire per diminuire a ogni iterazione della discesa del gradiente. Ma questo potrebbe risultare vero anche se potrebbe esserci qualche bug nell'implementazione del backprop. In questo modo sembra che la funzione di costo stia diminuendo, ma si potrebbe avere una rete neurale che ha un livello di errore più alto di quello che avreste con un'implementazione priva di bug. E potreste non sapere che c'era questo sottile bug che vi dava prestazioni peggiori. Quindi, cosa possiamo fare per risolvere questo problema? Esiste un'idea chiamata controllo del gradiente che elimina quasi tutti questi problemi.

- Che cos'è il controllo del gradiente?

Descriviamo un metodo per verificare numericamente le derivate calcolate dal codice per assicurarsi che l'implementazione sia corretta. L'esecuzione della procedura di controllo delle derivate aumenta significativamente la fiducia nella correttezza del codice.
In breve, il controllo dei gradienti è una sorta di debug dell'algoritmo di backprop. Il Gradient Checking esegue fondamentalmente la procedura di controllo delle derivate.

Quindi:

- Prendendo la matrice θ, approssimiamo la derivata come segue:

$$\frac{\partial}{\partial\theta_j} J(\boldsymbol{\theta}) \approx \frac{J(\theta_1,\ldots,\theta_j+\varepsilon,\ldots,\theta_n) - J(\theta_1,\ldots,\theta_j-\varepsilon,\ldots,\theta_n)}{2\varepsilon}$$

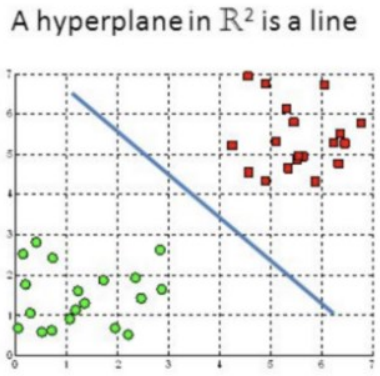
- Confrontiamo se dopo aver calcolato la matrice Δ, i due gradienti siano vicini tra di loro
- Una volta verificata la correttezza, rimuoviamo il gradient checking

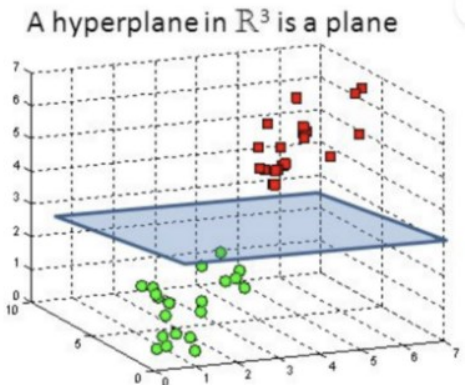
Support Vector Machines: loss function

SVM è un algoritmo molto potente e popolare.

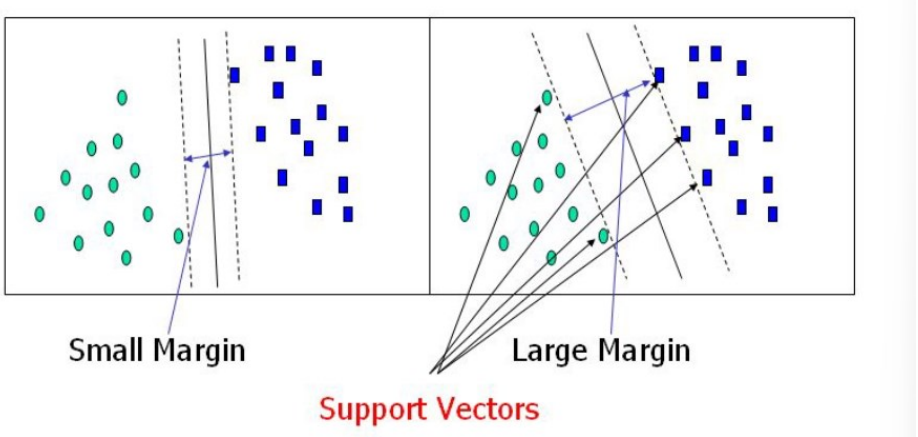
- È un algoritmo di apprendimento supervisionato, solitamente utilizzato per la classificazione (ma può essere utilizzato anche per la regressione).
- È un classificatore non probabilistico (binario), anche se esistono metodi come il Platt scaling un'interpretazione probabilistica dell'output di SVM
- Si tratta di un modello di classificazione lineare, ma SVM può eseguire in modo efficiente classificazione non lineare utilizzando il *kernel trick*

L'obiettivo dell'algoritmo della SVM è trovare un iperpiano (spazio a N dimensioni in cui le coordinate soddisfano un equazione lineare) in uno spazio N-dimensionale (N - il numero di caratteristiche) che classifichi distintamente i punti dati.
Per separare le due classi di punti dati, si possono scegliere molti iperpiani. Il nostro obiettivo è trovare un piano che abbia il massimo margine, cioè la massima distanza tra i punti dati di entrambe le classi. La massimizzazione del margine di distanza fornisce un rinforzo in modo che i punti dati futuri possano essere classificati con maggiore sicurezza.
Esempi di iperpiano:





Gli iperpiani sono confini decisionali che aiutano a classificare i punti di dati. I punti di dati che cadono su entrambi i lati dell'iperpiano possono essere attribuiti a classi diverse. Inoltre, la dimensione dell'iperpiano dipende dal numero di caratteristiche. Se il numero di caratteristiche in ingresso è 2, l'iperpiano è solo una linea. Se il numero di caratteristiche in ingresso è 3, l'iperpiano diventa un piano bidimensionale. Diventa difficile da immaginare quando il numero di caratteristiche è superiore a 3.



I vettori di supporto/support vectors sono punti di dati più vicini all'iperpiano e influenzano la posizione e l'orientamento dell'iperpiano. Utilizzando questi vettori di supporto, si massimizza il margine del classificatore. L'eliminazione dei vettori di supporto modifica la posizione dell'iperpiano. Questi sono i punti che ci aiutano a costruire la nostra SVM.

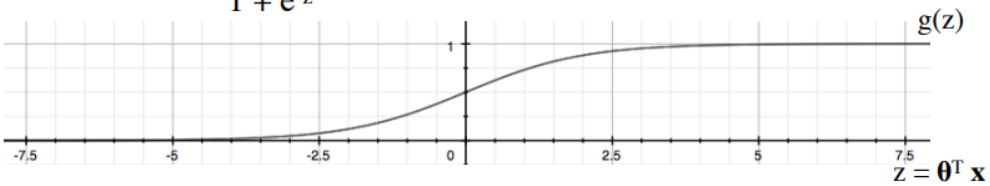
Scritto da Gabriel

Nella logistic regression, prendiamo l'output della funzione lineare e schiacciamo il valore all'interno dell'intervallo [0,1] utilizzando la funzione sigmoide. Se il valore schiacciato è maggiore di un valore di soglia (0,5), gli assegniamo l'etichetta 1, altrimenti gli assegniamo l'etichetta 0. In SVM, prendiamo l'uscita della funzione lineare e se questa è maggiore di 1, la identifichiamo con una classe e se l'uscita è -1, la identifichiamo con un'altra classe. Poiché in SVM i valori di soglia vengono modificati in 1 e -1, si ottiene un intervallo di valori di rinforzo ([-1,1]) che funge da margine.

- Hypothesis representation:

$$h_\theta(\mathbf{x}) = g(\boldsymbol{\theta}^T \mathbf{x}) = \frac{1}{1 + e^{-\boldsymbol{\theta}^T x}}$$

where $g(z) = \frac{1}{1 + e^{-z}}$ (*Sigmoid* or *Logistic* function)



  - Interpretation (i.e. what we'd like logistic regression to do):

    If $y = 1$, we want $h_\theta(\mathbf{x}) \approx 1$, $\boldsymbol{\theta}^T \mathbf{x} \gg 0$

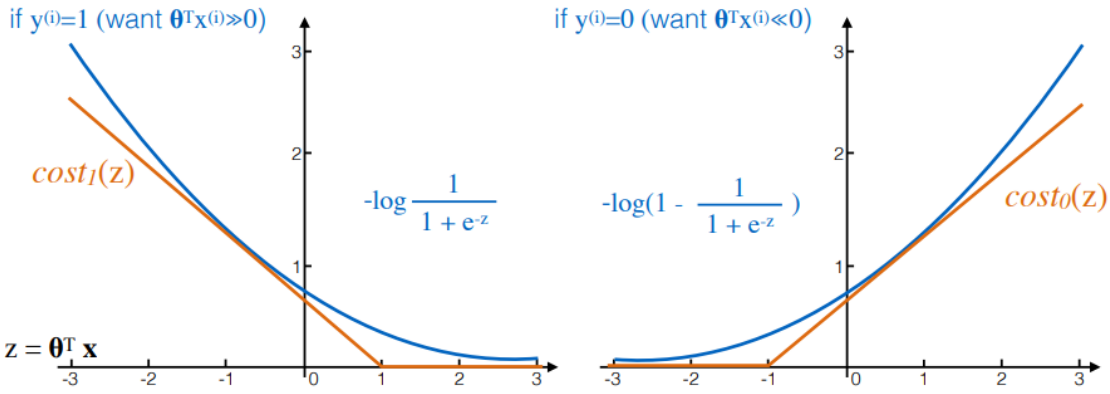
    If $y = 0$, we want $h_\theta(\mathbf{x}) \approx 0$, $\boldsymbol{\theta}^T \mathbf{x} \ll 0$

Nell'algoritmo SVM si cerca di massimizzare il margine tra i punti dati e l'iperpiano. La funzione di perdita che aiuta a massimizzare il margine è la perdita per cerniera (hinge loss).

- Loss function: $J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} cost(h_\theta(x^{(i)}), y^{(i)})$ *where:*

  - If we take the definition for $h_\theta(\mathbf{x})$ and plug it in, we get:

$$cost(\cdot) = -y^{(i)} \cdot \log\left(\frac{1}{1 + e^{-\boldsymbol{\theta}^T x^{(i)}}}\right) - (1 - y^{(i)}) \cdot \log\left(1 - \frac{1}{1 + e^{-\boldsymbol{\theta}^T x^{(i)}}}\right)$$



Il costo è 0 se il valore previsto e quello effettivo hanno lo stesso segno. In caso contrario, si calcola il valore di perdita. Aggiungiamo anche un parametro di regolarizzazione alla funzione di costo. L'obiettivo del parametro di regolarizzazione è quello di bilanciare la massimizzazione del margine e la perdita.

Il problema che affrontiamo è di *ottimizzazione convessa*, con una funzione obiettivo di ottimizzazione convessa e una serie di vincoli che definiscono un insieme convesso come regione fattibile (un esempio di funzione convessa è la parabola, con la curva rivolta verso l'alto). L'insieme convesso è un insieme di punti in cui una linea che unisce due punti qualsiasi giace interamente all'interno dell'insieme.
Si parte da:

- Support Vector Machine:

$$\min_{\theta} \frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} \cdot cost_1(\theta^T x^{(i)}) + (1-y^{(i)}) \cdot cost_0(\theta^T x^{(i)})\right] + \frac{1}{2m}\sum_{j=1}^{n} \theta_j^2$$

$$C\,A + B \quad , \quad C = \frac{1}{\lambda}$$

$$\Rightarrow \min_{\theta} C\sum_{i=1}^{m}\left[y^{(i)} \cdot (cost_1(\theta^T x^{(i)})) + (1-y^{(i)}) \cdot (cost_0(\theta^T x^{(i)}))\right] + \frac{1}{2}\sum_{j=1}^{n} \theta_j^2$$

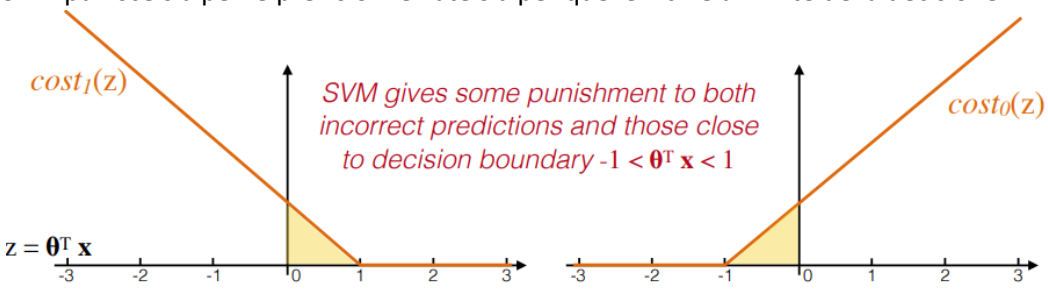
Alla fine, l'algoritmo SVM arriverà a risolvere:

$$\min \frac{1}{2}\|w\|^2$$
$$s.t.\ y_i(w \cdot x_i + b) \geq 1,\ \forall x_i$$

Quindi, l'interpretazione di quello che vorremo facesse SVM sarebbe:

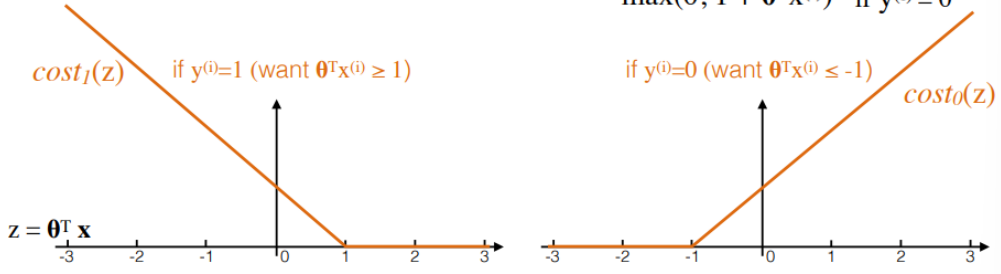
- Se $y^{(i)} = 1$, vorremmo $\theta^T x^{(i)} >= 1$ (non solo >=0)
- Se $y^{(i)} = 1$, vorremmo $\theta^T x^{(i)} < 1$ (non solo < 0)

SVM punisce sia per le previsioni errate sia per quelle vicine al limite della decisione $-1 < \theta^T x < 1$



La funzione di perdita di SVM sarebbe quindi:

- Let's write the SVM's loss: $J(\theta) = \sum_{i=1}^{m} cost(h_\theta(x^{(i)}), y^{(i)})$
  - Hinge loss: $cost(h_\theta(x^{(i)}), y^{(i)}) = \begin{cases} \max(0, 1 - \theta^T x^{(i)}) & \text{if } y^{(i)} = 1 \\ \max(0, 1 + \theta^T x^{(i)}) & \text{if } y^{(i)} = 0 \end{cases}$



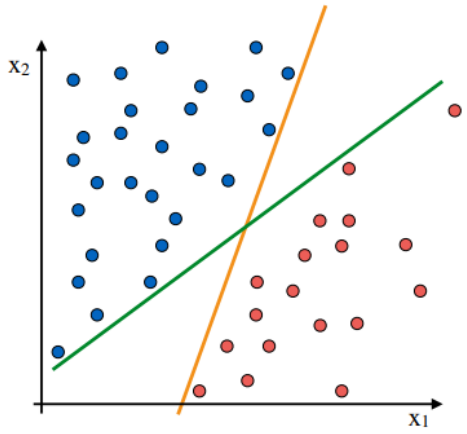
Gli obiettivi principali, prima di dare un'intuizione di SVM, sono:

1) SVM massimizza il margine apprendendo un confine decisionale/superficie di decisione/ iperpiano di separazione adeguato.

Scritto da Gabriel

2) SVM massimizza il margine geometrico apprendendo un confine decisionale/superficie di decisione/ iperpiano di separazione adeguato.

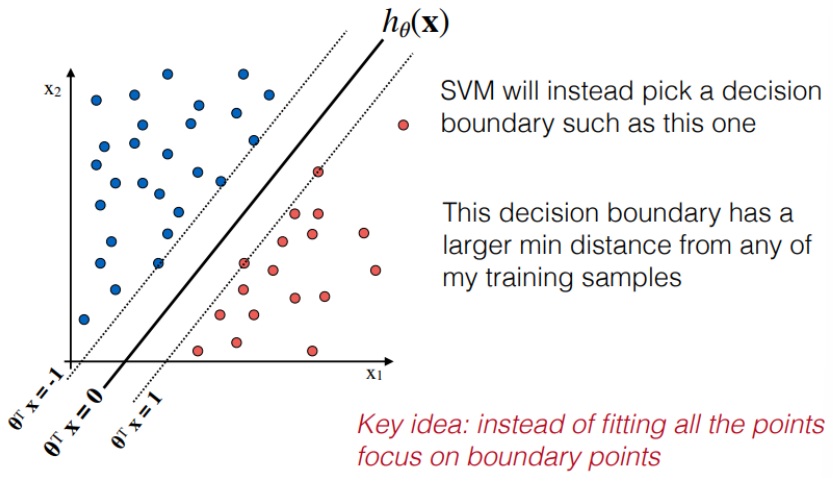
Partendo dalla figura seguente (caso linearmente separabile):



Ciò significa che tra tutti gli iperpiani possibili (ogni iperpiano ha un margine geometrico rispetto al punto più vicino che è il minore di tutti gli altri margini geometrici definiti rispetto a tutti gli altri punti), SVM sceglie l'iperpiano che ha il margine geometrico massimo.

Quindi, cercherà di concentrare il proprio campionamento in un intervallo relativo ai punti del confine decisionale.

La retta in mezzo è l'iperpiano migliore, tale che da questo punto di vista, essa rappresenti geometricamente il più piccolo di tutti i margini geometrici condivisi da almeno un paio di vettori di esempio/campionamento.
Tali punti sono proprio i *support vectors*.



Pertanto, il problema di ottimizzazione definito sopra è equivalente al problema della massimizzazione del valore del *margine* (non dei valori del margine geometrico/funzionale). Il margine è definito come la *distanza tra due iperpiani*, ciascuno dei quali è parallelo all'iperpiano di separazione e passa attraverso i vettori di supporto di ciascuna classe (cioè, un iperpiano passa attraverso i vettori di supporto della classe +ve, mentre l'altro iperpiano passa attraverso i vettori di supporto della classe -ve, ed entrambi sono paralleli agli iperpiani di separazione).

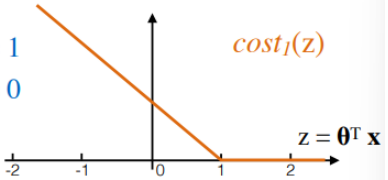
• Optimization objective:

$$\min_{\boldsymbol{\theta}} C\left[\sum_{i=1}^{m}\left[y^{(i)}\cdot(cost_1(\boldsymbol{\theta}^T x^{(i)}))+(1-y^{(i)})\cdot(cost_0(\boldsymbol{\theta}^T x^{(i)}))\right]\right]+\frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

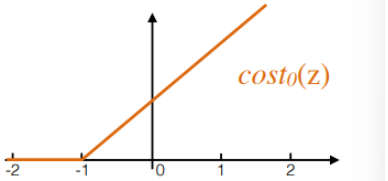
e.g. $C \approx 10^5$ $\qquad\qquad \approx 0$

$$\min_{\boldsymbol{\theta}} \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 \quad \text{s.t. } \boldsymbol{\theta}^T \mathbf{x}^{(i)} \geq 1 \text{ if } y^{(i)}=1,\ \boldsymbol{\theta}^T \mathbf{x}^{(i)} \leq -1 \text{ if } y^{(i)}=0$$

Whenever $y^{(i)}=1$: $\boldsymbol{\theta}^T x^{(i)} \geq 1$



Whenever $y^{(i)}=0$: $\boldsymbol{\theta}^T x^{(i)} \leq -1$



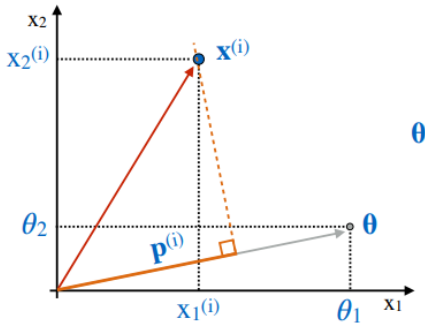
Prendendo un esempio concreto, vediamo che ad esempio, sulla funzione obiettivo, si avrà una tendenza a 0 del costo, vedendo quindi che SVM prenderà uno dei due valori nel suo confine di decisione (decision boundary):

Sulla base dell'obiettivo di minimizzazione, abbiamo che si ha un calcolo sulla base della porzione di spazio considerato, che aiuta a minimizzare la funzione di costo ottimizzandola, tale che da un punto di vista grafico questa sia esprimibile come proiezione del vettore di analisi $x^{(i)}$ sul vettore θ, quindi $p^{(i)}$.

- Optimization objective:

$$\min_{\theta} \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}(\theta_1^2+\theta_2^2) = \frac{1}{2}\sqrt{(\theta_1^2+\theta_2^2)} = \frac{1}{2}\|\theta\|^2$$

$$\text{s.t. } \begin{cases}\theta^T x^{(i)} \geq 1 & \text{if } y^{(i)}=1\\ \theta^T x^{(i)} \leq -1 & \text{if } y^{(i)}=0\end{cases} \Rightarrow \begin{cases}p^{(i)}\|\theta\| \geq 1 & \text{if } y^{(i)}=1\\ p^{(i)}\|\theta\| \leq -1 & \text{if } y^{(i)}=0\end{cases}$$



$$\theta^T x^{(i)} = p^{(i)}\|\theta\|$$

Thus we can write our optimization objective w.r.t. $p^{(i)}$

Ciò significa che la proiezione segue l'andamento del vettore, in particolare:

- Se la proiezione è piccola, il vettore θ è grande
- Se la proiezione è grande, il vettore θ è piccolo

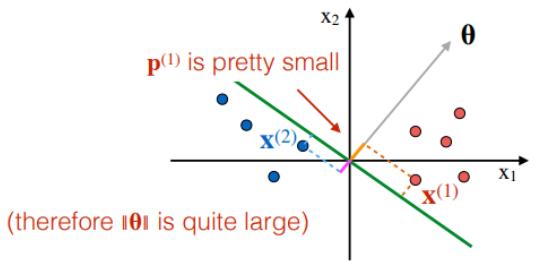
- Optimization objective:

$$\min_{\theta} \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\|\theta\|^2$$

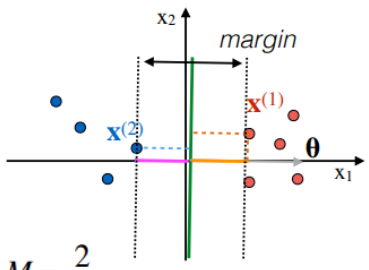
$$\text{s.t. } \begin{cases}p^{(i)}\|\theta\| \geq 1 & \text{if } y^{(i)}=1\\ p^{(i)}\|\theta\| \leq -1 & \text{if } y^{(i)}=0\end{cases}$$

Where $p^{(i)}$ *is the projection of* $x^{(i)}$ *onto the vector* θ

- An example:



SVM hypothesis



$$M = \frac{2}{\|\theta\|}$$

Sulla base del vettore di proiezione, è possibile capire il margine su cui trovare i vettori di supporto come si vede qui a sx.

Infatti, geometricamente, il margine è proprio
Conseguentemente, l'iperpiano a margine massimo è definito da quelle $x^{(i)}$ che si trovano più vicine ad esso (vettori di supporto, appunto).

Da un punto di vista di SVM si usa una notazione leggermente diversa sulla formula, che contribuisce a semplificare il calcolo considerato e pure il grafico come segue (s.t sta per “such that”, quindi “affinché”):

- Optimization objective:

$$\min_{\theta} \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\|\theta\|^2$$

s.t. $p^{(i)}\|\theta\| \geq 1$ if $y^{(i)} = 1$ *In SVM usually we use a different notation*

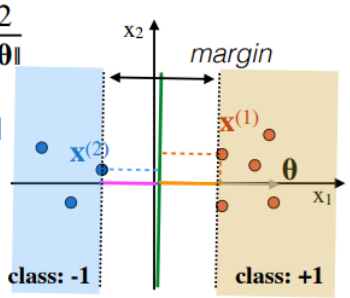
$p^{(i)}\|\theta\| \leq -1$ if $y^{(i)} = -1$

$$\min_{\theta} \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\|\theta\|^2$$

s.t. $y^{(i)}(p^{(i)}\|\theta\|) \geq 1$

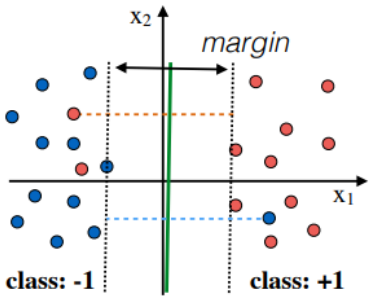
- Large-margin classification:
  - Geometrically, the margin is: $M = \frac{2}{\|\theta\|}$
  - A key consequence is that the max-margin hyperplane is defined by those $x^{(i)}$ that lie nearest to it *(support vectors)*

SVM hypothesis



Quanto visto fino ad ora è noto come *hard-margin* (margine rigido), cioè i dati sono linearmente separabili. Se i dati sono linearmente separabili, si opta per un margine rigido. Tuttavia, se questo non è il caso, non sarà possibile farlo. In presenza di punti dati che rendono impossibile trovare un classificatore lineare, dovremo essere più indulgenti e lasciare che alcuni punti dati vengano classificati in modo errato. In questo caso, un SVM a margine morbido è appropriato.
A volte, i dati sono linearmente separabili, ma il margine è così piccolo che il modello diventa incline all'overfitting o troppo sensibile agli outlier. Anche in questo caso, si può optare per un margine più ampio utilizzando SVM a margine morbido, per aiutare il modello a generalizzarsi meglio.

Se ci fossero degli esempi dal lato sbagliato, si introdurrebbe $\xi^{(i)}$ (cosiddetta *slack variable*, cio è una variabile che viene aggiunta a un vincolo di disuguaglianza per trasformarlo in un vincolo di uguaglianza). Come si vede dall’immagine, la serie su $\xi^{(i)}$ limita il numero di errori di allenamento (training errors). La variabile ha quindi uno scopo di allentamento sul margine, tale che si abbia proprio una *soft-margin extension*:



α trades off training error vs model complexity

- Introduce the slack variable $\xi^{(i)}$
- New optimization objective:

$$\min_{\theta} \frac{1}{2}\|\theta\|^2 + \alpha\sum_{i=1}^{m}\xi^{(i)}$$

s.t. $\xi^{(i)} \geq 0$, $\forall i$: $y^{(i)}(p^{(i)}\|\theta\|) \geq 1-\xi^{(i)}$

- $\sum_{i=1}^{m}\xi^{(i)}$ bounds num. of training errors
- This is called *soft-margin* extension

Tale estensione porta alla definizione di *hinge loss* che abbiamo visto prima, definita come una funzione convessa ottimizzata con l'uso di gradient descent.

- Hinge loss: $\ell(h_\theta(\mathbf{x}^{(i)}), y^{(i)}) = \max(0, 1 - y^{(i)}(\boldsymbol{\theta}^T \mathbf{x}^{(i)}))$

  where $\boldsymbol{\theta}^T \mathbf{x}^{(i)} = \mathbf{p}^{(i)} \|\boldsymbol{\theta}\|$, *i.e.* $\mathbf{p}^{(i)}$ *is the projection of* $\mathbf{x}^{(i)}$ *onto the vector* $\boldsymbol{\theta}$

L'interpretazione è come segue:

- Quando $y^{(i)}$ e $p^{(i)}$ hanno stesso segno e $|y^{(i)}| >= 1$, *l(.)=0*
- Quando hanno segno opposto, *l* aumenta linearmente con $y^{(i)}$
- Similmente, se $|y^{(i)}| < 1$ (ha lo stesso segno: previsione corretta ma non con un margine sufficiente) *l* aumenta linearmente con $y^{(i)}$

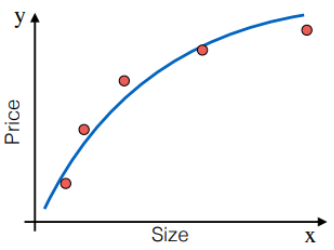
L'estensione del soft-margin è ottima, ma SVM è ancora un modello di classificazione lineare.
modello di classe. Tuttavia, SVM può eseguire in modo efficiente la classificazione non lineare utilizzando il cosiddetto "*trucco del kernel" (kernel trick)*.
Abbiamo visto come le trasformazioni dimensionali più elevate possano permetterci di separare i dati per fare previsioni di classificazione. Sembra che per addestrare un classificatore a vettori di supporto e ottimizzare la nostra funzione obiettivo, dovremmo eseguire operazioni con i vettori a più alta dimensione nello spazio delle caratteristiche trasformato. Nelle applicazioni reali, le caratteristiche dei dati potrebbero essere molte e l'applicazione di trasformazioni che coinvolgono molte combinazioni polinomiali di queste caratteristiche porterebbe a costi computazionali estremamente elevati e poco pratici.

Il kernel trick fornisce una soluzione a questo problema. Il "trucco" consiste nel fatto che i metodi kernel rappresentano i dati solo attraverso un insieme di confronti di similarità a coppie tra le osservazioni originali dei dati x (con le coordinate originali nello spazio dimensionale inferiore), invece di applicare esplicitamente le trasformazioni ϕ(x) e rappresentare i dati con queste coordinate trasformate nello spazio delle caratteristiche dimensionale superiore.

Nei metodi kernel, l'insieme di dati X è rappresentato da una matrice kernel n x n di confronti di similarità a coppie, in cui le voci (i, j) sono definite dalla funzione kernel: k(xi, xj). Questa funzione kernel ha una particolare proprietà matematica. La funzione kernel agisce come un prodotto di punti modificato.
La nostra funzione kernel accetta input nello spazio dimensionale inferiore originale e restituisce il prodotto di punti dei vettori trasformati nello spazio dimensionale superiore. Esistono anche teoremi che garantiscono l'esistenza di tali funzioni kernel sotto determinate condizioni.

Il vantaggio finale del trucco del kernel è che la funzione obiettivo che stiamo ottimizzando per adattarci al confine decisionale di dimensioni più elevate include solo il prodotto del punto dei vettori di caratteristiche trasformati. Pertanto, possiamo semplicemente sostituire questi termini del prodotto del punto con la funzione kernel, senza utilizzare ϕ(x).
Ricordiamo che i nostri dati sono separabili linearmente solo come i vettori ϕ(x) nello spazio dimensionale superiore e che stiamo trovando l'iperpiano di separazione ottimale in questo spazio dimensionale superiore senza dover calcolare o in realtà nemmeno conoscere ϕ(x).

Per "estendere" modelli lineari, introduciamo la *polinomial regression* per attuare le *feature trasformations* (che sono una trasformazione matematica in cui si applica una formula matematica a una particolare colonna (caratteristica) e si trasformano i valori utili per la nostra ulteriore analisi, in questo caso un polinomio).



Features and Polynomial Regression

$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$

How to get this $h_\theta(x)$ from our linear regression model $\boldsymbol{\theta}^T \mathbf{x}$?

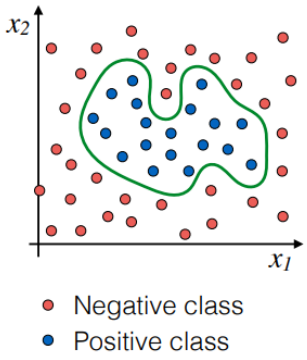
$$\Rightarrow h_\theta(x) = \begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\end{bmatrix}^T \begin{bmatrix}x_0\\ x_1\\ x_2\end{bmatrix} = \begin{bmatrix}\theta_0\\ \theta_1\\ \theta_2\end{bmatrix}^T \begin{bmatrix}1\\ x\\ x^2\end{bmatrix}$$

x_1: size of house

x_2: (size of house)^2

Scritto da Gabriel

Prediciamo quindi *y=1* se la nostra equazione polinomiale è >= 0, rispetto alla funzione indicatrice come segue. Il punto è come scegliere le features *f*.



Predict $y = 1$ if:

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \ldots \geq 0$$

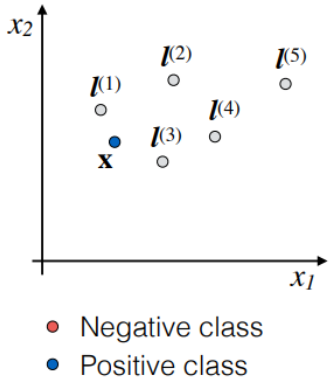
$$h_\theta(x) = \begin{cases} 1 & \text{if } \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \theta_4 f_4 + \theta_5 f_5 + \ldots$

where $f_1 = x_1, f_2 = x_2, f_3 = x_1 x_2, f_4 = x_1^2, f_5 = x_2^2, \ldots$

How to choose features $f_1, f_2, \ldots$?

Dato *x*, si computano le features *f* in base alla prossimità ai punti di riferimento/landmarks $l^{(i)}$ tale da poter approssimare ogni caratteristica con i punti vicini:



Given $\mathbf{x}$, compute features $\mathbf{f}$ depending on proximity to landmarks $\boldsymbol{l}^{(1)}, \boldsymbol{l}^{(2)}, \ldots$

$$f_1 = \text{similarity}(\mathbf{x}, \boldsymbol{l}^{(1)})$$
$$f_2 = \text{similarity}(\mathbf{x}, \boldsymbol{l}^{(2)})$$
$$\vdots$$

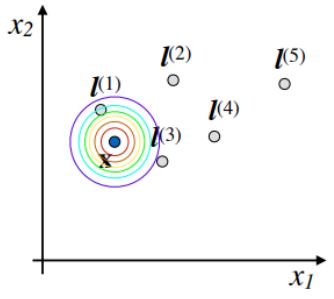
Predict $y = 1$ if:

$\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \theta_4 f_4 + \theta_5 f_5 + \ldots \geq 0$

where $f_1 = x_1, f_2 = x_2, f_3 = x_1 x_2, f_4 = x_1^2, f_5 = x_2^2, \ldots$

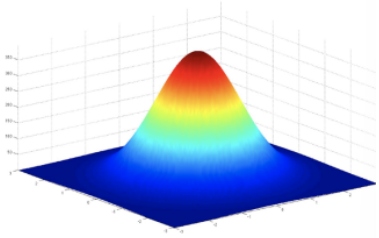
A questo punto:

- se *x* è circa uguale a $l^{(i)}$, allora f_1 assume come valore il termine generale della distribuzione gaussiana
- se *x* è lontano da $l^{(i)}$, allora $f_1 = 0$



Given $\mathbf{x}$, compute features $\mathbf{f}$ depending on proximity to landmarks $\boldsymbol{l}^{(1)}, \boldsymbol{l}^{(2)}, \ldots$

$$f_1 = k(\mathbf{x}, \boldsymbol{l}^{(1)})$$
$$f_2 = k(\mathbf{x}, \boldsymbol{l}^{(2)})$$
$$f_3 = k(\mathbf{x}, \boldsymbol{l}^{(3)})$$
$$f_4 = k(\mathbf{x}, \boldsymbol{l}^{(4)})$$
$$f_5 = k(\mathbf{x}, \boldsymbol{l}^{(5)})$$



if $\mathbf{x} \approx \boldsymbol{l}^{(i)}$: $f_1 = \exp(-0^2/(2\sigma^2)) \approx 1$

if $\mathbf{x}$ is far from $\boldsymbol{l}^{(i)}$: $f_1 \approx 0$

e.g. $k(\mathbf{x}, \boldsymbol{l}^{(i)}) = \exp\left(-\frac{\|\mathbf{x} - \boldsymbol{l}^{(i)}\|^2}{2\sigma^2}\right)$ *Gaussian Kernel*

Quanto dato è il popolare *RBF Kernel (Radius Basis Function)*, che è la forma più generalizzata di kernelizzazione ed è uno dei kernel più utilizzati per la sua somiglianza con la distribuzione gaussiana. La funzione kernel RBF per due punti X_1 e X_2 calcola la somiglianza o quanto sono vicini tra loro. Questo kernel può essere rappresentato matematicamente come segue (slide a sx e indicazione generica a dx):

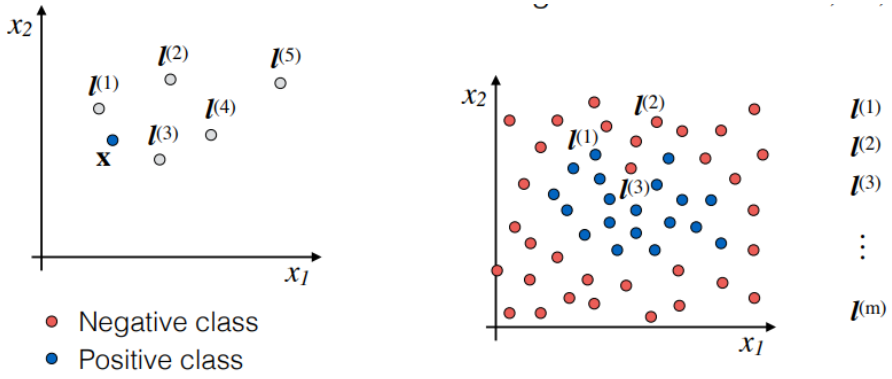
$$K(X_1, X_2) = exp(-\frac{||X_1 - X_2||^2}{2\sigma^2})$$

$$k(\mathbf{x}, \boldsymbol{l}^{(i)}) = \exp(-\gamma \|\mathbf{x} - \boldsymbol{l}^{(i)}\|^2)$$

where,

1. 'σ' is the variance and our hyperparameter
2. $||X_1 - X_2||$ is the Euclidean (L_2-norm) Distance between two points X_1 and X_2

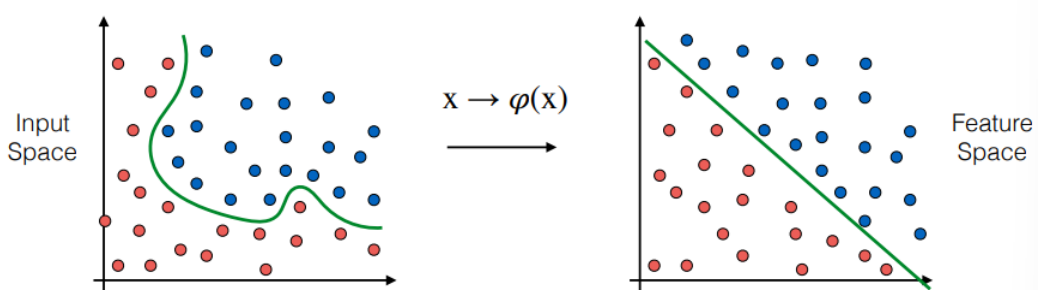
Ora, come trovare gli $l^{(i)}$?



- Obiettivo: trasformare/mappare i dati grezzi in vettori di caratteristiche

Invece di utilizzare mappe di caratteristiche esplicite specificate dall'utente, ci si affida a una funzione di somiglianza (kernel) su coppie di punti dati.

Un kernel è una funzione ϕ: R^n → R^m che mappa x → ϕ(x). Quindi, data questa mappatura, cerchiamo di trovare una decisione lineare nello spazio delle caratteristiche.



Come già accennato prima dalla definizione di kernel, si parla di "generalized dot product" (prodotto scalare generalizzato di punti), per cui grazie ai kernel è possibile calcolare i prodotti di punti in uno spazio di caratteristiche senza nemmeno sapere quale sia questo spazio e quale sia il ϕ .

$\mathbf{x}, \mathbf{y} \in \mathbb{R}^n \qquad k(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x})^T \varphi(\mathbf{y})$ corresponds to a dot product in $\mathbb{R}^n$

Un esempio (kernel polinomiale):

$\mathbf{x}, \mathbf{y} \in \mathbb{R}^n \qquad k(\mathbf{x}, \mathbf{y}) = \varphi(\mathbf{x})^T \varphi(\mathbf{y})$ corresponds to a dot product in $\mathbb{R}^n$

$k(\mathbf{x}, \mathbf{y}) = (1 + \mathbf{x}^T \mathbf{y})^2$ assuming $\mathbf{x}, \mathbf{y} \in \mathbb{R}^2$, i.e. $\mathbf{x} = (x_1, x_2), \mathbf{y} = (y_1, y_2)$

$k(\mathbf{x}, \mathbf{y}) = (1 + x_1 y_1 + x_2 y_2)^2 = 1 + x_1^2 y_1^2 + x_2^2 y_2^2 + 2x_1 y_1 + 2x_2 y_2 + 2x_1 x_2 y_1 y_2$

Note that this is nothing else but a dot product between vectors:

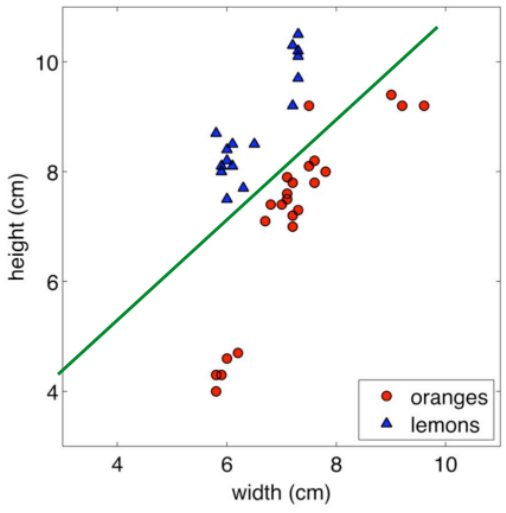
$\varphi(\mathbf{x}) = (1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2)$, $\varphi(\mathbf{y}) = (1, y_1^2, y_2^2, \sqrt{2}y_1, \sqrt{2}y_2, \sqrt{2}y_1 y_2)$

Non-parametric Models, kNN

I modelli non parametrici/non-parametric models sono modelli statistici che spesso non sono conformi a una distribuzione normale, in quanto si basano su dati *continui*, piuttosto che su valori *discreti*. Le statistiche non parametriche trattano spesso numeri ordinali o dati che non hanno un valore fisso come un numero discreto. Il termine non parametrico non significa che il valore non abbia parametri intrinseci, ma piuttosto che i parametri sono flessibili e possono variare. Si può ricorrere alla statistica non parametrica quando si ha a che fare con dati classificati, in cui parte del valore delle variabili è l'ordine in cui sono organizzate.

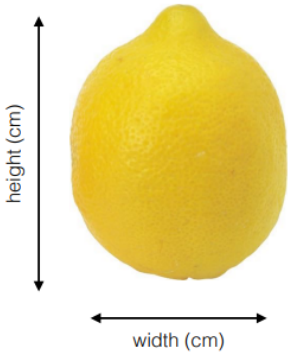
Gli algoritmi di apprendimento automatico non parametrici cercano di formulare ipotesi sui dati in base ai modelli osservati da istanze simili. Ad esempio, un popolare algoritmo di apprendimento automatico non parametrico è l'algoritmo K-Nearest Neighbor, che esamina modelli di formazione simili per le nuove istanze. L'unica ipotesi che fa sull'insieme dei dati è che i modelli di addestramento più simili hanno maggiori probabilità di avere un risultato simile. Sebbene gli algoritmi di apprendimento automatico non parametrici siano spesso più lenti e richiedano grandi quantità di dati, sono piuttosto flessibili in quanto riducono al minimo le ipotesi che fanno sui dati. Qui ci riferiamo ai metodi di apprendimento automatico che non richiedono molti parametri. Questi approcci sono spesso chiamati “apprendimento pigro/lazy learning”. Poiché non costruiscono alcun modello. Tutto il lavoro viene svolto al momento della predizione.

Dato un semplice esempio: “arance” vs “limoni”, si usa un classificatore binario basato su due semplici caratteristiche (“altezza” e “larghezza”), costruendo un semplice confine di decisione lineare.



We can construct simple linear decision boundary:

$y = \text{sign}(h_\theta(\mathbf{x})), \quad h_\theta(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x}$



La classificazione è non-lineare: mette cose non identiche nella stessa classe, per cui una differenza nell'input a volte causa un cambiamento nullo nella risposta. La classificazione lineare significa che la parte che si adatta attraverso l'apprendimento è lineare (i parametri).
La "parte adattiva/adaptive part" è seguita da una non linearità f per prendere la decisione:

$$y = f(h_\theta(\mathbf{x})) \ , \quad h_\theta(\mathbf{x}) = \boldsymbol{\theta}^{\mathrm{T}} \mathbf{x}$$

Si usa quindi la classificazione come induzione, al fine di capire come avvicinarci al nostro confine di decisione lineare. Si adotta quindi il cosiddetto *instance-based learning*.
I sistemi di apprendimento automatico classificati come apprendimento basato sull'istanza sono quelli che imparano gli esempi di addestramento a memoria e poi generalizzano a nuove istanze in base a una misura di somiglianza. Si chiama basato sull'istanza perché costruisce le ipotesi a partire dalle istanze di addestramento.

È noto anche come apprendimento basato sulla memoria o apprendimento pigro. La complessità temporale media è O(n), dove *n* sono le istanze di apprendimento. Ciò è vantaggioso, infatti
è possibile effettuare approssimazioni locali alla funzione target. Questo algoritmo può adattarsi facilmente a nuovi dati, raccolti man mano, mantenendo (*storing*) i dati di allenamento/training ed usarli per istanze similari.
Ipotesi di base:

- L'output varia senza problemi con l'input
- I dati occupano un sottospazio dello spazio di input multidimensionale

Prendiamo l'esempio dei *nearest neighbors*. Si supponga che gli esempi di addestramento corrispondano a punti nello spazio euclideo d-dimensionale

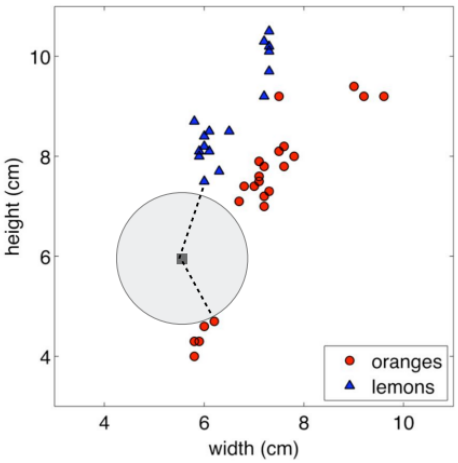
- Idea chiave: il valore della funzione target per un nuovo campione viene stimato a partire dagli esempi di addestramento noti (memorizzati)
  - Questo avviene calcolando le distanze tra il nuovo campione e tutti i campioni di formazione e tutti i campioni di addestramento
  - Regola decisionale: assegnare l'etichetta dell'esempio più vicino

Algorithm:

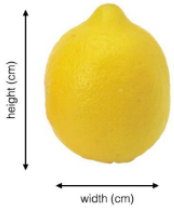
Find (x^*, y^*) (from the stored training set) closest to the test sample x

i.e. $x^* = \underset{x^{(i)} \in TrainSet}{\arg\min}\ \mathrm{Distance}(x^{(i)}, x)$. Output: $y = y^*$

Sull'esempio di prima, si cerca quindi di capire lo spazio in cui si opera e comprendere la distanza tra i punti di campionamento stabilita la dimensione:



Binary classifier based on two simple features:



Which distance?

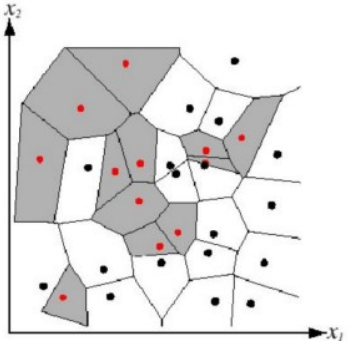
$$\|x_j^{(a)} - x_j^{(b)}\|_2 = \sqrt{\sum_{j=1}^{d} (x_j^{(a)} - x_j^{(b)})^2}$$

L2 (Euclidean)

$$\|x_j^{(a)} - x_j^{(b)}\|_1 = \sum_{j=1}^{d} |x_j^{(a)} - x_j^{(b)}|$$

L1

Il Nearest Neighbor non calcola esplicitamente i confini di decisione, ma questi possono essere dedotti
Visualizzazione del diagramma di Voronoi (cioè una partizione di un piano in regioni vicine a ciascuno di un dato insieme di oggetti (discreto)). Mostra come lo spazio di input è diviso in classi ed ogni segmento di linea è equidistante tra due campioni di classi opposte.



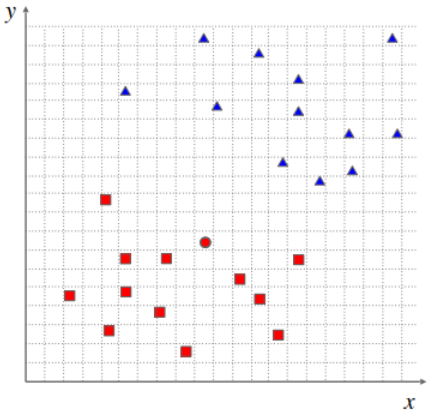
Computando le distanze tra i punti, si cerca di capire se il punto considerato appartenga di volta in volta alla classe 1 oppure alla classe 2:

- Compute distances (Euclidean):

$$d = \text{sqrt}((x_q - x_p)^2 + (y_q - y_p)^2)$$

Class 1

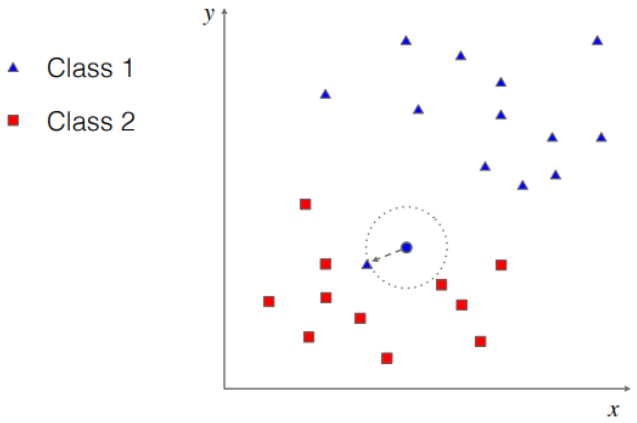
Class 2



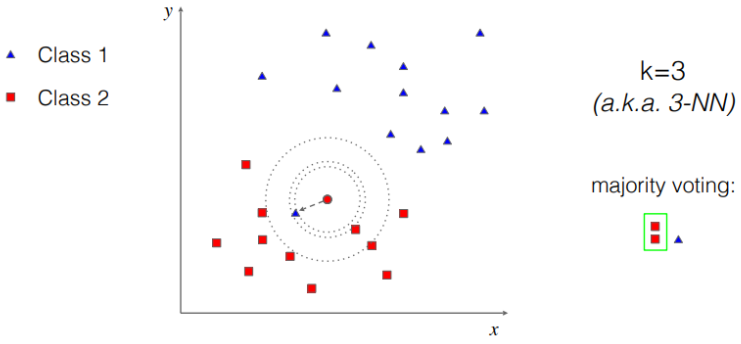
x_q	y_q
10	8

x_q	y_q	d	x_q	y_q	d
9	2	6.1	16	11	6.7
5	3	7.1	14	12	5.7
14	3	6.4	18	12	8.9
8	4	4.5	18	14	10
3	5	7.6	20	14	11.7
6	5	5.0	11	15	7.1
13	5	4.2	15	15	8.6
12	6	2.8	6	16	8.9
6	7	4.1	15	17	10.3
8	7	2.2	13	18	10.4
15	7	5.1	10	19	11
5	10	5.4	20	19	14.9

L'NN è sensibile agli *outliers* (cioè ai valori estremi, quelli molto distanti dalle osservazioni disponibili).
Come si vede, un punto della classe 1 completamente distante da quelli della sua classe e vicino a quelli della classe 2.



Ciò può essere generalizzabile a *k* dimensioni, per esempio come si vede k = 3, in cui la maggioranza dei punti rispetto a quello blu sono rossi:



Ora, dobbiamo modificare l'algoritmo come segue, generalizzandolo:

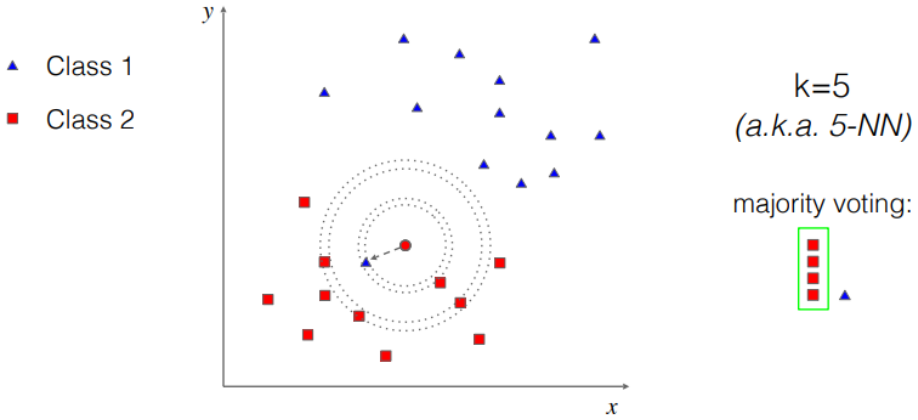
Algorithm:

Find k examples $(x^{(i)}, y^{(i)})$ (from training set) closest to the test sample x

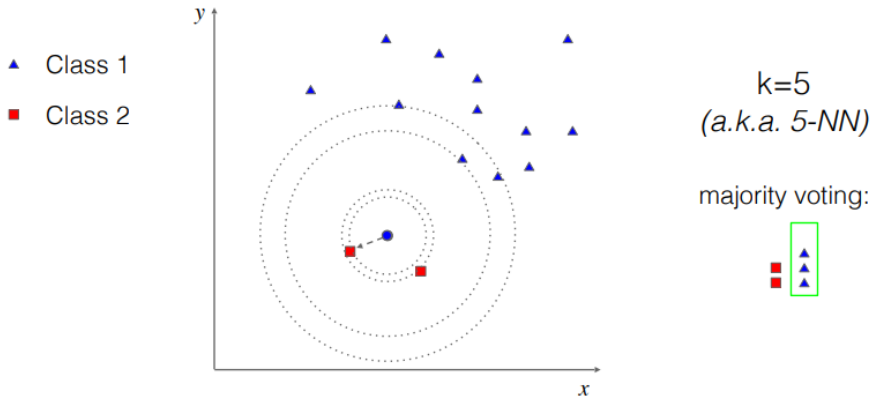
Output: $y = \arg\max_{y^{(z)}} \sum_{j=1}^{k} \delta(y^{(z)}, y^{(j)})$

Il parametro *k* ha un effetto molto forte:

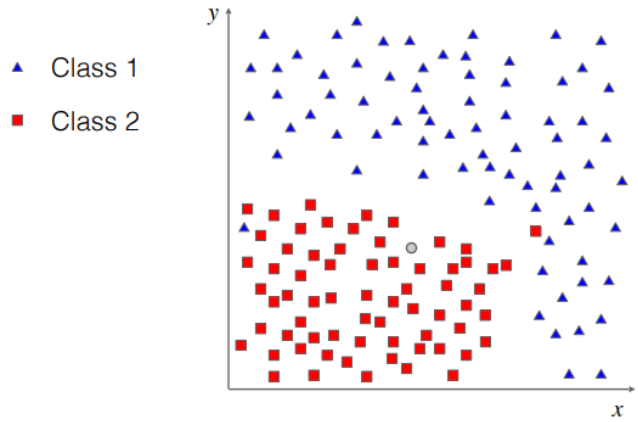
- Quando *k* è piccolo, si ha sensibilità agli outliers



- Quando *k* è grande, tutto è classificato come la classe più frequente



Ecco quindi che con l'algoritmo generalizzato, è bene avere tanti dati:



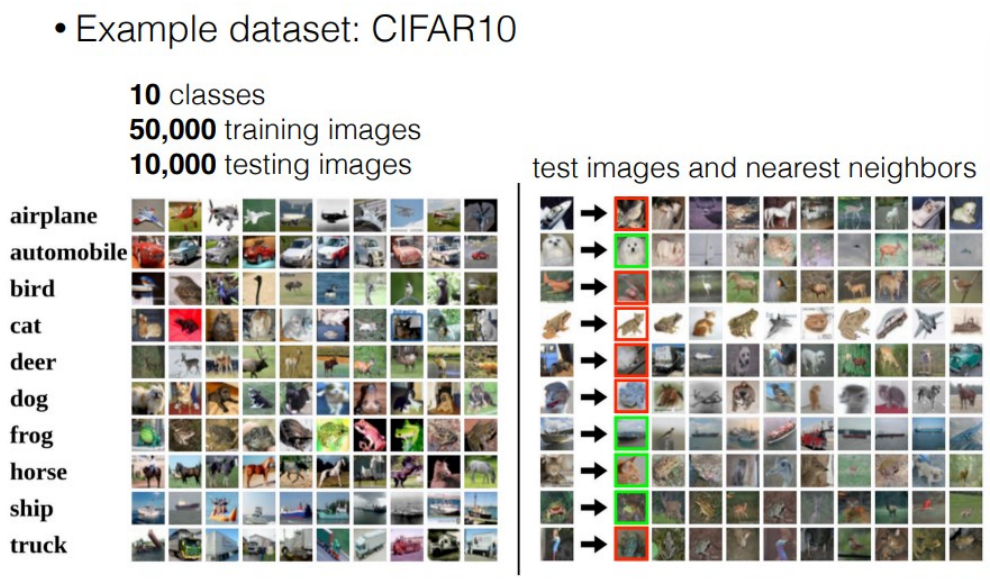
Quindi, come scegliere *k*?
Un *k* grande può portare a prestazioni migliori (se il set di addestramento è sufficientemente grande).

- Se scegliamo k troppo grande, potremmo finire con l'esaminare esempi che non sono "veri" vicini (sono lontani dal campione di test).
- Si può usare la convalida incrociata (cross-validation) per trovare il giusto k
- Regola empirica (rule of thumb): scegliere $k < sqrt(m)$, dove *m* è il numero di esempi di addestramento.

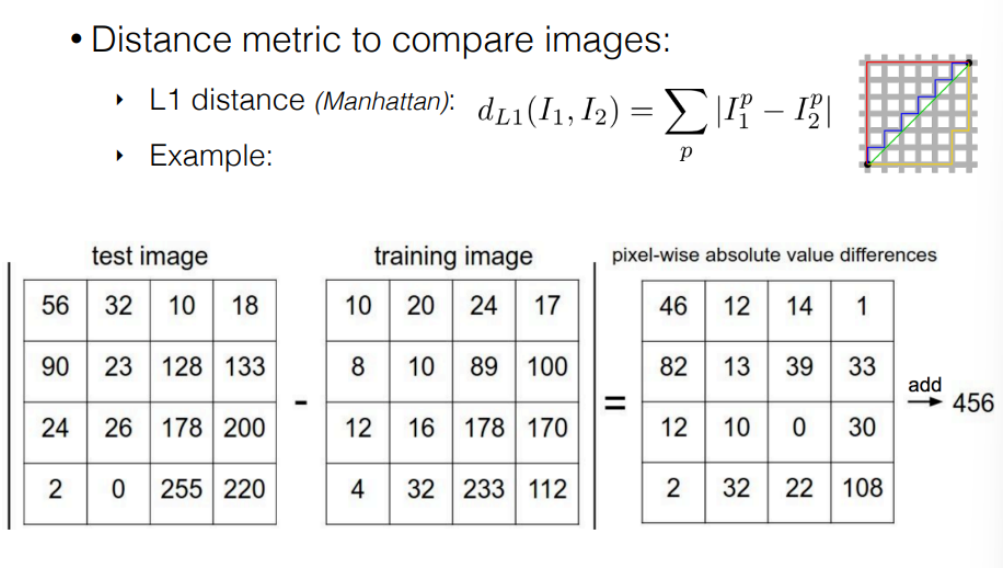
In merito invece alla classificazione delle immagini con k-NN, la rappresentazione più semplice è quella fornita dai pixel grezzi (raw pixels).
Una semplice pipeline di classificazione delle immagini:

- ridimensionare le immagini alla stessa scala/risoluzione
- rappresentare le immagini come una matrice 2D o 1D di pixel grezzi.
- definire una distanza (ad esempio L1 o L2) per confrontare le immagini.
- utilizzare k-NN per la classificazione

Un esempio è il dataset CIFAR-10, composto da 60000 immagini a colori 32x32 in 10 classi, con 6000 immagini per classe. Ci sono 50000 immagini di addestramento e 10000 immagini di test.



La metrica della distanza adottata è la cosiddetta *Manhattan distance*, dunque la distanza tra due punti misurata lungo assi perpendicolari. In un piano con p1 in (x1, y1) e p2 in (x2, y2), è |x1 - x2| + |y1 - y2|.



Un esempio di implementazione in Python:

- da una parte “memorizzo” i dati di addestramento (operando anche la previsione):

```
class NearestNeighbor:
  def __init__(self):
    pass

  def train(self, X, y):
    """ X is N x D where each row is an example. Y is 1-dimension of size N """
    # the nearest neighbor classifier simply remembers all the training data
    self.Xtr = X
    self.ytr = y

  def predict(self, X):
    """ X is N x D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # lets make sure that the output type matches the input type
    Ypred = np.zeros(num_test, dtype = self.ytr.dtype)
```

- per ciascuna immagine di test, si cerca l'immagine più vicina e si predice l'etichetta dell'immagine più vicina di addestramento:

```
# loop over all test rows
for i in xrange(num_test):
  # find the nearest training image to the i'th test image
  # using the L1 distance (sum of absolute value differences)
  distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
  min_index = np.argmin(distances) # get the index with smallest distance
  Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

return Ypred
```

Con N esempi, l'addestramento e la predizione di NN impiegano rispettivamente tempo O(1) ed O(N). Questo non va bene: vogliamo avere dei classificatori che siano veloci nella previsione (lenti nell'addestramento va bene, comunque).
Alcuni attributi (caratteristiche) hanno intervalli più ampi, quindi sono trattati come più importanti
È necessario applicare la scalatura delle caratteristiche, come ad esempio:
▸ Scalare linearmente l'intervallo di ogni caratteristica in modo che sia compreso in [0,1].
▸ Scalare linearmente ogni dimensione in modo da avere media 0 e varianza 1 (ad esempio, usando come media e varianza i parametri della Normale Standard).

Le caratteristiche *irrilevanti* ma correlate *aggiungono rumore* al calcolo della misura della distanza

- Si può applicare la selezione delle caratteristiche.
- Scegliere una misura di distanza migliore o applicare l'apprendimento metrico.

Costoso al momento del test: per ogni campione di test dobbiamo calcolare la distanza da tutti gli m esempi di addestramento

- È possibile calcolare una distanza approssimata (ad esempio, LSH. L'hashing sensibile alla località (LSH) è una tecnica algoritmica che fa confluire con alta probabilità elementi di input simili negli stessi "bucket").
- Utilizzare strutture di dati preordinate e veloci (ad esempio, kd-trees, cioè alberi che organizzano dati in uno spazio k-dimensionale).

Requisiti di memorizzazione: dobbiamo memorizzare tutti i dati di addestramento

- Rimuovere i dati ridondanti (condensazione)
- Il preordinamento spesso aumenta i requisiti di memorizzazione.

Dati altamente dimensionali ("maledizione della dimensionalità/curse of dimensionality")

- I dati di addestramento richiesti aumentano esponenzialmente con la dimensione; considerare l'utilizzo di una riduzione della dimensionalità (ad esempio, PCA. La PCA è una tecnica di riduzione lineare della dimensionalità che converte una serie di caratteristiche correlate nello spazio ad alta dimensionalità in una serie di caratteristiche non correlate nello spazio a bassa dimensionalità. Queste caratteristiche non correlate sono chiamate anche componenti principali. La PCA è una trasformazione lineare ortogonale, il che significa che tutte le componenti principali sono perpendicolari tra loro. Trasforma i dati in modo tale che la prima componente cerchi di spiegare la massima varianza rispetto ai dati originali. È un algoritmo non supervisionato, cioè non prende in considerazione le etichette di classe.).

Si possono ottenere buoni risultati in vari compiti, come ad esempio la *MNIST digit classification*, Si tratta di un set di dati composto da 60.000 immagini quadrate di 28×28 pixel in scala di grigi di cifre singole scritte a mano tra 0 e 9. Il compito è classificare una data immagine di una cifra in una delle 10 classi che rappresentano i valori interi da 0 a 9 inclusi. Si tratta di un set di dati ampiamente utilizzato e profondamente compreso e, per la maggior parte, "risolto".

Altro esempio utile è la *geolocalizzazione delle immagini*, cioè capendo dove è stata scattata una certa foto (il campionamento considera milioni di immagini attraverso un campionamento denso globalmente e rappresentano l'immagine sulla base di caratteristiche significative; anche qui usiamo k-NN con un *k* grande, ad es. 120).
Altre ancora sono il completamento di un'immagine (capendo la parte mancante), oppure l'annotazione di immagini e apprendimento della rilevanza dei tag tramite voto dei vicini.

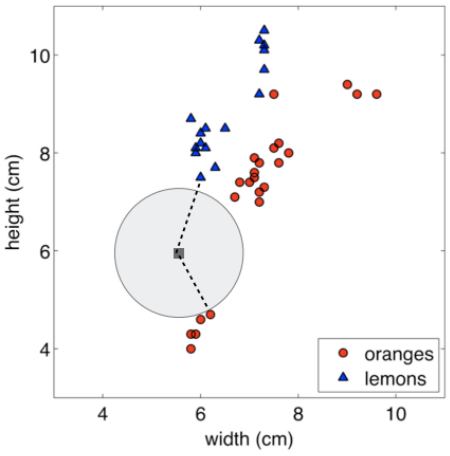
In conclusione, k-NN forma naturalmente confini decisionali complessi; si adatta alla densità dei dati
- Se abbiamo molti campioni, k-NN funziona bene.
- Principali limitazioni/problemi:
▸ Sensibile al rumore della classe e alle scale delle caratteristiche (attributi) Le distanze sono meno significative in dimensioni elevate.
▸ Scala linearmente con il numero di esempi di addestramento: cioè è estremamente costoso al momento del test.
- Bias induttivo: presuppone che i campioni vicini nello spazio delle caratteristiche condividano la stessa classe.

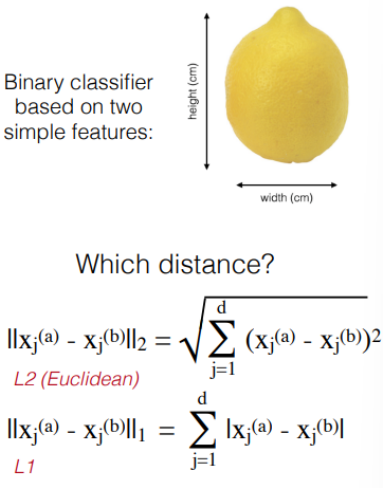
Decision Trees

Gli *alberi decisionali/decision trees (DT)* sono un metodo di apprendimento supervisionato non parametrico utilizzato per la classificazione e la regressione. L'obiettivo è creare un modello che preveda il valore di una variabile target attraverso l'apprendimento di semplici regole decisionali desunte dalle caratteristiche dei dati. Un albero può essere visto come un'approssimazione costante e frammentaria.

I nodi interni rappresentano gli attributi (features/caratteristiche) e i nodi foglia rappresentano etichette di classe o valori target.
Gli alberi decisionali sono ampiamente utilizzati nella ricerca per supportare l'analisi delle decisioni. Sono anche un algoritmo popolare nell'AI/ML. Una buona proprietà degli alberi decisionali è quella di fornire risultati interpretabili. Un albero decisionale rappresenta la funzione di ipotesi *h(x)* che si vuole modellare attraverso l'apprendimento automatico.

Torniamo all'esempio del classificatore binario "arance" vs "limoni" in cui sulla base delle due caratteristiche "altezza" e "peso" si caratterizza la distanza tra le componenti vettoriali di analisi (non-parametric k-NN):

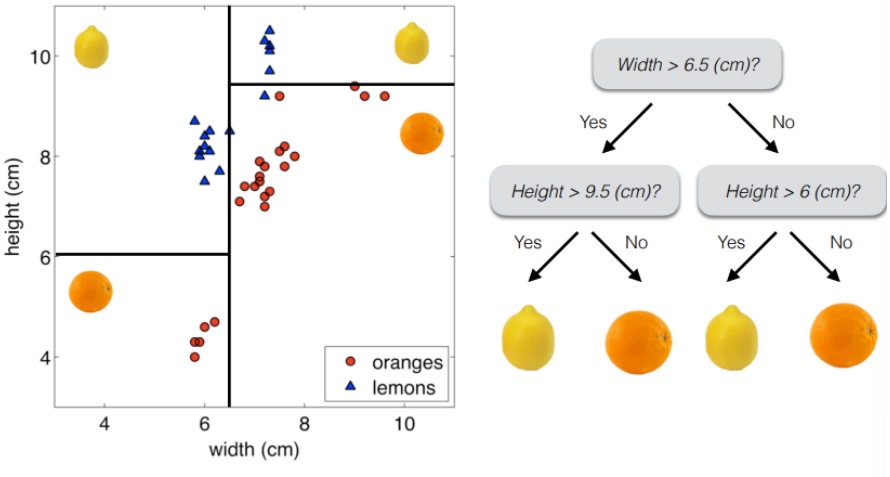




Idea generale:

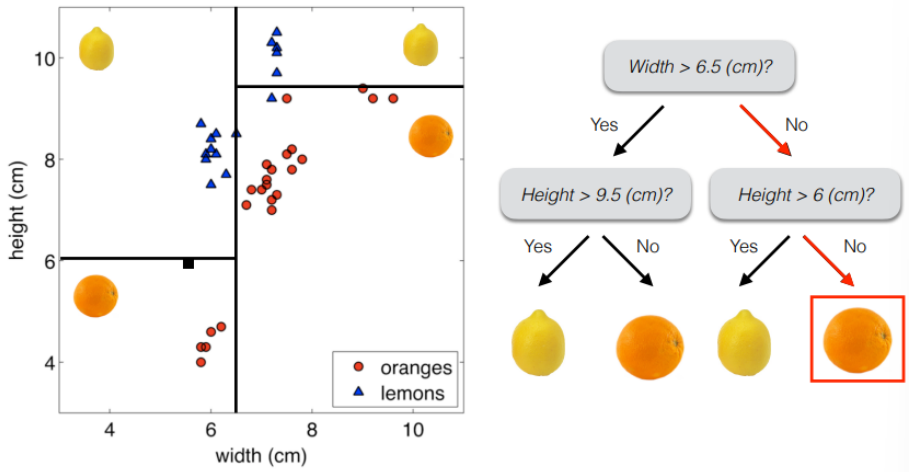
- Scegliere un attributo e fare un semplice test
- Condizionato da una scelta, scegliere un altro attributo
- Nelle foglie, assegnare una classe a voto di maggioranza
- Fare anche gli altri rami

• Decision tree classifier: “oranges” vs “lemons”



Al momento del test, si consideri “arancia”:

• At test time: ■ = → “orange”



Chiaro quindi come gli alberi di decisione siano anche visti come modelli generativi di regole induttive partendo da dati empirici. Come detto, i nodi interni sono relativi ad attributi di test, il branching è determinato dal valore dell’attributo e i nodi foglia sono gli output.

Vediamo un altro esempio in cui gli input hanno valori discreti e l'output è binario

- Costruiremo un albero decisionale per decidere se aspettare un tavolo al ristorante.
- Questo può essere rappresentato come una funzione booleana e il nostro obiettivo è quello di imparare una definizione per il predicato *WillWait*
- Un albero decisionale booleano è equivalente all'asserzione che l'attributo obiettivo è *true* se e solo se gli attributi di ingresso soddisfano uno dei percorsi che conducono a una foglia con valore *true*.

Per prima cosa, listiamo gli attributi considerati come input:

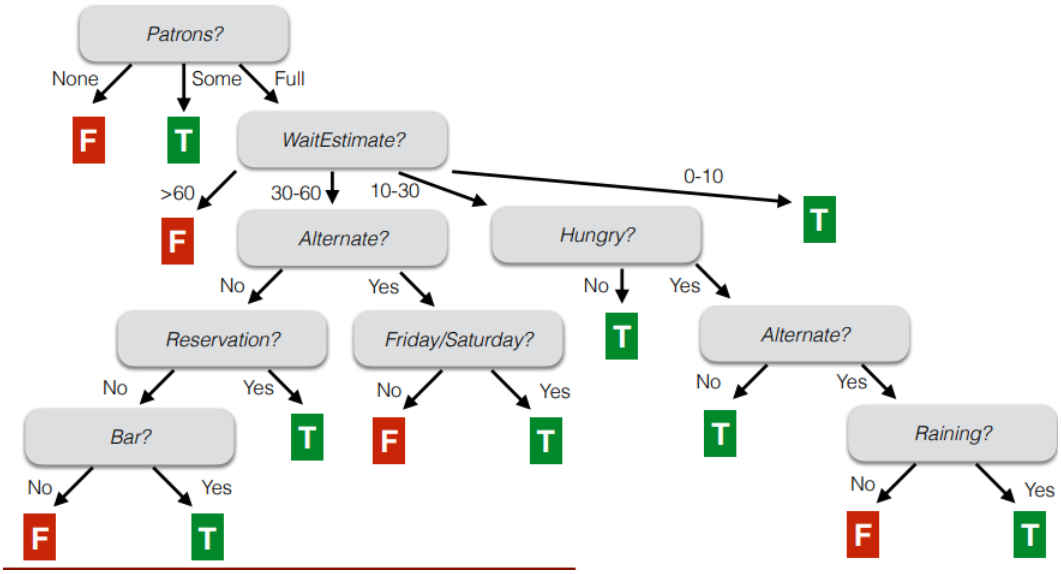
	Input Attributes										
Example	*Alt*	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	Target (*WillWait*)
$x^{(1)}$	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	$y^{(1)}$
$x^{(2)}$	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	$y^{(2)}$
$x^{(3)}$	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	$y^{(3)}$
$x^{(4)}$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	$y^{(4)}$
$x^{(5)}$	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	$y^{(5)}$
$x^{(6)}$	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	$y^{(6)}$
$x^{(7)}$	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	$y^{(7)}$
$x^{(8)}$	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	$y^{(8)}$
$x^{(9)}$	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	$y^{(9)}$
$x^{(10)}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	$y^{(10)}$
$x^{(11)}$	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	$y^{(11)}$
$x^{(12)}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	$y^{(12)}$

m=12 "training" examples

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. Hungry: whether we are hungry.
5. Patrons: how many people are in the restaurant (values are None, Some, and Full).
6. Price: the restaurant's price range ($, $$, $$$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (French, Italian, Thai or Burger).
10. WaitEstimate: the wait estimated by the host (0-10 minutes, 10-30, 30-60, >60).

A.Y. 2021/22: Introduction to Machine Learning

L'albero decide se aspettare (*true*) oppure non aspettare (*false*).



Un albero viene costruito dividendo il set di dati (di addestramento), in sottoinsiemi che costituiscono i figli successori

- Scegliere un attributo su cui scendere a ogni livello.
- Condizionare i punteggi precedenti (più alti)
- Limitare solo una dimensione alla volta
  - Nota: nell'esempio del limone/arancio, dividiamo ogni dimensione una sola volta, ma non è obbligatorio

- Dichiarare un'uscita quando si arriva in fondo, cioè che il sottoinsieme in un nodo ha tutti lo stesso valore dell'obiettivo y

Questa procedura viene ripetuta su ogni sottoinsieme derivato in modo ricorsivo (partizionamento ricorsivo). Ogni percorso dalla radice alla foglia definisce una regione R_n dello spazio di input.
In particolare, avendo un insieme di esempi di addestramento nella regione considerata, si avranno due alberi, uno di classificazione e regressioni, con output discreto (il valore più comune tra i nodi) e continuo (il valore medio tra i nodi degli alberi).

- Let $\{(x^{(n_1)}, t^{(n_1)}), \ldots, (x^{(n_k)}, t^{(n_k)})\}$ be the training examples falling into the region R_n
- Classification Tree:
  - Discrete output (class labels)
  - Leaf $y^{(k)}$ is set to the most common value in $\{(t^{(n_1)}, \ldots, t^{(n_k)})\}$
- Regression Tree:
  - Continuous output
  - Leaf $y^{(k)}$ is set to the mean value in $\{(t^{(n_1)}, \ldots, t^{(n_k)})\}$

Dato un input discreto, si avrà un output discreto:

- Un albero può esprimere qualsiasi funzione degli attributi di input (ad esempio, per le funzioni logiche: tabella di verità = percorso verso la foglia)

Dato un input continuo, si avrà un output continuo:

- Un albero può approssimare qualsiasi funzione (in modo arbitrariamente stretto)

Un albero decisionale rappresenta una funzione booleana. Ciascun percorso dalla radice alla foglia rappresenta una congiunzione di test sugli attributi. Diversi percorsi che portano alla stessa classificazione rappresentano una disgiunzione di congiunzioni. Queste due regole definiscono una serie di DNF (Disjunctive Normal Form), una per ogni classe.

La forma normale disgiuntiva (DNF) è la normalizzazione di una formula logica nella matematica booleana. In altre parole, una formula logica si dice in forma normale disgiuntiva se è una disgiunzione di congiunzioni con ogni variabile e la sua negazione è presente una volta in ogni congiunzione.
Un esempio:

- An example:

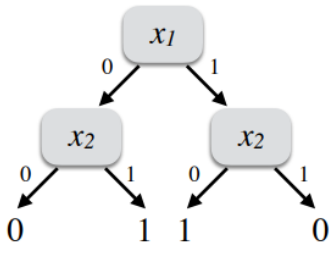
$x_1, x_2 \in \{0, 1\}$

$y = x_1 \text{ XOR } x_2$

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

DNF for $y = 1$

$(x_1=0 \text{ AND } x_2=1) \text{ OR } (x_1=1 \text{ AND } x_2=0)$



Banalmente, esiste un albero decisionale per qualsiasi set di allenamento con un percorso verso la foglia per ogni campione (a meno che f non sia deterministica in x).
Tuttavia, probabilmente non sarà generalizzabile a nuovi esempi.
Abbiamo bisogno di un qualche tipo di regolarizzazione per garantire alberi decisionali più compatti (e utili).

Apprendere il più semplice (cioè, il più piccolo) albero di decisione è un problema NP-Completo (non risolvibile in tempo polinomiale); cominciando da un albero di decisione vuoto, si divide sul miglior attributo successivo e si ricorre. Per capire qual è il miglior attributo, usiamo la *teoria dell'informazione (information theory)*.

Scritto da Gabriel

In particolare, abbiamo bisogno di un algoritmo chiamato *ID3 Algorithm*.
L'algoritmo ID3, acronimo di Iterative Dichotomiser 3, è un algoritmo di classificazione che segue un approccio greedy per la costruzione di un albero decisionale, selezionando il miglior attributo che produce il massimo guadagno di informazioni (IG) o la minima entropia (H).
In particolare:

- L'entropia è una misura della quantità di incertezza presente nel set di dati S
- Il guadagno di informazioni IG(A) indica quanto si è ridotta l'incertezza in S dopo la suddivisione dell'insieme S sull'attributo A.

I passi dell'algoritmo sono come segue:

1) Inizia creando un nodo radice *r* per l'albero.
2) Se tutti gli esempi in S appartengono alla stessa classe *y*, restituisce la nota radice *r* con l'etichetta y.
3) Se A è vuoto, restituisce r etichettato come la classe maggioritaria in S
4) Altrimenti, selezionare l'attributo "ottimale" $a \in A$
    a. Partire/suddividere l'insieme S in sottoinsiemi utilizzando *a* per i quali l'entropia risultante è minimizzata (▸) (o il guadagno di informazione è massimizzato)
    b. Creare un nodo dell'albero decisionale contenente l'attributo a
    c. Ricorrere ai sottoinsiemi usando gli attributi rimanenti

In generale:

1. Calcolare l'entropia del set di dati.
2. Per ogni attributo/caratteristica.

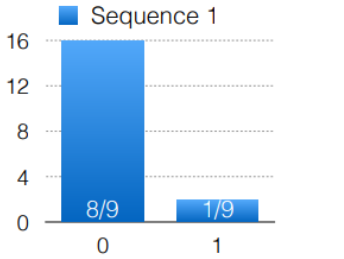
2.1. Calcolare l'entropia per tutti i valori categoriali.
2.2. Calcolare il guadagno di informazioni per la caratteristica.

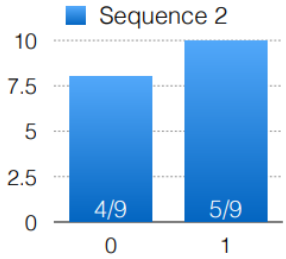
3. Trovare la caratteristica con il massimo guadagno di informazioni.
4. Ripetere l'operazione fino a ottenere l'albero desiderato.

Quantificare l'incertezza si comprende dall'esempio seguente, quindi la distribuzione dell'ottenere testa o croce da due sequenze di lanci:

Sequence 1: 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

Sequence 2: 0 1 1 0 1 0 1 1 0 1 0 1 1 1 0 1 0 0





Come detto, l'entropia H è una misura della quantità di incertezza (o di casualità) del set di dati S (o casualità) nell'insieme di dati S
Ci permette di rispondere a domande quali:

- Quanto ci sorprende un nuovo valore nella sequenza?
- Quanta informazione trasmette?

Matematicamente, abbiamo che:

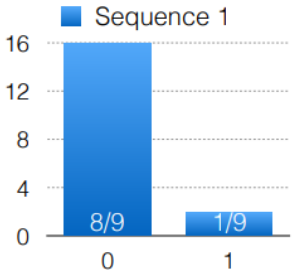
$$H(S) = -\sum_{c \in C} p(c) \log_2 p(c)$$, where:

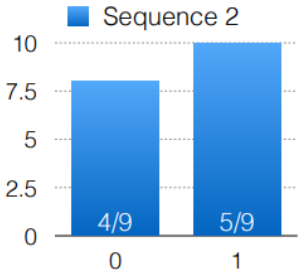
C is the set of classes in S

$p(c)$ is the proportion of elements in c to the elements in S

Nel nostro set di dati, si ottiene che:

$$H(S_1) = -\frac{8}{9}\log_2\frac{8}{9} - \frac{1}{9}\log_2\frac{1}{9} \approx 0.5 \qquad H(S_2) = -\frac{4}{9}\log_2\frac{4}{9} - \frac{5}{9}\log_2\frac{5}{9} \approx 0.99$$





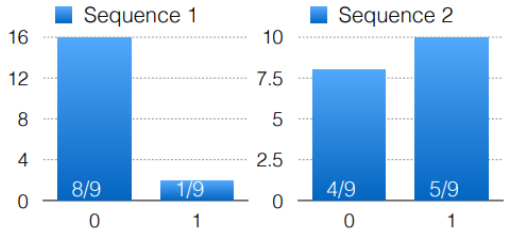
Data questa definizione di entropia, abbiamo che questo è un problema di classificazione binario. Pertanto, è utile definire *B(q)* come l'entropia di una variabile casuale booleana che è vera con probabilità *q*.

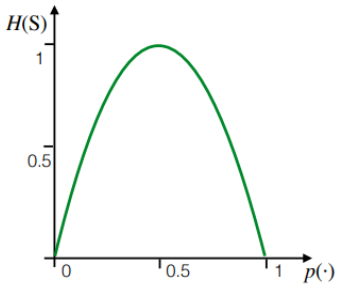
$$B(q) = -(q\log_2 q + (1-q)\log_2(1-q))$$

$$H(S_1) = B(\frac{8}{9}) \approx 0.5 \qquad H(S_2) = B(\frac{4}{9}) \approx 0.99$$

E graficamente abbiamo che:

$H(S_1) = B(\frac{8}{9}) \approx 0.5$ $H(S_2) = B(\frac{4}{9}) \approx 0.99$





In particolare:

- Entropia alta, la variabile ha una distribuzione simile a quella uniforme, l'istogramma è piatto e i valori campionati sono meno prevedibili
- Entropia bassa, la distribuzione ha vari massimi e minimi (similmente l'istogramma e i valori campionati sono più prevedibili

Tornando al secondo concetto cardine, Il guadagno di informazione (information gain) misura la differenza di entropia da prima a dopo la divisione dell'insieme *S* su un attributo *a*
In altre parole, IG misura quanto l'incertezza in *S* si è ridotta dopo la divisione dell'insieme *S* sull'attributo *a.*
Matematicamente:

- IG is defined as: $IG(S, a) = H(S) - \sum_{t \in T} p(t)\, H(t) = H(S) - H(S \mid a)$

  $H(S)$ is the entropy of dataset S

  T: the subsets created from splitting set S by attribute a

  $p(t)$ is the proportion of elements in t to the elements in S

  $H(t)$ is the entropy of subset t

Nell'ID3, il guadagno di informazione viene calcolato per ogni attributo rimanente; l'attributo con l'IG maggiore viene utilizzato per dividere l'insieme in questa iterazione. L'infogain (così abbreviato, dunque) è di interesse, secondo i passi detti dell'algoritmo, per capire quando questo *sia massimo*.

Tornando all'esempio di prima:

Example	Input Attributes										Target (*WillWait*)
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	
$x^{(1)}$	Yes	No	No	Yes	Some	$$$	No	Yes	French	0–10	$y^{(1)}$
$x^{(2)}$	Yes	No	No	Yes	Full	$	No	No	Thai	30–60	$y^{(2)}$
$x^{(3)}$	No	Yes	No	No	Some	$	No	No	Burger	0–10	$y^{(3)}$
$x^{(4)}$	Yes	No	Yes	Yes	Full	$	Yes	No	Thai	10–30	$y^{(4)}$
$x^{(5)}$	Yes	No	Yes	No	Full	$$$	No	Yes	French	>60	$y^{(5)}$
$x^{(6)}$	No	Yes	No	Yes	Some	$$	Yes	Yes	Italian	0–10	$y^{(6)}$
$x^{(7)}$	No	Yes	No	No	None	$	Yes	No	Burger	0–10	$y^{(7)}$
$x^{(8)}$	No	No	No	Yes	Some	$$	Yes	Yes	Thai	0–10	$y^{(8)}$
$x^{(9)}$	No	Yes	Yes	No	Full	$	Yes	No	Burger	>60	$y^{(9)}$
$x^{(10)}$	Yes	Yes	Yes	Yes	Full	$$$	No	Yes	Italian	10–30	$y^{(10)}$
$x^{(11)}$	No	No	No	No	None	$	No	No	Thai	0–10	$y^{(11)}$
$x^{(12)}$	Yes	Yes	Yes	Yes	Full	$	No	No	Burger	30–60	$y^{(12)}$

m=12 "training" examples

1. Alternate: whether there is a suitable alternative restaurant nearby.
2. Bar: whether the restaurant has a comfortable bar area to wait in.
3. Fri/Sat: true on Fridays and Saturdays.
4. Hungry: whether we are hungry.
5. Patrons: how many people are in the restaurant (values are None, Some, and Full).
6. Price: the restaurant's price range ($, $$, $$$).
7. Raining: whether it is raining outside.
8. Reservation: whether we made a reservation.
9. Type: the kind of restaurant (French, Italian, Thai or Burger).
10. WaitEstimate: the wait estimated by the host (0-10 minutes, 10-30, 30-60, >60).

A.Y. 2021/22: Introduction to Machine Learning

La selezione sul dataset *S* su m=12 esempi, nuovamente, è un problema di classificazione binaria e, come prima, è utile definire *B(q)* come l'entropia di una variabile casuale booleana che è vera con probabilità *q*.

- Attribute selection:

[recall: $IG(S, a) = H(S) - H(S \mid a)$]

$$H(S) = -\sum_{c \in C} p(c) \log_2 p(c)$$

1 3 4 6 8 12
2 5 7 9 10 11

Dataset S of m=12 examples

This is a binary classification problem: $c \in \{c_1, c_2\}$ (+ c_1, - c_2)

Therefore it will help to define $B(q)$ as the entropy of a Boolean random variable that is true with probability q:

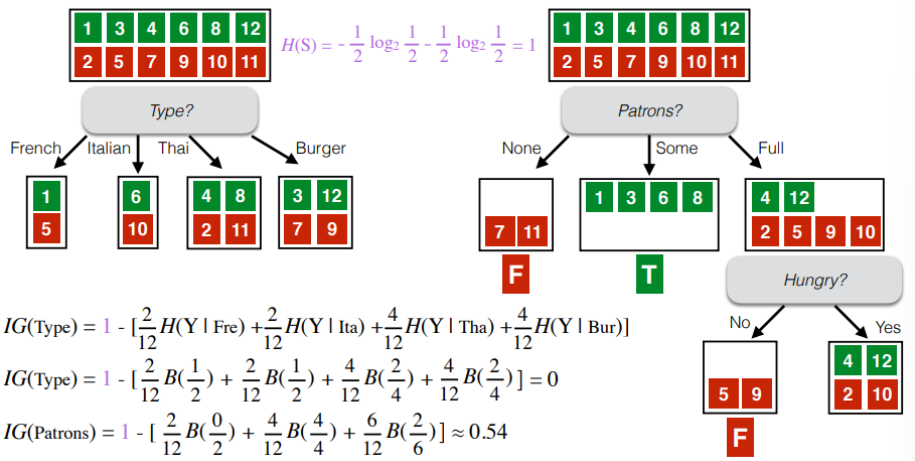
$$B(q) = -(q \log_2 q + (1-q) \log_2(1-q)) \text{ , where } q = \frac{|c_1|}{|c_1| + |c_2|}$$

$$H(S) = B(\tfrac{1}{2}) = -\tfrac{1}{2}\log_2\tfrac{1}{2} - \tfrac{1}{2}\log_2\tfrac{1}{2} = 1$$

Applicato al nostro esempio:

- Attribute selection:

[recall: $IG(S, a) = H(S) - H(S \mid a)$]

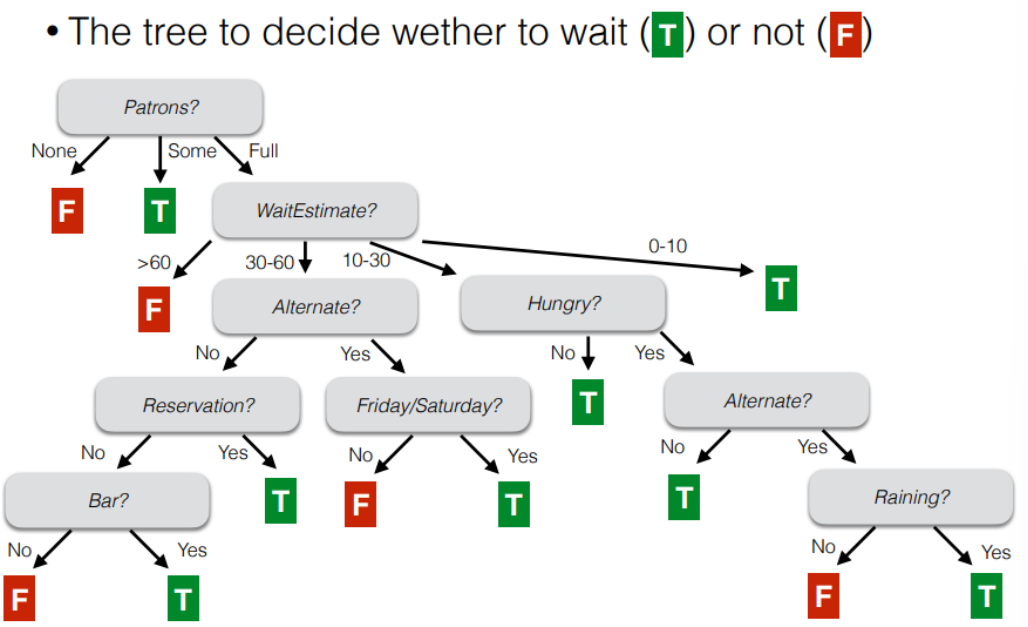


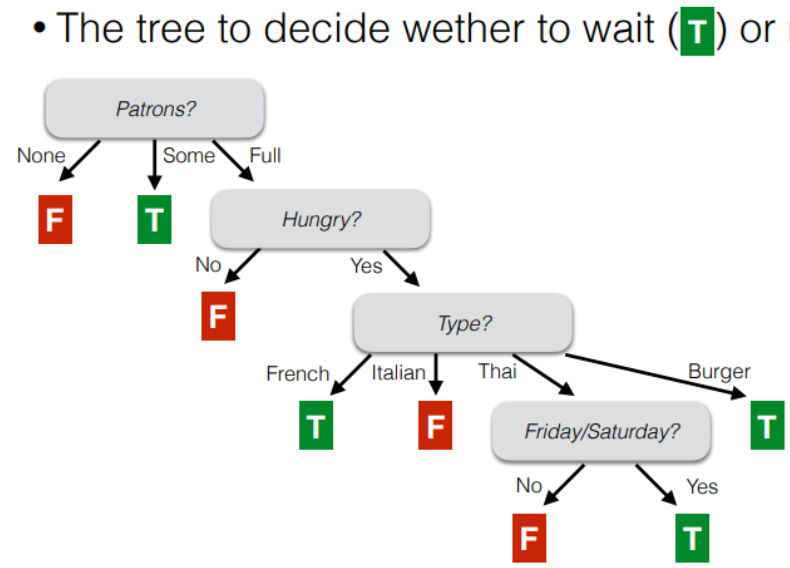
$$IG(\text{Type}) = 1 - [\tfrac{2}{12}H(Y \mid \text{Fre}) + \tfrac{2}{12}H(Y \mid \text{Ita}) + \tfrac{4}{12}H(Y \mid \text{Tha}) + \tfrac{4}{12}H(Y \mid \text{Bur})]$$

$$IG(\text{Type}) = 1 - [\tfrac{2}{12}B(\tfrac{1}{2}) + \tfrac{2}{12}B(\tfrac{1}{2}) + \tfrac{4}{12}B(\tfrac{2}{4}) + \tfrac{4}{12}B(\tfrac{2}{4})] = 0$$

$$IG(\text{Patrons}) = 1 - [\tfrac{2}{12}B(\tfrac{0}{2}) + \tfrac{4}{12}B(\tfrac{4}{4}) + \tfrac{6}{12}B(\tfrac{2}{6})] \approx 0.54$$

Il nostro albero, dunque, deve compiere il giusto numero di passi, in maniera tale da comprendere e gestire sottili distinzioni tra i dati analizzati (quindi, non sia troppo piccolo) e sia in grado di evitare attributi ridondanti, evitando anche problemi di overfitting sui dati di addestramento (quindi, non troppo grande). Cerchiamo l'ipotesi più semplice: l'albero più piccolo in grado di adattarsi alle osservazioni compiute. Alberi piccoli che piazzano grandi informazioni vicino alla radice sono preferibili.
In summa, graficamente, vediamo le osservazioni dall'albero di partenza al risultato:



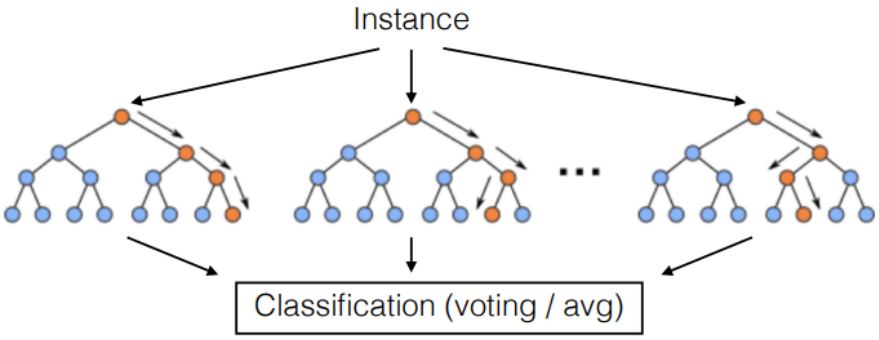


Problemi degli alberi di decisione

- Si hanno esponenzialmente meno dati ai livelli più bassi
- Un albero troppo grande può adattarsi troppo ai dati.
- Gli algoritmi greedy di solito non producono l'ottimo globale.
- Il guadagno di informazioni privilegia gli attributi che assumono un'ampia gamma di valori

In pratica, spesso si regolarizza il processo di costruzione per ottenere alberi piccoli ma altamente informativi. L'obiettivo è di prevenire l'overfitting, ad esempio attraverso il *pruning* (potatura), andando a settare una soglia sul numero di esempi dei nodi foglia o sulla profondità raggiunta dall'albero come avviene in *sklearn*.

Abbiamo per esempio la *random forest/foresta casuale*, algoritmo di classificazione costituito da molti alberi decisionali. Utilizza il *bagging* (in cui vari modelli dello stesso tipo addestrati su dataset diversi sono ottenuti dal dataset iniziale tramite campionamento casuale con rimpiazzo, definito bootstrap) e la casualità delle caratteristiche per combinare gli alberi.
In questo modo riduce la varianza del classificatore "originale" (albero decisionale) per poi farne un ensemble (parte del concetto di *ensemble learning*, usando vari algoritmi per ottenere migliori performance di predizione).



Probabilistic models & NLP

Si parte dal modello del *Turing test*, quindi l'*imitation game*:

- L'interrogatore (C) non può vedere i giocatori (A, B) e può comunicare con loro solo attraverso note scritte.
- L'interrogatore cerca di determinare quale giocatore è un computer e quale sia un umano

Rappresenta uno dei primi esempi di riconoscimento attraverso il ML, espandibile ad esempio attraverso *named entity recognition,* che cerca di individuare e classificare le entità denominate citate in un testo non strutturato in categorie predefinite, come nomi di persone, organizzazioni, località, codici medici, espressioni temporali, quantità, valori monetari, percentuali, ecc.

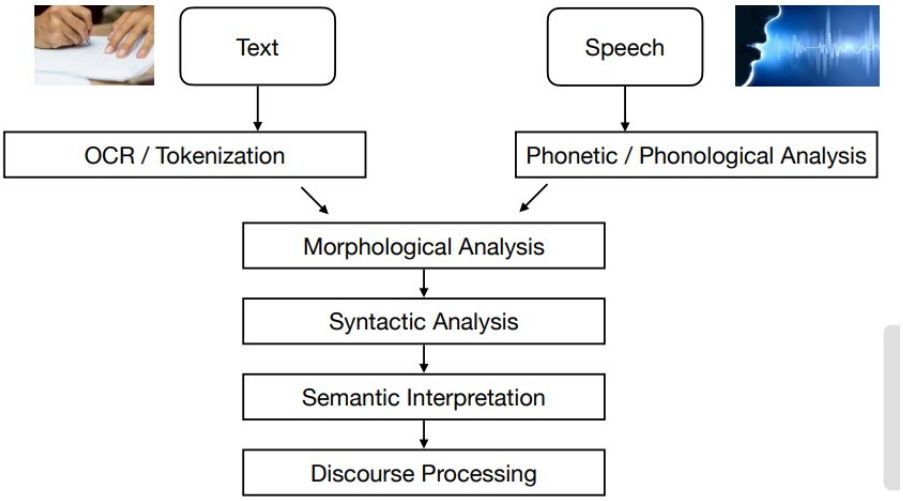
Cosa caratterizza il linguaggio umano?

- Un linguaggio umano è un sistema costruito per trasmettere il significato del parlante/scrittore. Non è solo un segnale ambientale, è una comunicazione intenzionale
- Utilizzando una codifica che i bambini possono imparare rapidamente e che cambia, evolvendosi nel corso del tempo. Un linguaggio umano è per lo più un sistema (discreto) di "segnalazione" simbolico/ categorico, un sistema di "segnalazione".
- I simboli di una lingua possono essere codificati come segnale per la comunicazione in diversi modi, quali il suono, un gesto, la scrittura / le immagini. Un simbolo è *invariante* tra le diverse codifiche.

Fatte queste premesse, introduciamo l'*NLP (Natural Language Processing)*, quindi la capacità di un programma informatico di comprendere il linguaggio umano così come viene parlato e scritto, definito linguaggio naturale. L'NLP consente ai computer di comprendere il linguaggio naturale come gli esseri umani. Che si tratti di linguaggio parlato o scritto, l'elaborazione del linguaggio naturale utilizza l'intelligenza artificiale per prendere input dal mondo reale, elaborarli e dargli un senso in modo comprensibile per il computer. Proprio come gli esseri umani hanno diversi sensori, come le orecchie per sentire e gli occhi per vedere, i computer hanno programmi per leggere e microfoni per raccogliere l'audio.

E proprio come gli esseri umani hanno un cervello per elaborare questi input, i computer hanno un programma per elaborare i rispettivi input. A un certo punto dell'elaborazione, l'input viene convertito in codice che il computer può comprendere.
Riuscirci porta ad alcuni goal utili (traduzione del linguaggio, sintesi del testo, risposta alle domande, ecc.).
Riuscire a comprendere completamente un linguaggio è molto difficile (data dalla variabilità ed ambiguità del nostro linguaggio).
Simile a questo è *Natural Language Understanding (NLU)* (qui nelle slide, sono sinonimi).

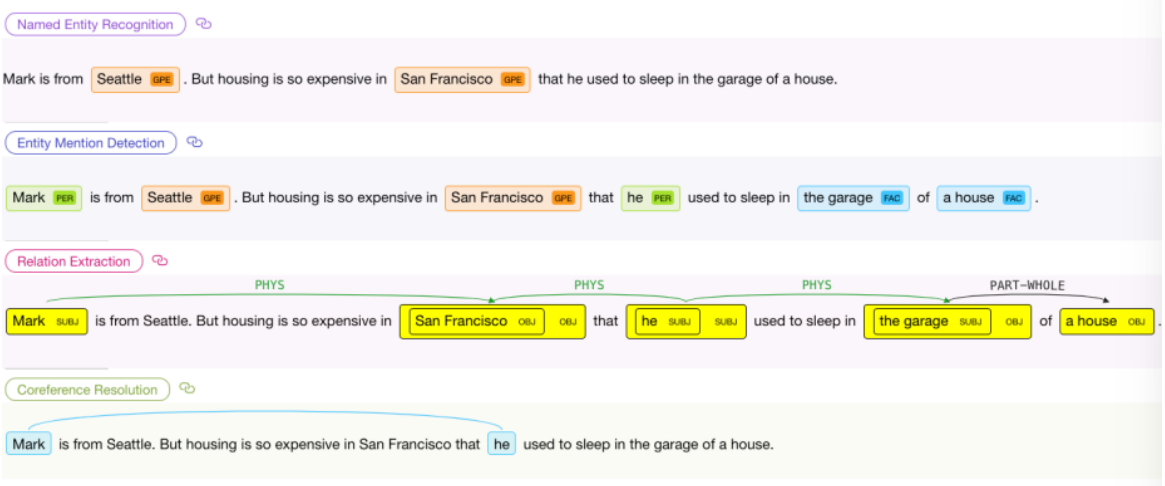
I vari livelli di NLP sono:



Esempi di applicazioni di NLP sono:

- Controllo ortografico, ricerca di parole chiave, ricerca di sinonimi
- Estrazione di informazioni da documenti (o siti web) (ad esempio, il prezzo di un prodotto, la data, la località, il nome di una persona o di un'azienda)
- Classificazione del testo, livello di lettura di testi scolastici, sentiment analysis di documenti più lunghi (capire ed estrarre opinioni da un testo)
- Traduzione automatica
- Chatbot
- Risposta alle domande

Ecco un esempio dei 4 compiti fondamentali di NLP/NLU:



Considereremo (principalmente) NLP in termini di compiti di ricerca di informazioni:

- Classificazione del testo
- Recupero di informazioni
- Estrazione di informazioni

Un fattore comune nell'affrontare questi compiti è l'uso di *modelli linguistici/language models*. cioè di modelli che predicono la distribuzione di probabilità delle espressioni linguistiche.
Definiamo quindi i *language models*:

- Un modello linguistico (statistico) è una distribuzione di probabilità su sequenze di parole
- Un *linguaggio formale* (ad esempio Python) è costituito da stringhe specificate da un insieme di regole chiamate *grammatica*.

  La grammatica definisce i programmi legali, ad esempio:

  ```
  print(2 + 2)
  ```
  This is a legal program in Python

  ```
  )2 +2) print
  ```
  This is not

- Un linguaggio formale ha anche regole che definiscono la *semantica* di un programma, ad esempio *4* è il *significato* di *(2 + 2)*.

Le lingue naturali non possono essere caratterizzate come un insieme definitivo di frasi

- “Non essere invitato è triste". tutti concordano sul fatto che sia una frase inglese.
- "Non essere invitati è triste". le persone non sono d'accordo sulla sua grammaticalità.

Le lingue naturali sono molto ampie e ambigue

- “Mucca infuriata ferisce il contadino con un'ascia".

Piuttosto che chiedersi se una stringa di parole sia o meno un membro dell'insieme che definisce il linguaggio, chiediamo *P(S=parole)*.

Richiamiamo concetti utili di probabilità:

- La *probabilità condizionata/conditional probability*, che si riferisce alle probabilità che un certo risultato si verifichi dato che si è verificato anche un altro evento. Spesso viene indicata come la probabilità di B dato A e viene scritta come P(B|A), dove la probabilità di B dipende da quella che si verifichi A.

  $$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

  Probability of *A given B* | Probability of *A and B* | Probability of *B*

- Nella teoria della probabilità, la *regola della catena/chain rule* (detta anche regola generale del prodotto) consente di calcolare qualsiasi membro della distribuzione congiunta di un insieme di variabili casuali utilizzando solo le probabilità condizionali. La regola è utile nello studio delle reti *bayesiane*, che descrivono una distribuzione di probabilità in termini di probabilità condizionali.

  $$P(A \cap B) = P(B \mid A) \cdot P(A).$$

- Per introdurre il concetto di *bayesiano*, dobbiamo introdurre il *teorema di Bayes*. Supponiamo di avere due insiemi di risultati A e B (detti anche eventi). Le probabilità di ciascun evento sono rispettivamente P(A) e P(B). La probabilità di entrambi gli eventi è indicata con la probabilità congiunta P(A, B), che possiamo espandere con le probabilità condizionali:

  $$P(A, B) = P(A|B)P(B),$$

  cioè la probabilità condizionata di A data B e la probabilità di B ci dà la probabilità congiunta di A e B. Ne consegue che:

  $$P(A, B) = P(B|A)P(A),$$

anche. Poiché i membri sinistri di entrambe le formule sono uguali, possiamo vedere che anche i membri destri sono uguali, dando il teorema di Bayes, che descrive le *concause che portano ad un certo evento*. Il bayesianismo si basa sulla nostra *conoscenza* degli eventi.

$$P(A|B)P(B) = P(B|A)P(A)$$
$$P(A|B) = \frac{P(B|A)P(A)}{P(B)},$$

Partendo da questo teorema, parliamo di *Bayesian inference/inferenza bayesiana*, che è un metodo di inferenza statistica in cui il teorema di Bayes viene utilizzato per aggiornare la probabilità di un'ipotesi man mano che si rendono disponibili ulteriori prove o informazioni. Essa permette di massimizzare la *stima a posteriori*. Per esempio, ci chiediamo: quale valore di θ massimizzerà la probabilità di θ dato D? Formalmente:

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} P(\theta|D),$$

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} \overbrace{P(\theta|D)}^{\text{posterior}}$$
$$= \arg\max_{\theta} \frac{\overbrace{P(D|\theta)}^{\text{likelihood}} \overbrace{P(\theta)}^{\text{prior}}}{\underbrace{P(D)}_{\text{evidence}}}$$

Per rispondere a questa domanda, usiamo il teorema di Bayes (sx) dando come risultato (dx)

$$\hat{\theta}_{\text{MAP}} = \arg\max_{\theta} P(D|\theta)\, P(\theta).$$

Andando ad attuare misurazioni, caratterizziamo i dati attraverso distribuzioni, ad esempio la *distribuzione binomiale*, cioè la probabilità di ottenere x successi in n prove indipendenti, che ci permette di ottenere la *media* delle nostre misurazioni, non dipendente da θ. La stima a posteriori, facendo varie prove, *dipende dalla stima a priori*. Si descrive quindi, in poche parole*, che il teorema di Bayes riesce a dare una stima considerando i valori ottenuti e misurati e i valori previsti, condizionati da eventi successivi, su tutti i dati, dando una distribuzione uniforme ed operando un'inferenza.*

- La *catena di Markov/Markov chain*, cioè un sistema matematico che sperimenta transizioni da uno stato all'altro secondo determinate regole probabilistiche. La caratteristica distintiva di una catena di Markov è che, indipendentemente dal modo in cui il processo è arrivato allo stato attuale, i possibili stati futuri sono fissi. In altre parole, la probabilità di passare a un particolare stato dipende esclusivamente dallo stato corrente e dal tempo trascorso. Lo spazio degli stati, o l'insieme di tutti gli stati possibili, può essere qualsiasi cosa: lettere, numeri, condizioni meteorologiche, risultati di baseball o performance azionarie. È un processo senza memoria ed è una sequenza di variabili aleatorie che soddisfano la proprietà di indipendenza condizionata, cioè esprimere con un prodotto tutti gli eventi poiché indipendenti:

$$P(X_n = i_n \mid X_{n-1} = i_{n-1}) = P(X_n = i_n \mid X_0 = i_0,\, X_1 = i_1,\, \ldots,\, X_{n-1} = i_{n-1}).$$

Introduciamo il cosiddetto *n-gram model*, che basa il language model come distribuzione probabilistica su sequenze di caratteri, in particolare di N caratteri e sequenze di simboli di lunghezza *n* (n-grammi). Si può pensare a un N-gramma come alla sequenza di N parole; in base a questa nozione, un 2-gramma (o bigramma/bigram) è una sequenza di due parole come "per favore gira", "gira i tuoi" o "i tuoi compiti", e un 3-gramma (o trigramma/trigram) è una sequenza di tre parole come "per favore gira i tuoi" o "gira i tuoi compiti".
n-gram character model: è il modello di linguaggio definito come distribuzione di probabilità su sequenze di caratteri. Più in generale, possiamo avere modelli a n-grammi su sequenze di parole (o altre unità), non solo su caratteri.

Scritto da Gabriel

Un n-gram è una catena di Markov di ordine *n-l*. In una catena di Markov la probabilità di un carattere ci dipende solo dalla probabilità dei caratteri immediatamente precedenti.

- Ad esempio, in un modello trigramma (catena di Markov di ordine 2) si ha:

$$P(c_i \mid c_{1:i-1}) = P(c_i \mid c_{i-2:i-1})$$

- Possiamo definire $P(c_{1:N})$ secondo il modello a trigrammi fattorizzando con la regola della catena e utilizzando l'ipotesi di Markov:

$$P(c_{1:N}) = \prod_{i=1}^{N} P(c_i \mid c_{1:i-1}) = \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1})$$

Cosa possiamo fare con i modelli di caratteri n-gram?

- Identificazione del linguaggio (ad esempio, "Hello world" vs "Come stai?")
- Un approccio semplice per l'identificazione della lingua si basa sulla costruzione di un modello di trigrammi di ogni lingua candidata $P(c_i \mid c_{i-2:i-1}, \ell)$
- Per ogni lingua il modello viene costruito contando i trigrammi in un corpus di quella specifica lingua (sono necessari circa 100K caratteri)
- Questo ci dà la probabilità: *P(Testo | Lingua)*

Nel caso dell'identificazione del linguaggio, vogliamo selezionare la lingua più probabile in base al testo, quindi (usando Bayes):

$$\ell^* = \operatorname*{argmax}_{\ell} P(\ell \mid c_{1:N}) = \operatorname*{argmax}_{\ell} P(\ell) P(c_{1:N} \mid \ell)$$

$$= \operatorname*{argmax}_{\ell} P(\ell) \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1}, \ell)$$

Questo è il modello di trigramma da un corpus (campione), ma che ne è della probabilità precedente *P(l)*?

- Possiamo stimare questi valori (ad esempio, controllando una pagina web a caso).
- La scelta di questi priori non è critica perché il modello trigramma seleziona una lingua che è di un ordine di grandezza superiore alle altre.

Alcune altre applicazioni popolari: correzione ortografica, classificazione del testo (genere), riconoscimento di entità denominate.

Intuizione: scegliere il modello che assegna una maggiore probabilità alla parola (successiva) che si verifica effettivamente.
Modello di Shannon: quanto bene riusciamo a prevedere la parola successiva (modello che cerca di analizzare come i messaggi sono inviati e ricevuti, tra mittente, ricevente e canale utilizzato).

"I always order pizza with cheese and ____"
"The 33rd President of the US was ____"
"I saw a ____"

mushrooms (0.1)
pepperoni (0.1)
anchovies (0.05)
pineapple (0.03)
…
fried rice (0.0001)
…
and (1e-100)

- Unigrams are terrible at this game
- So the best language model gives (on average) the best *P*(Sentence)

Lo stesso approccio si applica anche ai modelli di parole.

- Una differenza fondamentale è che il vocabolario è significativamente più ampio.
  - Di solito ci sono circa 100 caratteri nella maggior parte delle lingue.
  - I modelli di parole hanno decine di migliaia (o milioni) di simboli.
- Dobbiamo anche gestire le parole fuori dal vocabolario (un trucco comune è quello di definire una "parola speciale"). Vediamo cosa fanno i word models:

  Unigram: logical are as are confusion a may right tries agent goal the was . . .
  Bigram: systems are very similar computational approach would be represented . . .
  Trigram: planning and scheduling are integrated the success of naive bayes model is . . .

  Due esempi:
  1) Google n-gram viewer: https://books.google.com/ngrams
  2) Usa Presidents Example: https://goo.gl/u2TK3o

Obiettivo: calcolare la probabilità di una frase (sequenza di parole). $P(\mathrm{W}) = P(w_1, w_2, w_3, \ldots, w_n)$

Un *language model* cerca di computare una di queste: $P(\mathrm{W})$ or $P(w_n \mid w_1, w_2, w_3, \ldots, w_{n-1})$

La *chain rule* cerca di calcolare la probabilità condizionata delle parole:

$$P(w_1 w_2 \ldots w_n) = \prod_i P(w_i \mid w_1 w_2 \ldots w_{i-1})$$

Ad esempio:

$$P(\text{“its water is so transparent that”}) =$$
$$= P(\text{“its”})\, P(\text{“water} \mid \text{its”})\, P(\text{“is} \mid \text{its water”})\, P(\text{“so} \mid \text{its water is”})$$
$$P(\text{“transparent} \mid \text{its water is so”})\, P(\text{“that} \mid \text{its water is so transparent”})$$

Per stimare queste probabilità, non è sufficiente contarle, ci sono troppe frasi. Non vedremo mai abbastanza dati per stimare la probabilità.

Probabilistic models & NLP (part 2)

Cerchiamo di semplificare l'ipotesi di Markov in due modi diversi:

$$P(\text{“the} \mid \text{its water is so transparent that”}) \approx P(\text{“the} \mid \text{that”})$$

Oppure:

$$P(\text{“the} \mid \text{its water is so transparent that”}) \approx P(\text{“the} \mid \text{transparent that”})$$

In altre parole, si tratta di due esempi di n-gram models.

- Il primo esempio è un bigramma, cioè condizionato dalla parola precedente.
- Il secondo esempio è un trigramma, descritto come catena di Markov:

$$P(w_1 w_2 \ldots w_N) = \prod_{i=1}^{N} P(w_i | w_{1:i-1}) = \prod_{i=1}^{N} P(w_i | w_{i-2:i-1})$$

Dobbiamo dare una stima delle probabilità degli n-gram, tramite la stima di massima verosimiglianza/Maximum Likelihood Estimate/MLE. Esso è un metodo che determina i valori dei parametri di un modello. I valori dei parametri vengono trovati in modo da massimizzare la probabilità che il processo descritto dal modello abbia prodotto i dati effettivamente osservati.

Un problema di modellazione comune riguarda la stima di una distribuzione di probabilità congiunta per un insieme di dati.
Ad esempio, dato un campione di osservazioni (X) da un dominio (x1, x2, x3, ..., xn), dove ogni osservazione è estratta in modo indipendente dal dominio con la stessa distribuzione di probabilità (la cosiddetta distribuzione indipendente e identica, i.i.d., o quasi).

La stima della densità implica la selezione di una funzione di distribuzione di probabilità e dei parametri di tale distribuzione che meglio spiegano la distribuzione di probabilità congiunta dei dati osservati (X).

- Come si sceglie la funzione di distribuzione di probabilità?
- Come si scelgono i parametri della funzione di distribuzione di probabilità?

Questo problema è reso più impegnativo dal fatto che il campione (X) estratto dalla popolazione è piccolo e presenta rumore, il che significa che qualsiasi valutazione di una funzione di densità di probabilità stimata e dei suoi parametri avrà un certo errore.

Il metodo MLE è un approccio frequentista, rispetto ad esempio alla stima a posteriori/Maximum a Posteriori (MAP) di Bayes, che cerca di risolvere questo problema.
La stima della massima verosimiglianza comporta il trattamento del problema come un problema di ottimizzazione o di ricerca, in cui si cerca un insieme di parametri che risulti il migliore per la probabilità congiunta del campione di dati (X).

In primo luogo, si tratta di definire un parametro chiamato *theta* che definisce sia la scelta della funzione di densità di probabilità sia i parametri di tale distribuzione. Può essere un vettore di valori numerici i cui valori cambiano in modo uniforme e corrispondono a diverse distribuzioni di probabilità e ai loro parametri.

Nella stima di massima verosimiglianza, si vuole massimizzare la probabilità di osservare i dati dalla distribuzione di probabilità congiunta data una specifica distribuzione di probabilità e i suoi parametri, formalmente indicati come:

- $P(X \mid theta)$

Questa probabilità condizionale è spesso espressa usando la notazione del punto e virgola (;) invece della notazione a barre (|) perché theta non è una variabile casuale, ma un parametro sconosciuto. Ad esempio:

- $P(X\ ;\ theta)$

oppure

- $P(x1, x2, x3, \ldots, xn\ ;\ theta)$

La probabilità condizionale risultante viene definita probabilità di osservare i dati sulla base dei parametri del modello e viene scritta con la notazione L() per indicare la funzione di verosimiglianza. Ad esempio:
$L(X\ ;\ theta)$
L'obiettivo della stima della massima verosimiglianza è quello di trovare l'insieme dei parametri (*theta*) che massimizzano la funzione di verosimiglianza, ad esempio che producono il valore di verosimiglianza maggiore.

- $max\ L(X\ ;\ theta)$

Possiamo scomporre la probabilità condizionale calcolata dalla funzione di verosimiglianza.

Dato che il campione è composto da n esempi, possiamo inquadrarlo come la probabilità congiunta dei campioni di dati osservati x1, x2, x3, ..., xn in X dati i parametri della distribuzione di probabilità (theta).

- $L(x1, x2, x3, \ldots, xn\ ;\ theta)$

La distribuzione di probabilità congiunta può essere riformulata come la moltiplicazione della probabilità condizionata di osservare ciascun esempio dati i parametri della distribuzione.

- $prodotto\ da\ i\ a\ n\ P(xi\ ;\ theta)$

La moltiplicazione di molte piccole probabilità può essere numericamente instabile nella pratica; pertanto, è comune riformulare questo problema come la somma delle probabilità condizionali log dell'osservazione di ciascun esempio dati i parametri del modello.

Scritto da Gabriel

Data la spiegazione passo per passo in probabilità, applichiamolo al machine learning.
Possiamo inquadrare il problema dell'adattamento di un modello di apprendimento automatico come un problema di stima della densità di probabilità. In particolare, la scelta del modello e dei parametri del modello viene definita ipotesi di modellazione *h*, e il problema consiste nel trovare *h* che spieghi al meglio i dati *X*.

Partendo da $P(X\ ;\ h)$, cerchiamo $max\ L(X\ ;\ h)$, o meglio $max\ (somma\ da\ 1\ ad\ N\ di\ P(xi\ ;\ h)$.
Ciò significa che lo stesso schema di stima della massima verosimiglianza, generalmente utilizzato per la stima della densità, può essere utilizzato per trovare un modello di apprendimento supervisionato e i relativi parametri.
Questo fornisce la base per le tecniche fondamentali di modellazione lineare, come ad esempio:

- Regressione lineare, per la previsione di un valore numerico.
- Regressione logistica, per la classificazione binaria.

Nel caso della regressione lineare, il modello è vincolato a una retta e consiste nel trovare un insieme di coefficienti per la retta che meglio si adatta ai dati osservati. Fortunatamente, questo problema può essere risolto analiticamente (ad esempio, utilizzando direttamente l'algebra lineare).

Nel caso della regressione logistica, il modello definisce una linea e consiste nel trovare un insieme di coefficienti per la linea che meglio separa le classi. Questo problema non può essere risolto analiticamente e spesso viene risolto cercando nello spazio dei possibili valori dei coefficienti utilizzando un algoritmo di ottimizzazione efficiente come l'algoritmo BFGS o varianti.

In effetti, la maggior parte dei modelli di apprendimento automatico può essere inquadrata nell'ambito della stima della massima verosimiglianza, fornendo un modo utile e coerente di affrontare la modellazione predittiva come un problema di ottimizzazione. Un importante vantaggio dello stimatore di massima verosimiglianza nell'apprendimento automatico è che, all'aumentare delle dimensioni del set di dati, la qualità dello stimatore continua a migliorare.

Nel nostro caso, usiamo come formula e un successivo esempio di stima di probabilità su bigrammi/bigram probabilities :

$$P(w_i|w_{i-1}) = \frac{Count(w_i, w_{i-1})}{Count(w_{i-1})}$$

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

$P(\texttt{I}\,|\,\texttt{<s>}) = \frac{2}{3} = .67$ $P(\texttt{Sam}\,|\,\texttt{<s>}) = \frac{1}{3} = .33$ $P(\texttt{am}\,|\,\texttt{I}) = \frac{2}{3} = .67$
$P(\texttt{</s>}\,|\,\texttt{Sam}) = \frac{1}{2} = 0.5$ $P(\texttt{Sam}\,|\,\texttt{am}) = \frac{1}{2} = .5$ $P(\texttt{do}\,|\,\texttt{I}) = \frac{1}{3} = .33$

Altri esempi: Progetto Ristorante Berkeley (BeRP)

(Il Berkeley Restaurant Project (BeRP) è un banco di prova per un sistema di riconoscimento vocale in fase di sviluppo presso l'International Computer Science Institute di Berkeley, CA, in un progetto diretto da Nelson Morgan. BeRP è un sistema di comprensione del parlato spontaneo continuo a medio vocabolario, indipendente dal parlante, attualmente in fase di sviluppo presso l'ICSI. BeRP funziona come un consulente di conoscenza il cui dominio è costituito dai ristoranti della città di Berkeley. Il sistema funge da banco di prova per diversi progetti di ricerca, tra cui l'estrazione robusta delle caratteristiche, la stima di connessione

della verosimiglianza tra suoni e frasi, l'induzione automatica di lessici multipli di pronuncia, il rilevamento e la modellazione degli accenti stranieri, i modelli linguistici avanzati e la lettura delle labbra.)

Esempio di frasi raccolte: (circa 9.200 frasi)

- "Parlami di Chez Panisse" (è un ristorante a Berkeley).
- "Sto cercando un ristorante thailandese a prezzi medi".
- "Sto cercando un buon posto dove fare colazione".
- "Mi può dire se c'è qualche buon ristorante cantonese nelle vicinanze?"
- Quando è aperto il Caffè Venezia durante il giorno?

Conteggio dei bigrammi grezzi (su circa 9.200 frasi):

	i	want	to	eat	chinese	food	lunch	Spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

Probabilità bigrammi grezzi: normalizzare per unigrammi/unigrams.

- Raw unigram counts:

i	want	to	eat	chinese	food	lunch	spend
2553	927	2417	746	158	1093	341	278

- Result:

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.0008
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.0008	0	0.0017	0.28	0.0008	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.0009	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Stime dei bigrammi delle probabilità delle frasi. Un esempio:

$$P(\text{<s> I want English food </s>}) =$$
$$P(\text{I} \mid \text{<s>})$$
$$\cdot\, P(\text{want} \mid \text{I})$$
$$\cdot\, P(\text{English} \mid \text{want})$$
$$\cdot\, P(\text{food} \mid \text{English})$$
$$\cdot\, P(\text{</s>} \mid \text{food})$$
$$= 0.000031$$

In base alle probabilità e ai calcoli capiamo:

- Errori di grammatica
- Dati poco popolari
- Dati non utili alle osservazioni

$P(\text{I} \mid \text{<s>}) = 0.25$

$P(\text{English} \mid \text{want}) = 0.0011$ → *Chinese food is more popular*

$P(\text{Chinese} \mid \text{want}) = 0.0065$

$P(\text{to} \mid \text{want}) = 0.66$ → *Grammar (infinitive)*

$P(\text{eat} \mid \text{to}) = 0.28$

$P(\text{food} \mid \text{to}) = 0$ → *Contingency: unobserved in the corpus*

$P(\text{want} \mid \text{spend}) = 0$ → *Grammatical mistake*

Ora, questo è la base per la text classification/classificazione del testo: dato un testo/documento, decidere a quale di un insieme predefinito di classi appartiene (ad esempio identificazione della lingua, classificazione dei generi, oppure anche trovare lo spam, attraverso identificazione di parole chiave (comprare/economico, ecc.)).
Possiamo avere due modi diversi di affrontare il problema di classificazione del testo:

- Utilizzando regole di classificazione codificate a mano (hand-coded classification rules)
- Utilizzando la modellazione del linguaggio e l'apprendimento automatico.

Nell'approccio basato su regole (rule-based approach):

- Definiamo regole basate su una combinazione di parole (o di altre caratteristiche)
- Spam: indirizzi in lista nera O ("dollari" E "sono stati selezionati")
- Se le regole sono ben raffinate da un esperto, l'accuratezza può essere elevata.
- La creazione di tali regole è costosa e non sempre possibile.

Nell'approccio di modellazione linguistica (language modeling approach) definiamo:

- Un n-gram model della cartella spam: $\mathbf{P}(Message \mid spam)$
- Un n-gram model della cartella in arrivo: $\mathbf{P}(Message \mid ham)$

- Classifichiamo un messaggio usando la regola di Bayes (stima del massimo a posteriori):

$$\underset{c \in \{spam, ham\}}{\operatorname{argmax}} P(c \mid message) = \underset{c \in \{spam, ham\}}{\operatorname{argmax}} P(message \mid c)\, P(c)$$

where $P(c)$ is estimated by counting the total number of spam/ham

Introduciamo Naive Bayes: un semplice approccio di classificazione basato solo sulla regola di Bayes.
Si parte dall'applicare la formula di Bayes, in questo caso su un documento *d* e su una classe *c*:

$$P(c|d) = \frac{P(d|c)P(c)}{p(d)}$$

L'intuizione considera la stima a posteriori di Bayes, l'applicazione della regola di Bayes, togliendo il denominatore:

$$c_{MAP} = \arg\max_{c \in C} P(c|d) = \arg\max_{c \in C} \frac{P(d|c)P(c)}{p(d)} = \arg\max_{c \in C} P(d|c)P(c)$$

MAP is maximum a posteriori, i.e. most likely class

Bayes rule

Dropping the denominator

Consideriamo che *d* sia rappresentato usando *n* caratteristiche x_i. L'equazione precedente è riscritta sulla base della verosimiglianza (likelihood) e la stima a priori (prior).

$$c_{MAP} = \arg\max_{c \in C} P(d|c)P(c) = \arg\max_{c \in C} P(x_1, x_2, \ldots, x_n|c)P(c)$$

Likelihood *Prior*

Il calcolo di queste probabilità è difficile: potrebbe essere stimato solo se è disponibile un numero molto elevato di esempi di addestramento. Non possiamo prevedere quante volte capiti una classe, ma possiamo solo contare le frequenze relative ad un campione considerato.

$$c_{MAP} = \arg\max_{c \in C} P(x_1, x_2, \ldots, x_n|c)P(c)$$

This is hard: could be estimated only if a very large number of training example is available.

Q: how often does this class occur?

A: we can just count the relative frequencies in a corpus.

Facciamo alcune ipotesi semplificative per calcolare le probabilità $P(x_1, x_2, \ldots, x_n|c)$
Si consideri la *probabilità condizionata*: assumendo le probabilità sulle caratteristiche come $P(x_i|c)$ siano indipendenti dalla classe *c*.
Questo ci restituisce il calcolo del classificatore *multinomiale* (su tutte le singole probabilità, a prescindere dal loro numero):

$$c_{MAP} = \arg\max_{c \in C} P(x_1, x_2, \ldots, x_n|c)P(c)$$

$$c_{NB} = \arg\max_{c \in C} P(c) \prod_{x \in X} P(x|c)$$

Nell'approccio di apprendimento automatico rappresentiamo il messaggio come un insieme di caratteristiche (x, y) e utilizziamo un classificatore Language Model e Naive Bayes
Ma anche altri: k-NN, SVM, Regressione Logistica, ecc.

Usiamo un modello alternativo semplice: il modello *Bag-of-Words*/bagaglio di parole/*vector model*.

Il Bag of words è una tecnica di elaborazione del linguaggio naturale per la modellazione del testo. In termini tecnici, possiamo dire che si tratta di un metodo di estrazione di caratteristiche con dati testuali. Questo approccio è un modo semplice e flessibile di estrarre caratteristiche dai documenti.

Un bagaglio di parole è una rappresentazione del testo che descrive l'occorrenza di parole all'interno di un documento. Si tiene conto solo del numero di parole e non si tiene conto dei dettagli grammaticali e dell'ordine delle parole. Si chiama "bag" di parole perché qualsiasi informazione sull'ordine o sulla struttura delle parole nel documento viene scartata. Il modello si preoccupa solo di sapere se le parole conosciute sono presenti nel documento, non di sapere in quale punto del documento.

Si ha quindi il calcolo della frequenza di ciascuna parola. Un esempio:

- A simple example:

D_1 = "John likes to watch movies. Mary likes movies too."
D_2 = "John also likes to watch football."

BoW_1 = {John:1,likes:2,to:1,watch:1,movies:2,Mary:1,too:1}
BoW_2 = {John:1,also:1,likes:1,to:1,watch:1,footbal:1}

- I vettori di caratteristiche sono grandi e scarsi.
- La nozione di ordine delle parole è persa: BoWs (Bag of Words) e unigrammi danno la stessa probabilità a qualsiasi permutazione di un testo
  - Gli n-grammi di ordine superiore mantengono una nozione locale dell'ordine delle parole.
  - Ma usando i bigrammi il numero di caratteristiche è al quadrato, con i trigrammi è al cubo, ecc.

Nella BLP "tradizionale" le parole vengono considerate come simbolo discreto (numerabile).
Partendo dalla rappresentazione BoW, usiamo una rappresentazione ancora più semplice, il vettore *one-hot vector*, quindi lineare, avendone come dimensione il numero di parole in un vocabolario (almeno 500000).

HoT_1 = [0,1,1,0,1,1,0,1,1,0]

La rappresentazione delle parole come simboli discreti, come i vettori a un solo punto/one-hot vectors, presenta un problema importante:

- "San Francisco Hotel": `hotel = [0000000010000…00]`
- "San Francisco Motel": `motel = [0001000000000…00]`

Questi due vettori sono ortogonali, cioè non esiste una nozione naturale di somiglianza per i vettori one-hot. Possibili soluzioni:

- La "via della rappresentazione della conoscenza/knowledge representation way": ci si può basare su liste di sinonimi o tassonomie (ad esempio WordNet) per ottenere la somiglianza. Il significato prende in considerazione:

Il simbolo (significatore) – Il significato (concetto) – Notazione

e.g. synonym sets containing "good":

```
from nltk.corpus import wordnet as wn
for synset in wn.synsets("good"):
    print "(%s)" % synset.pos(),
    print ", ".join([l.name() for l in synset.lemmas()])
```

```
(adj) full, good
(adj) estimable, good, honorable, respectable
(adj) beneficial, good
(adj) good, just, upright
(adj) adept, expert, good, practiced,
proficient, skillful
(adj) dear, good, near
(adj) good, right, ripe
…
(adv) well, good
(adv) thoroughly, soundly, good
(n) good, goodness
(n) commodity, trade good, good
```

e.g. hypernyms of "panda":

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(panda.closure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

- Problemi con risorse come WordNet
  - Ottima risorsa, ma mancano le sfumature.
    - Ad esempio, "proficient" è elencato come sinonimo di "good", ma questo è corretto solo in alcuni contesti.
  - Mancano nuovi significati delle parole
    - Ad es. malvagio, cazzuto, intelligente, mago, genio, ninja, ...
    - Impossibile tenersi aggiornati!
  - Soggettivo
  - Richiede il lavoro umano per essere creato e adattato
  - È difficile calcolare un'accurata somiglianza di parole.

- La "via dell'apprendimento automatico/machine learning way": imparare a codificare la somiglianza nei vettori stessi.

Idea fondamentale: il significato di una parola è dato dalle parole che compaiono frequentemente nelle vicinanze. Quando una parola *w* appare in un testo, il suo contesto è l'insieme delle parole che appaiono nelle vicinanze, all'interno di una finestra di dimensioni fisse. Usare i molti contesti di w per costruire una rappresentazione di w.

…government debt problems turning into ***banking*** *crises as happened in 2009…*
…saying that Europe needs unified ***banking*** *regulation to replace the hodgepodge…*
…India has just given its ***banking*** *system a shot in the arm…*

These context words will represent banking

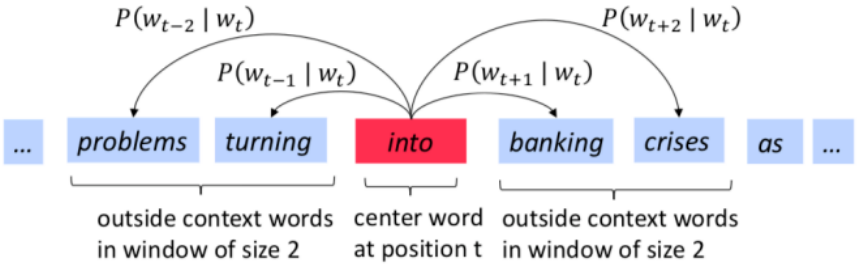
Costruiamo un vettore denso per ogni parola, in modo che sia simile ai vettori di parole che appaiono in contesti simili. Questo vettore è un *word vector* (detti anche *word embeddings/word representations)*.

$$banking = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

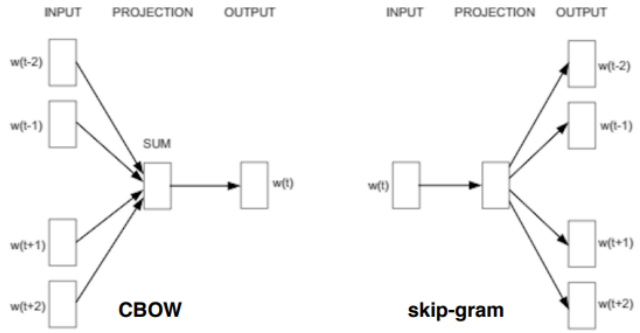
Esistono dei framework di apprendimento per questi vettori:
ad esempio, *Word2Vec*.

- Ogni parola di un vocabolario fisso è rappresentata da un vettore
- Passare in rassegna ogni posizione *t* del testo, che ha una parola centrale *c* e parole di contesto ("esterne") *o*
- Utilizzare la somiglianza dei vettori di parole per *c* ed *o* per calcolare la probabilità di *o* in base a *c* (o viceversa)

- Continuare a regolare i vettori di parole per massimizzare questa probabilità.

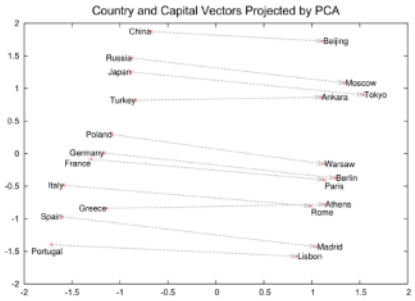


Word2Vec è una rete neurale (un autoencoder, rete neurale utilizzata per ottenere una rappresentazione compressa dei dati in input.) che implementa questa idea per imparare vettori di parole.
Lo fa utilizzando il contesto per predire una parola target (CBOW, Continuous Bag of Words).
Oppure usando una parola per predire un contesto di destinazione (skip-gram, usato per predire la parola di contesto per una data parola di destinazione. È l'inverso dell'algoritmo CBOW).



I vettori di parole appresi catturano regolarità sintattiche e semantiche significative e semantiche in modo molto semplice (link: https://projector.tensorflow.org/)

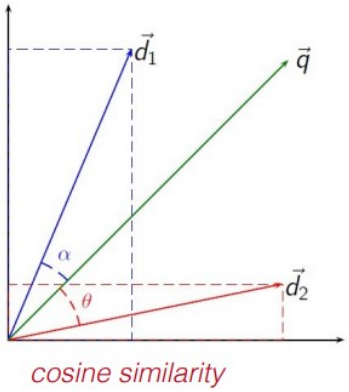
Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza



Abbiamo appena visto i vettori di parole come un approccio alternativo per apprendere le rappresentazioni del testo dai dati. Come visto, la parola "banking" viene rappresentata con un vettore;
ma come rappresentare frasi/documenti? L'approccio semplice (naive) potrebbe essere la media (o massima o la somma) dei word vector oppure adottare schemi di fusione più avanzati.

Il modello spaziale vettoriale (o modello vettoriale dei termini) è il modello in cui un testo viene rappresentato come un vettore di termini. L'enfasi è data sull'interpretazione geometrica, infatti:

- I documenti sono rappresentati da: $\mathbf{d}_j = (w_{1,j}, w_{2,j}, \ldots, w_{v,j})$
- Ora abbiamo un nuovo documento: $\mathbf{q} = (q_{1,i}, q_{2,i}, \ldots, q_{v,i})$
- Calcoliamo la somiglianza tra *q* e d_2 con: $cos(\theta) = \frac{\mathbf{d}_2 \cdot \mathbf{q}}{\|\mathbf{d}_2\| \cdot \|\mathbf{q}\|}$



cosine similarity

La somiglianza del coseno (cosine similarity) misura la somiglianza tra due vettori di uno spazio a prodotto interno. È misurata dal coseno dell'angolo tra due vettori e determina se due vettori puntano più o meno nella stessa direzione. Viene spesso utilizzata per misurare la somiglianza tra documenti nell'analisi del testo.

Normalizzazione della lunghezza/length normalization: un vettore può essere normalizzato dividendo ciascuna delle sue componenti per la sua lunghezza

- Possiamo usare la norma L2 (che mappa i vettori sulla sfera unitaria).
- Come risultato, i documenti più lunghi e quelli più brevi hanno pesi dello stesso ordine di grandezza
- Per i vettori normalizzati è equivalente al dot product/prodotto scalare:

$$cos(\theta) = \frac{\mathbf{d}_2 \cdot \mathbf{q}}{\underbrace{\|\mathbf{d}_2\|}_{1} \cdot \underbrace{\|\mathbf{q}\|}_{1}}$$

$$\|\mathbf{d}_2\| = \sqrt{\sum_i w_{i,2}^2} = 1$$

Information Retrieval: il compito di trovare documenti di natura non strutturata (di solito testo) che soddisfa un'esigenza informativa all'interno di grandi collezioni.
Un sistema IR può essere caratterizzato da:
‣ Un corpus di documenti
‣ Query poste in un linguaggio di interrogazione (ad esempio utilizzando operatori booleani)
‣ Un insieme di risultati, cioè un sottoinsieme di documenti giudicati rilevanti per l'interrogazione
‣ Una presentazione dell'insieme di risultati (di solito una lista classificata)

Una *funzione di scoring/scoring function* prende un documento, una query e restituisce un punteggio numerico
‣ Principio chiave: il più rilevante ha il punteggio più alto.
‣ Ad esempio, la funzione di punteggio Okapi BM25 (definita da tre fattori)

- La frequenza con cui un termine della query appare in un documento (noto anche come TF: Term Frequency)
- La frequenza inversa del documento (IDF) del termine
- La lunghezza del documento

Sempre nel contesto di questa funzione, assumiamo di aver creato un indice di N documenti.
- Calcoliamo TF(q_i, d_j), cioè il numero di volte q_i appare nel documento d_j
- Assumiamo una tabella dei conteggi di frequenza dei documenti DF(q_i)

$$BM25(d_j, q_{1:N}) = \sum_{i=1}^{N} IDF(q_i) \cdot \frac{TF(q_i, d_j) \cdot (k+1)}{TF(q_i, d_j) + k \cdot (1 - b + b \cdot \frac{|d_j|}{L})}$$

$$where\ \ IDF(q_i) = \log \frac{N - DF(q_i) + 0.5}{DF(q_i) + 0.5}$$

La *funzione di peso tf-idf (term frequency–inverse document frequency)* può essere utilizzato come uno schema di ponderazione (smoothing) nel modello classico dello spazio vettoriale.
Nel modello di spazio vettoriale i documenti sono rappresentati da:

$$\mathbf{d}_j = (w_{1,j}, w_{2,j}, \ldots, w_{v,j})$$

Invece di avere semplici termini di peso abbiamo:

$$w_{t,j} = tf_{t,j} \cdot \log \frac{|D|}{\{d' \in D | t \in d'\}}$$

where $tf_{t,j}$ is term frequency of t in document j (local parameter)

and $\log \frac{|D|}{\{d' \in D | t \in d'\}}$ is inverse doc. frequency (global parameter)

Abbiamo un campione di 2 documenti come segue:

Document 1

Term	Term Count
this	1
is	1
a	2
sample	1

Document 2

Term	Term Count
this	1
is	1
another	2
example	3

In generale:
- Si calcola il TF di una parola
- IDF conta la frazione di documenti che include la parola
- Se TF-IDF è zero, la parola non è informativa, altrimenti restituisce una parola usata spesso.

1)

- The TF for the term “this” is computed as follows:

  $tf(\text{“this”}, d_1) = 1/5 = 0.2 \quad tf(\text{“this”}, d_2) = 1/7 \approx 0.14$

- IDF accounts for the ratio of documents that include “this”:

  $idf(\text{“this”}, D) = \log(2/2) = 0$

- So TF-IDF is zero for the word “this” (i.e. not informative):

  $tf\text{-}idf(\text{“this”}, d_1, D) = 0.2 * 0 = 0$
  $tf\text{-}idf(\text{“this”}, d_2, D) = 0.14 * 0 = 0$

2)

- The TF for the term “this” is computed as follows:

  $tf(\text{“example”}, d_1) = 0/5 = 0 \qquad tf(\text{“example”}, d_2) = 3/7 \approx 0.429$

- IDF accounts for the ratio of docs that include “example”:

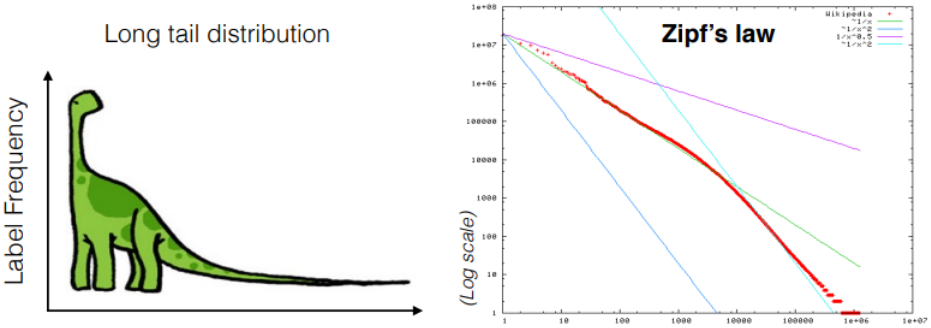
  $idf(\text{“example”}, D) = \log(2/1) = 0.301$

- So TF-IDF is more interesting in this case:

  $tf\text{-}idf(\text{“example”}, d_1, D) = 0 * 0.301 = 0$
  $tf\text{-}idf(\text{“example”}, d_2, D) = 0.429 * 0.301 \approx 0.129$

Quanti termini frequenti e quanti infrequenti dobbiamo aspettarsi in una raccolta di documenti/su campioni (corpus)?

- Ci sono pochi termini molto frequenti e molti termini rari.
- Legge di Zipf / Zipf’s Law: dato un corpus, la frequenza di una parola è inversamente proporzionale al suo rango nella raccolta di documenti. Alcuni esempi della legge su cinque linguaggi nell’immagine successiva a questa.

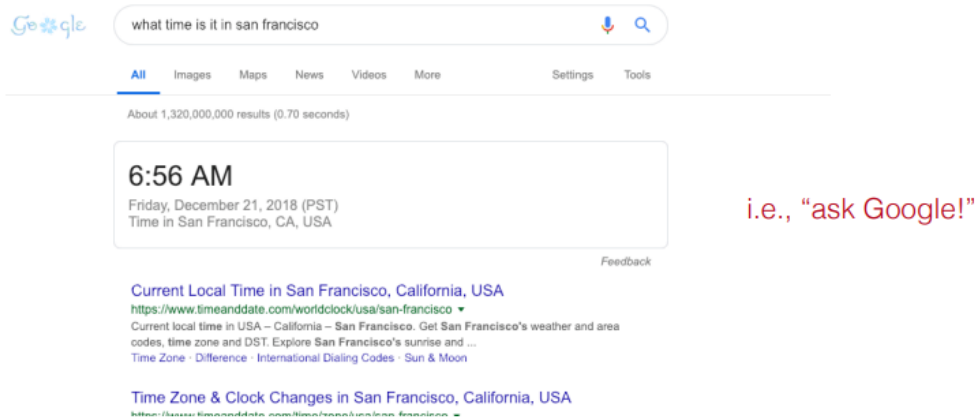


	English		German		Spanish		Italian		Dutch	
1	the	61,847	der	7,377,879	que	32,894	non	25,757	de	4,770
2	of	29,391	die	7,036,092	de	32,116	di	22,868	en	2,709
3	and	26,817	und	4,813,169	no	29,897	che	22,738	het/’t	2,469
4	a	21,626	in	3,768,565	a	22,313	è	18,624	van	2,259
5	in	18,214	den	2,717,150	la	21,127	e	17,600	ik	1,999
6	to	16,284	von	2,250,642	el	18,112	la	16,404	te	1,935
7	it	10,875	zu	1,992,268	es	16,620	il	14,765	dat	1,875
8	is	9,982	das	1,983,589	y	15,743	un	14,460	die	1,807
9	to	9,343	mit	1,878,243	en	15,303	a	13,915	in	1,639
10	was	9,236	sich	1,680,106	lo	14,010	per	10,501	een	1,637

(Top-10 most frequent terms in a large language sample)

Sintesi: recupero nello spazio vettoriale
- Rappresentare la query come un vettore (ponderato tf-idf)
- Rappresentare ogni documento come un vettore (tf-idf ponderato)
- Calcolare la somiglianza del coseno tra il vettore della query ed ogni vettore di documento
- Classificare i documenti in base alla somiglianza con la query
- Restituire all'utente i primi K risultati (ad es. K=10)

L'Information Retrieval è il compito di trovare documenti che sono rilevanti per una query (ad esempio, una parola chiave, un argomento, ...). La *risposta alle domande/Question Answering (QA)* può essere vista come un'applicazione dell'IR / Information Retrieval in cui l'interrogazione è in realtà una domanda.



Da questo principio, partono gli *IR-based dialog agents*, con tante conversazioni umani o machine simili all'uomo. In pratica, si cerca per una serie di risposte e dialoghi di campionamento, una risposa simile andando con TF-IDF a trovare la risposta più simile al contesto.

- A simple IR-based chatbot architecture:
  - Take user's $\mathbf{q}$ and find a TF-IDF similar $\mathbf{t}$ in the corpus C

    $\mathbf{q}$ = *"Do you like Doctor Who"*

    $\mathbf{t}$ = *"Do you like Doctor Strangelove"*
  - Return the response for the most similar turn t:

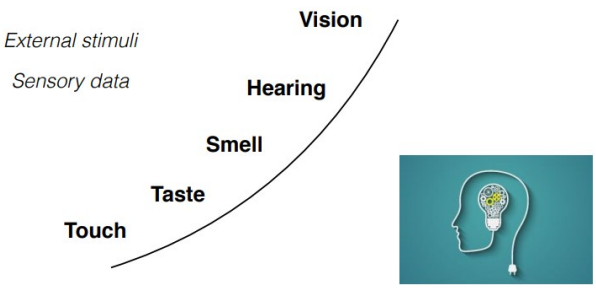
$$\mathbf{r} = response\left(\arg\max_{\mathbf{t}\in C} \frac{\mathbf{q}\cdot\mathbf{t}}{\|\mathbf{q}\|\cdot\|\mathbf{t}\|}\right)$$

- Pro:
▸ Molto facili da implementare
▸ Sono ottimi per le applicazioni ristrette e scrivibili.
▸ I grandi dati di solito aiutano
- Contro:
▸ Non capiscono realmente
▸ I chatbot basati su IR possono solo rispecchiare i dati di addestramento.

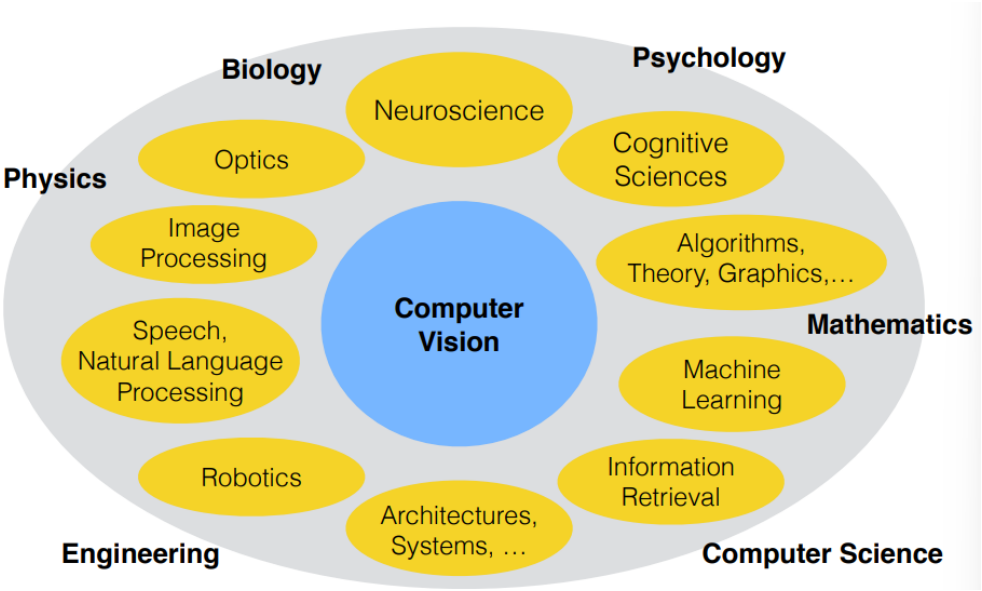
Un sistema di QA richiede una reale comprensione delle domande in linguaggio naturale. Deve anche fornire risposte in linguaggio naturale.

Teaching machines to see: a quest to visual intelligence

Partiamo da un concetto base:
- La percezione è la capacità di catturare, elaborare e dare senso alle informazioni che i nostri sensi ricevono e come vengono da noi elaborate.

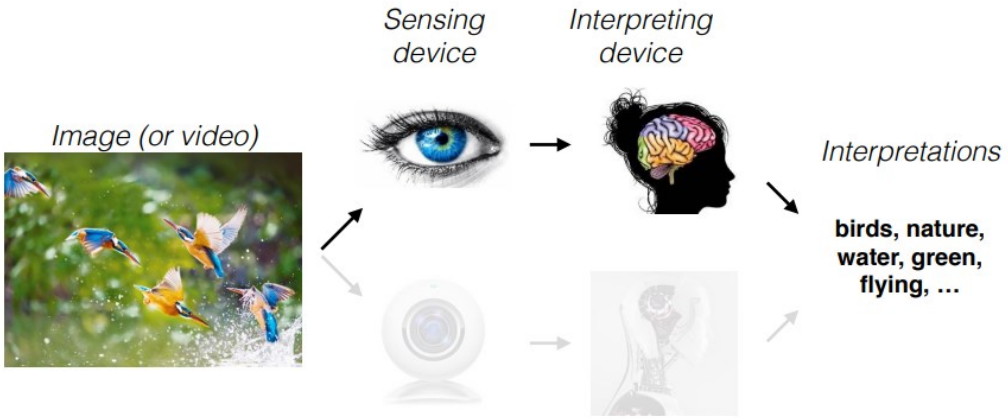


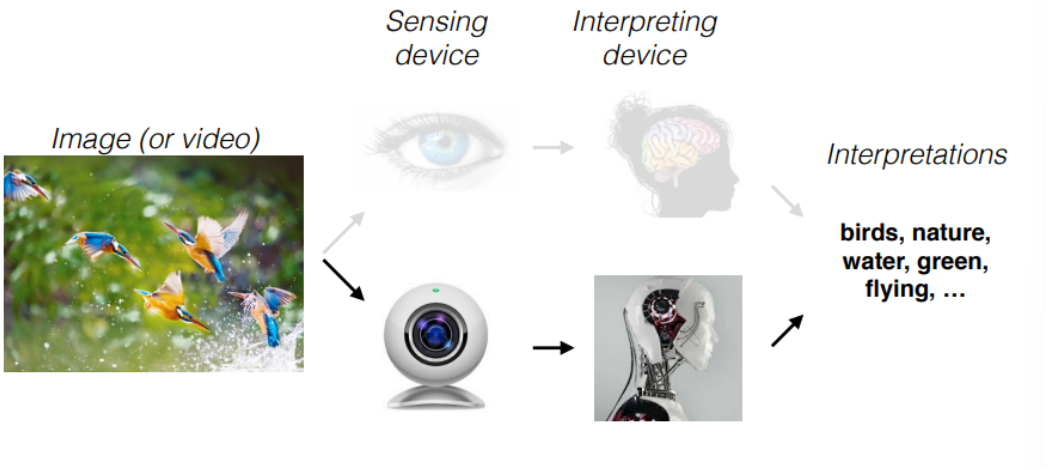
Cerchiamo di interpretare la computer vision.
Le tecniche e gli strumenti di computer vision, noti anche come visione artificiale o occhio digitale comprendono tutti quei processi utili a creare un modello approssimativo del mondo reale (3D) partendo da immagini bidimensionali (2D).



La computer vision quindi ha come scopo principale la riproduzione della vista dell'uomo, intesa non solo come riproduzione fotografica ma anche come comprensione di ciò che il sistema osserva.
Componenti ottiche, elettroniche e meccaniche costituiscono i sistemi di computer vision, che grazie a queste componenti sono così in grado di vedere immagini sia nello spettro della luce visibile che quelle ad infrarossi, ultravioletti e raggi X.

La maggior parte dei dati nel Web stesso è visuale:
partendo da un immagine/video, si cerca di comprendere un dispositivo rilevatore (sensing), un dispositivo interpretatore (interpreting) e quindi giungere ad un'interpretazione.



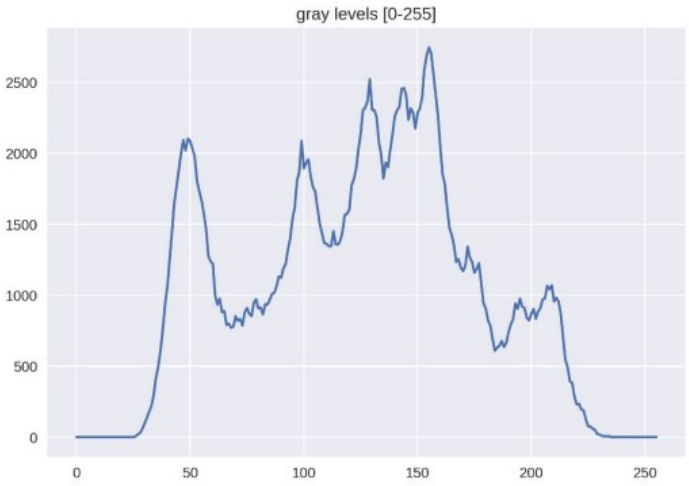


Altre sfide non da poco:

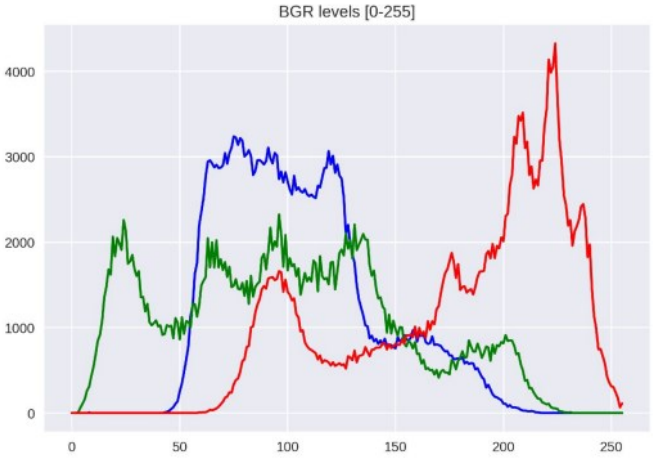
- Comprendere l'illuminazione degli ambienti
- Comprendere la dimensione e la posizione degli elementi in scala
- Comprendere la prospettiva
- Comprendere il punto di vista
- Comprendere la deformazione dello stesso soggetto
- Comprendere quando un soggetto è nascosto
- Comprendere quando si ha un ambiente affollato e si vuole trovare un soggetto in un mucchio

La nostra visione è permessa attraverso la corteccia celebrale, poi attraverso le varie parti collaborando per comprendere, attuare il riconoscimento, colorare l'immagine, comprendere il movimento.
Un importante premio Nobel in questo senso fu nel 1981.
(Hubel e Wiesel, autori della scoperta e vincitori del premio, dimostrarono che alcuni neuroni rispondevano solo alle informazioni provenienti da un singolo occhio, un fenomeno che definirono "dominanza oculare". È interessante notare che i neuroni sintonizzati su un particolare occhio si raggruppano in colonne anatomiche nella corteccia visiva del cervello. Le hanno chiamate "colonne di dominanza oculare". Hanno anche misurato come neuroni distinti rispondono a caratteristiche visive distinte, come l'orientamento di una linea proiettata su uno schermo o specifici modelli di linee. Questi esperimenti sono stati fondamentali per la comprensione dell'elaborazione visiva.)

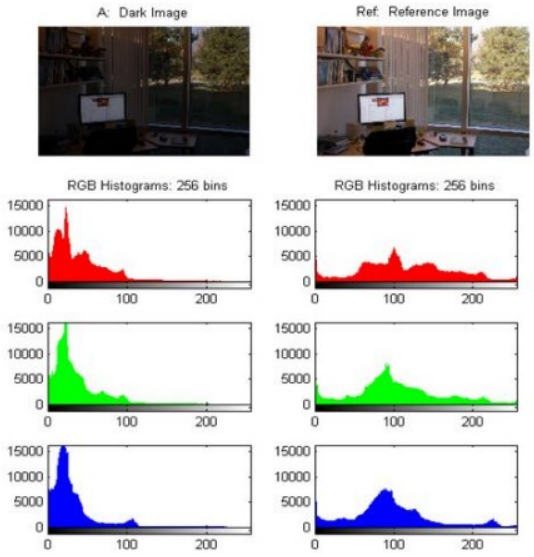
Un computer percepisce numeri e pixel, quando noi percepiamo un'immagine completa. Anche le immagini hanno diverse colorazioni; scala di grigi, immagini colorate. Ogni immagine contiene un numero discreto di pixel, solitamente in RGB, tale che siano scomponibili in matrici ed ogni pixel abbia delle proprie coordinate.
Per comprendere le immagini, si possono usare degli istogrammi, per esempio, per evidenziare il livello di grigio e la sua distribuzione su un'immagine:



Similmente, anche la distribuzione dei colori:



Anche le diverse parti di un'immagine hanno differenti gradazioni di colore:



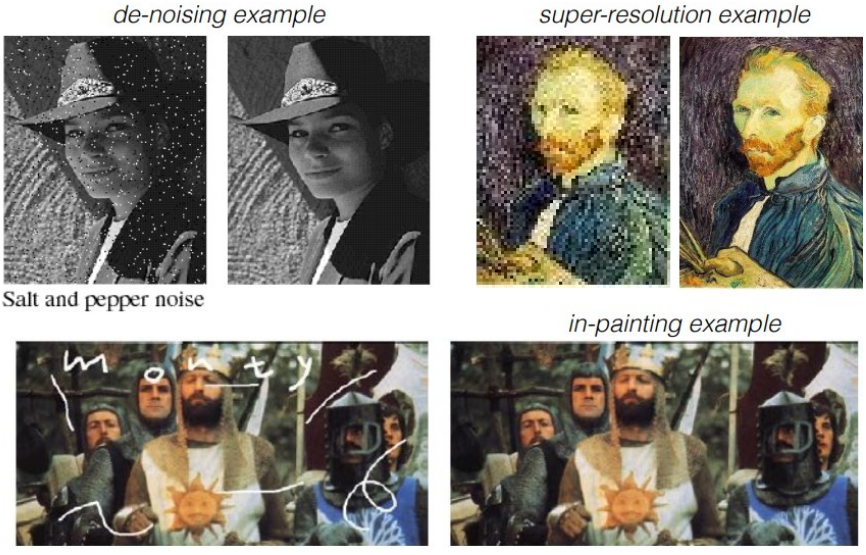
Un caso d'uso degli istogrammi può essere comparare una stessa immagine, per esempio per capire le gradazioni di colore; ciò non è robusto.

Formalmente, un'immagine può essere definita come una funzione *f* da R^2 ad R^M in cui:

- F(x, y) dà l'intensità alla posizione (x, y)
- È definita da un rettangolo con un range finito $f: \underbrace{[a,b] \times [a,b]}_{\text{Domain support}} \rightarrow \underbrace{[0,255]}_{\text{range}}$
- Un immagine a colori: $f(x,y) = \begin{bmatrix} r(x,y) \\ g(x,y) \\ b(x,y) \end{bmatrix}$

Filtraggio/filtering: formazione di una nuova immagine i cui valori dei pixel sono trasformati rispetto ai valori dei pixel originali. Quali sono gli obiettivi?

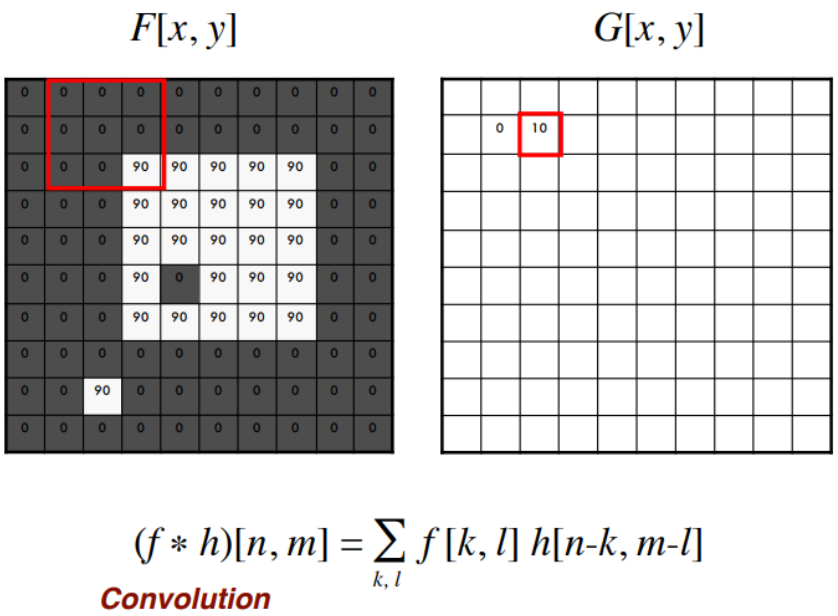
- estrarre informazioni utili dalle immagini
- trasformare le immagini in un altro dominio in cui è possibile modificare/aumentare le proprietà dell'immagine



Altro esempio utile: *moving average filter*. Come suggerisce il nome, il filtro a media mobile opera calcolando la media di un certo numero di punti del segnale di ingresso per produrre ogni punto del segnale di uscita. Utilizza un loop per i punti di output, calcola come accumulatore la somma degli input e completando la media dividendo per ogni punto.

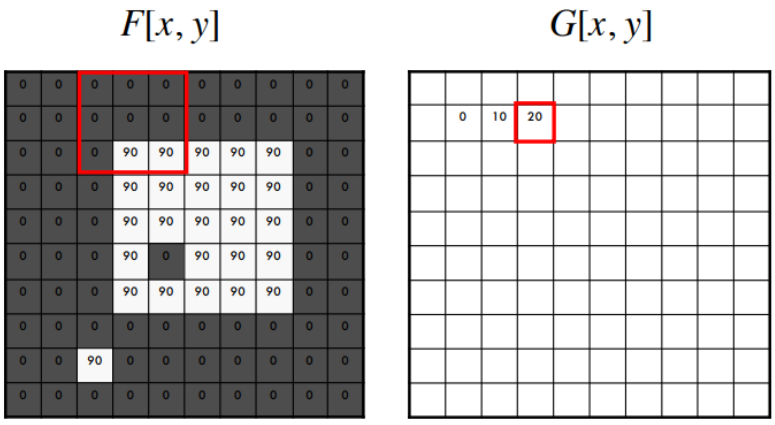
Scritto da Gabriel

Rimpiazza quindi ogni pixel con una media dei vicini e ottiene un effetto *smoothing* (rimuovendo difetti dalla chiarezza dell'immagine). Link: http://setosa.io/ev/image-kernels/



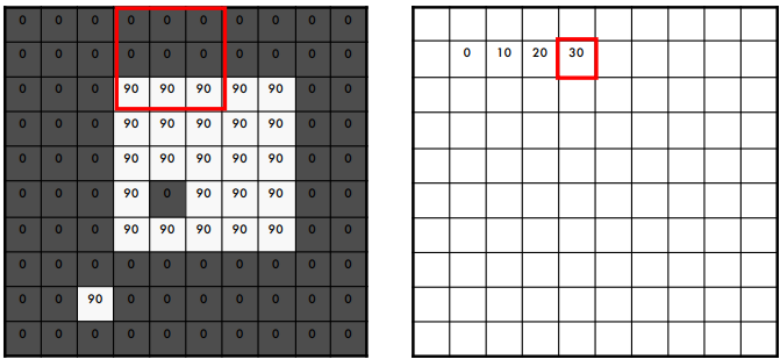
$$(f * h)[n, m] = \sum_{k, l} f[k, l]\, h[n-k, m-l]$$

In matematica (in particolare nell'analisi funzionale), la *convoluzione* è un'operazione matematica su due funzioni (f e g) che produce una terza funzione (f*g) che esprime come la forma di una sia modificata dall'altra. Il termine *convoluzione* si riferisce sia alla funzione risultante sia al processo di calcolo. È definita come l'integrale del prodotto delle due funzioni dopo che una è stata riflessa intorno all'asse y e spostata.



$$(f * h)[n, m] = \sum_{k, l} f[k, l]\, h[n-k, m-l]$$

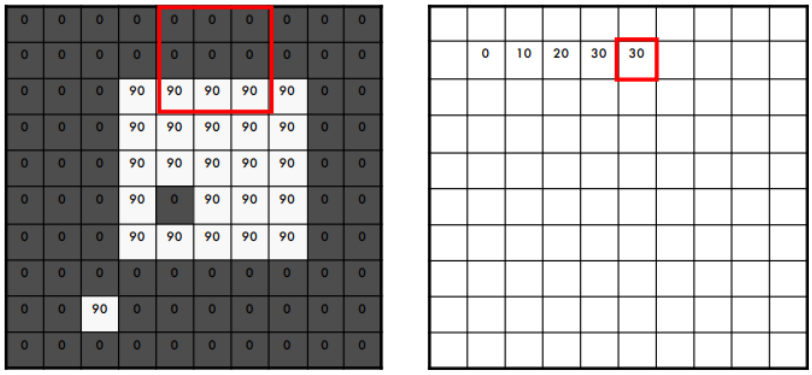
$F[x, y]$ $G[x, y]$



$$(f * h)[n, m] = \sum_{k, l} f[k, l]\, h[n-k, m-l]$$

Convolution

$F[x, y]$ $G[x, y]$



$$(f * h)[n, m] = \sum_{k, l} f[k, l]\, h[n-k, m-l]$$

Convolution

Il risultato assomiglia ad una cosa di questo tipo:

$F[x, y]$

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

$G[x, y]$

	0	10	20	30	30	30	20	10	
	0	20	40	60	60	60	40	20	
	0	30	60	90	90	90	60	30	
	0	30	50	80	80	90	60	30	
	0	30	50	80	80	90	60	30	
	0	20	30	50	50	60	40	20	
	10	20	30	30	30	30	20	10	
	10	10	10	0	0	0	0	0	

$$(f * h)[n, m] = \sum_{k,\, l} f\,[k, l]\; h[n\text{-}k, m\text{-}l]$$

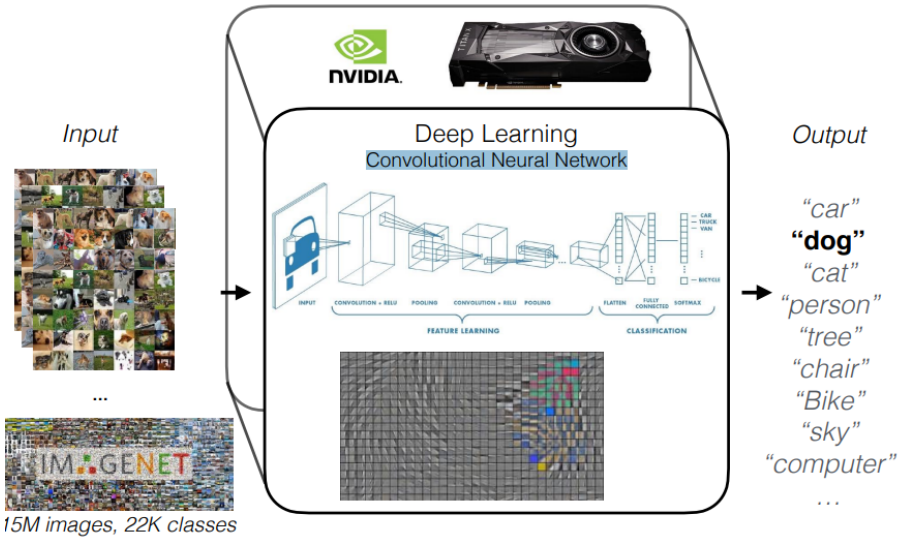
Convolution

Filtraggio delle immagini: idea principale

- Calcolo di una funzione del *local neighborhood* di ogni pixel dell'immagine (i pixel vicini a quello considerato)
  - La funzione è specificata da un "filtro" che indica come combinare i valori dei vicini
- Applicazioni del filtraggio delle immagini:
  - estrarre informazioni (bordi, angoli, blob, ...)
  - rilevare modelli/pattern (template matching)
  - de-noising (è l'operazione di rimozione del rumore da un'immagine, ad esempio l'applicazione di un rumore gaussiano a un'immagine), super-risoluzione (mettere a fuoco un'immagine migliorando una risoluzione), in-painting (ricostruzioni di parti di immagini/video)

La computer vision cerca di rispondere alla domanda: che oggetti ci sono in un'immagine?
Si parte ad esempio da un cane, per esempio riconoscendo i vari esemplari; partendo dal campione di un cane, si operava un confronto tra i singoli esemplari (exemplar-based object recognition) e decideva.
Funzionava abbastanza bene; non molto efficiente.
L'algoritmo di Machine Learning invece → prende un insieme di dati, confronta con ML e decide.

Da questa esigenza nasce il Deep Learning, riducendo di molto l'errore umano. Utilizzano le Convolutional Neural Networks per operare riconoscimenti da un grandissimo insieme di dati, come ad esempio:



- ConvNet (CNN): una sequenza di strati in cui ogni strato trasforma un volume di attivazioni in un altro attraverso una *funzione differenziabile.* Abbiamo tre tipi principali di strati:
    - Strato convoluzionale / convolutional layer
    - Strato di pooling / pooling layer
    - Strato completamente connesso / fully connected layer (esattamente come in una rete artificiale "normale").

Impileremo questi strati per formare un'architettura ConvNet completa.

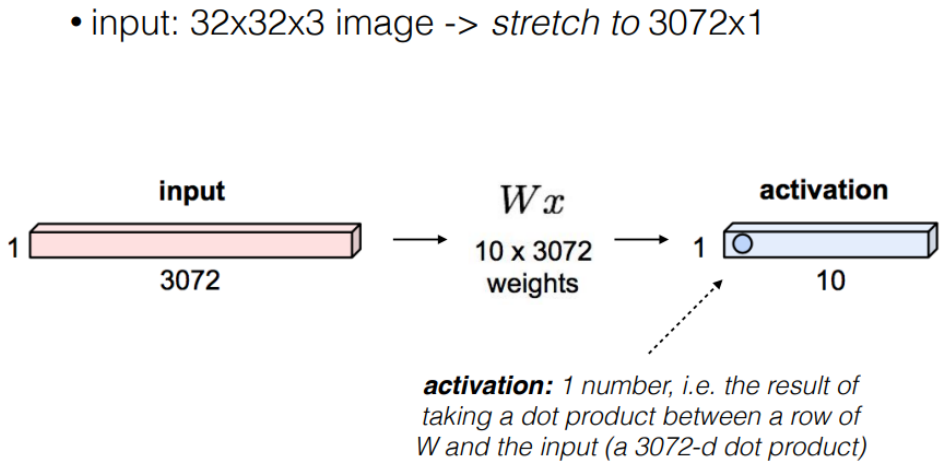
Un esempio di architettura per elaborazione di immagini:

- Lo strato INPUT [32x32x3] conterrà i valori grezzi dei pixel dell'immagine, in questo caso un'immagine di larghezza 32, altezza 32 e con tre canali di colore R,G,B.
- Lo strato CONV calcolerà l'uscita dei neuroni collegati a regioni locali nell'input, ciascuno dei quali calcolerà il prodotto di punti tra i propri pesi e una piccola regione a cui è collegato nel volume di input. Questo può risultare in un volume come [32x32x12] se si decide di utilizzare 12 filtri.
- Lo strato RELU applica una funzione di attivazione elementare, come la soglia max(0,x) a zero. Questo lascia invariata la dimensione del volume ([32x32x12]).
- Lo strato POOL esegue un'operazione di downsampling lungo le dimensioni spaziali (larghezza, altezza), ottenendo un volume come [16x16x12].
- Lo strato FC (ovvero fully-connected) calcolerà i punteggi delle classi, ottenendo un volume di dimensioni [1x1x10], dove ognuno dei 10 numeri corrisponde a un punteggio di classe, come ad esempio tra le 10 categorie di CIFAR-10. Come per le normali reti neurali e come suggerisce il nome, ogni neurone di questo strato sarà connesso a tutti i numeri del volume precedente.

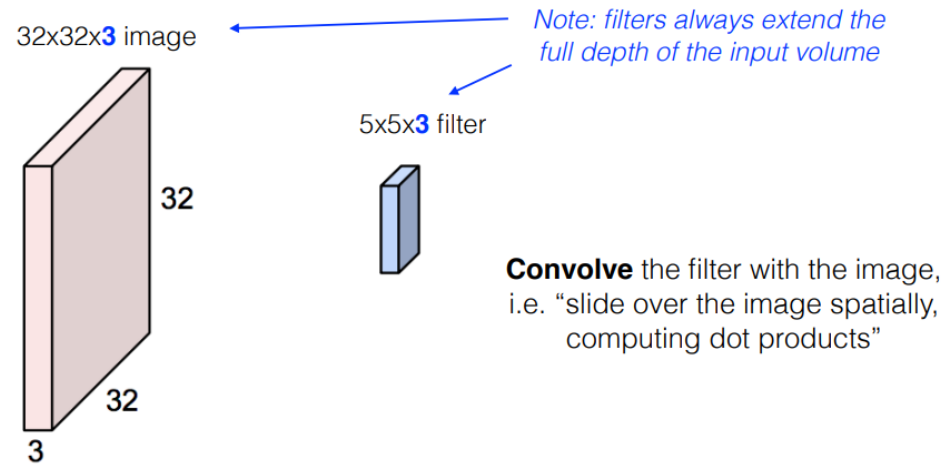
In questo modo, le ConvNet trasformano l'immagine originale strato per strato dai valori dei pixel originali ai punteggi finali della classe. Si noti che alcuni strati contengono parametri e altri no. In particolare, gli strati CONV/FC eseguono trasformazioni che sono funzione non solo delle attivazioni nel volume di ingresso, ma anche dei parametri (i pesi e le polarizzazioni dei neuroni). D'altra parte, gli strati RELU/POOL implementano una funzione fissa. I parametri degli strati CONV/FC saranno addestrati con discesa del gradiente in modo che i punteggi delle classi che la ConvNet calcola siano coerenti con le etichette del set di addestramento per ogni immagine.

In dettaglio:

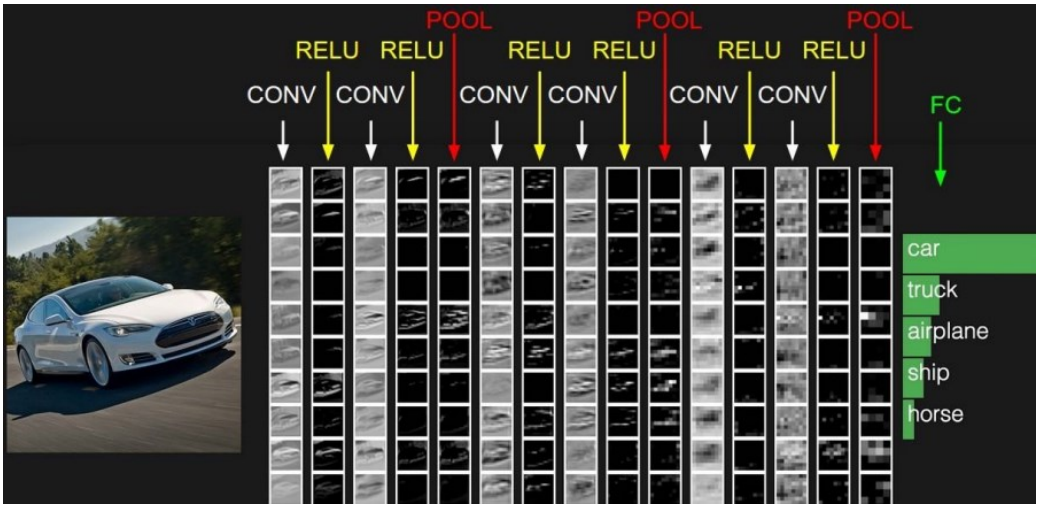
- Nel fully connected layer, si allunga l'immagine e si esegue un prodotto scalare



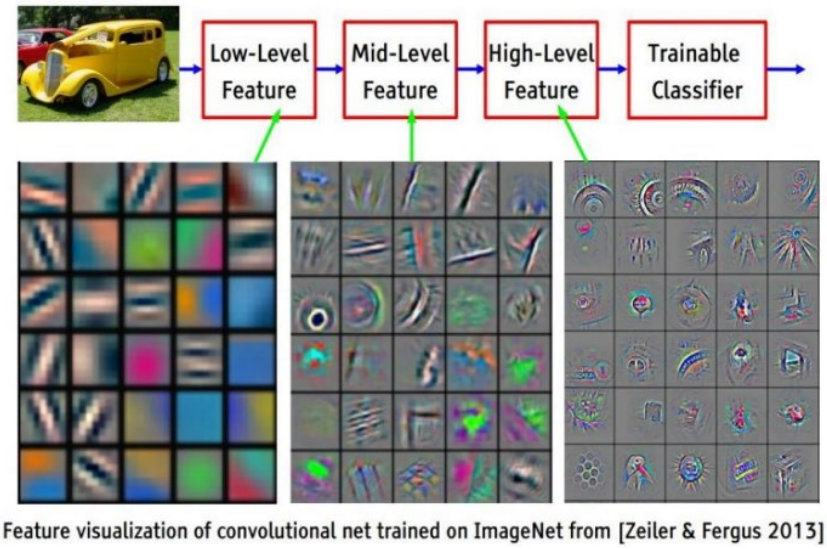
- Nel convolution layer, si preserva la struttura spaziale. Convolgere (integrare il prodotto tra una funzione e la seconda traslata di un valore, qui il prodotto scalare) il filtro con l'immagine, cioè "scorrere sull'immagine in senso spaziale", calcolando il prodotto scalare



Come detto, la usiamo per applicare lo smoothing alle immagine, sfruttando la funzione di Gauss. Tramite il convolution layer teniamo conto anche del bias nell'immagine. Un esempio visivo di architettura ConvNet:



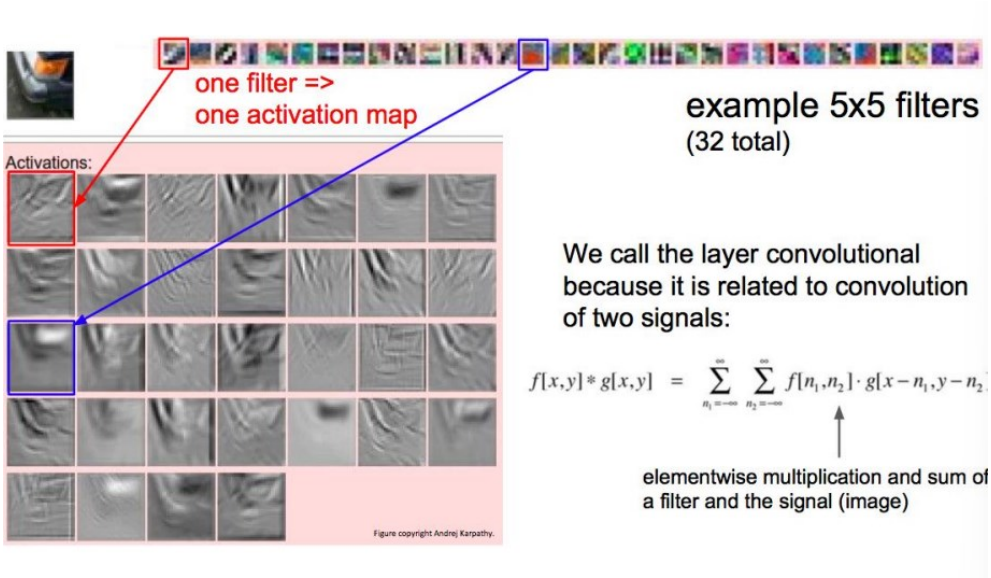
Le caratteristiche sono classificate a livelli e si considera un classificatore addestrabile:



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Scritto da Gabriel

Le funzioni di attivazione operano sui filtri, operando delle convoluzioni sulle immagini e, quindi, operando capendo sia i possibili rumori nell'immagine, svolgendo operazioni vettoriali ed operando classificazioni. Chiaro quindi come le Deep Networks implementino a piene mani tale principio, operando un miglioramento continuo di un algoritmo adattabile:



Tutto ciò è espandibile a tanti altri ambiti applicativi nelle Deep Networks, quali:

- Riconoscimento del volto, biometria
- Sorveglianza su scala urbana, Sicurezza
- Contenuti mediatici sintetizzati dall'intelligenza artificiale, "Deep fakes".
- Trasporto intelligente, auto a guida autonoma
- Intelligenza ambientale, mobilità, negozi e ospedali