NoiStudiamo

APPUNTI

Machine Learning

Regressione, classificazione, reti neurali, decision trees, random forest, SVM ed ensemble learning

A CURA DI

Paolo Volpini

Novembre 2026

INDICE

Indice degli argomenti

Gli argomenti principali degli appunti e la pagina in cui iniziano.

1. **Introduzione** 4
2. **Strumenti per il machine learning, primi algoritmi** 8
3. **Problema dello XOR, regressione** 14
4. **Overfitting, classificazione** 20
5. **Ottimizzazione, valutazione, algorithm design, Softmax** 28
6. **Reti neurali, decision trees** 38
7. **Decision trees, random forest** 48
8. **SVM** 58
9. **Ensemble learning** 62

Machine Learning

Paolo Volpini

00 - Introduzione

Ci sono determinate classi di problemi per cui non si hanno regole specifiche o logiche. Abbiamo allora dei dati da cui dobbiamo ricavare un output. Il processo pertanto prevede di fornire alla macchina dei dati per ottenere soluzione su questi.

Nel 1997 Tom Mitchell formulò la definizione classica di apprendimento automatico. Una sorgente **impara** da un'esperienza E rispetto ad una task T e ad una misura di performance P se la performance sulla task T misurata con P migliora con l'esperienza E. Ciò richiede definire in maniera formale le variabili del nostro problema.

Supponiamo che il nostro obiettivo sia distinguere email spam da non spam. Allora la performance è l'efficacia dell'algoritmo nel riconoscimento dello spam, il task è distiguere la mail o meno e i dati sono variabili random. La performance dell'algoritmo è misurata in virtù di una funzione $P(\{y^{(i)}\}, \{\hat{y}^{(i)}\})$. I valori saranno alti se l'algoritmo è accurato e basso se non lo è.
Con (i) indichiamo uno dei singoli **esempi**. Un generico esempio x appartiene all'insieme R^d, dove con d si identifica la dimensione. Pertanto l'esempio $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, ..., x_d^{(i)})$. Ogni singola componente del vettore prende il nome di **feature**. Gli attributi sono caratteristiche derivate dal tipo di problema. Ad esempio, nel caso del problema delle mail spam, possiamo definire due attributi:

- numero di errori ortografici;
- numero di parole chiavi presenti in un vocabolario.

A questo punto possiamo utilizzare una funzione $f(x)$ che ci permette di estratte tali informazioni. Questa funzione prende il nome di **funzione di rappresentazione**. Tale funzione permette di ottenere il vettore di feature $\overline{x}$. Per cercare un modello parametrizzato per risolvere il problema, è necessario trovare le feature necessarie. Pertanto la rappresentazione dei dati, ossia la funzione di rappresentazione, è fondamentale. Infine, queste funzioni buttano via molte informazioni originali.

Tipi di feature

Le feature possono essere di diverso tipo.

Categoriche

Si tratta di feature dai valori discreti. Ad esempio, “rosso”, “viola” e “blu” sono valori per la feature colore.

Continue

Si tratta di feature dai valori continui. Anche gli interi sono usati in questa accezione.

Tipi di task

Ci sono diversi tipi di task. I due che approfondiremo maggiormente sono **regressione** e **classificazione**.

Classificazione

Dato un vettore in input, vogliamo stabilire un’etichetta discreta. Abbiamo un set di esempi $X = \{x^{(1)}, x^{(2)}, ..., x^{(m)}\}$, e un output atteso $Y = \{y^{(1)}, y^{(2)}, ..., y^{(m)}\}$. Se abbiamo una classificazione binaria $0, 1$, allora $y^{(i)} \in \{0, 1\}$. Se invece abbiamo una k classificazione, allora $y^{(i)} \in \{0, ..., k-1\}$.

Il problema nel risolvere questo tipo di problemi è che non abbiamo una formulazione matematica analitica, ossia una formula. Quindi abbiamo un modello parametrizzato e cercheremo di modificare i parametri tramite una funzione $h_v(x^{(i)}) = \hat{y}^{(i)}$, dove v rappresenta i parametri del modello. Il valore di ritorno di questa funzione può non essere identico a quello atteso $y^{(i)}$. La nostra funzione h_v inoltre può essere una distribuzione di probabilità, che associa ad ogni possibile classe k del problema di classificazione un valore di probabilità che esprime quanto è probabile che l’output sia di una certa classe. Allora su questi valori di output possiamo operare un’operazione come argmax$(\hat{y})$, ossia prendere la classe corrispondente alla componente dal valore più alto.

Regressione

Il task di regressione può essere un task come prevedere il prezzo di una casa dalla posizione, dal numero di vani e dalla dimensione. Il nostro prezzo è un $y \in \mathbb{R}$. Abbiamo qui 3 parametri, per cui $x \in R^3$. La regressione in questo caso deve stimare una funzione facendo un **fitting dei dati** con il grafico di una possibile funzione. Ossia, si cerca una funzione che approssimi la relazione tra input e output minimizzando l’errore di predizione.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Supervised, unsupervised, reinforcement learning

I tipi di task visti per ora fanno parte della categoria di task risolvibili tramite **supervised learning**. Sono generalmente più semplici da risolvere rispetto a problemi **unsupervised**. La differenza risiede nel fatto che nel supervised learning sappiamo quali sono i valori di input e di output. In unsupervised learning il modello parametrizzato lavora con dati senza etichetta.

Esiste anche un'altra categoria di apprendimento, noto come **reinforcement learning**. Si prova a capire come risolvere un task e si cerca di capire una **policy**, ossia una strategia che definisce quale azione intraprendere in ogni stato per massimizzare il reward cumulativo. Ad ogni azione corretta per risolvere un task viene associato un reward positivo, mentre ad ogni azione sbagliata si associa un reward negativo. Alcune tecniche di Dynamic Programming sono utili per questi problemi.

In generale in ogni algoritmo di machine learning esistono due fasi:

- una fase di **training**, che prevede l'utilizzo di un una banca di dati per cercare il modello corretto per il problema da risolvere;
- una fase di **testing** o **infering**, dove a dati in input si associa un'inferenza. I parametri in questo caso sono bloccati.

Design Matrix

I nostri dati devono essere memorizzati in qualche modo. Allora dati i nostri vettori $x^{(1)}, ..., x^{(m)}$. Il nostro set di dati viene considerato come una matrice $X \in R^{m \times d}$. Per i nostri dati in output abbiamo bisogno di un altro vettore Y che contiene etichette dei campioni. Si ha che $Y \in A^{c \times d}$. Solitamente per Y si ha che $d = 1$. A è il set di etichette $\{0, ..., k-1\}$.

Approcci all'apprendimento

Ritorniamo al problema spam-no spam e consideriamo una delle feature $x_1 =$ "numero errori ortografici". Una possibile funzione è:

```
classifica(x):
    if x > theta then "Spam"
    else "No spam"
```


Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Il parametro `theta` dev'essere cercato in modo tale che funzioni per gran parte dei dati in input. Si utilizza allora un algoritmo di learning.

Un algoritmo di learning è una strategia per cercare i migliori parametri del modello. Un approccio naive per questo problema potrebbe essere quello di inizializzare `theta` a zero, e poi fornire il set di input alla funzione. A questo punto si ha un output $\{\widehat{y}^{(i)}\}$, e si misurano le performance $P(\{y^{(i)}\}, \{\widehat{y}^{(i)}\})$, ottenendo un valore di performance p_1. Poi si utilizza `theta = 1` e si ottengono nuove performance p_2. Cerchiamo allora il parametro che ci dà i valori più alti di performance.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

01 - Strumenti per il machine learning, primi algoritmi

Elementi di un algoritmo di machine learning

Ogni algoritmo di machine learning ha i seguenti elementi principali:

- i **dati**, che costituiscono i set di training, validation e testing;
- un **modello parametrizzato**;
- una **procedura di training** per addestrare il modello;
- una **cost** (o **loss**) function per affinare i parametri del modello;
- misure di valutazione, ossia la funzione P.

I dati

I dati vengono divisi in tre insiemi:

- il **training set**;
- il **validation set**;
- il **testing set**.

Questi tre insiemi hanno **intersezione vuota**, in modo tale che l'algoritmo non incontri nuovamente dei dati su cui ha già lavorato per ottimizzare i parametri del modello. Il modo in cui il dataset viene diviso è in maniera randomica tra dati di training, di validation e di test. Bisogna tuttavia stare attenti a questo campionamento. Se ad esempio abbiamo un problema di classificazione con due classi $0, 1$, e utilizziamo l'80% dei nostri dati per training e 20% dei dati per testing, dobbiamo prendere in maniera equa dati di entrambe le classi per training e testing. Altrimenti questo causa un **bias** nei nostri dati. Vedremo meglio come il bias possa impattare il nostro modello.

Potrebbe essere necessario **normalizzare i dati**. La normalizzazione dei dati è necessaria quando il dominio delle feature non è costante. Ad esempio, se come dati abbiamo "età" e "stipendio annuo",

osserviamo che le scale di queste feature sono molto diverse. Questo porta ad uno **sbilanciamento di importanza**: "stipendio annuo", che può avere valori molto alti, influenzerà maggiormente sull'errore e avrà maggiore importanza nella fase di learning, dominando completamente "età".

Un'idea potrebbe essere quella di assegnare a priori un peso molto basso per "stipendio annuo", così da poter pesar di meno la sua importanza dati i valori alti. Ma ha molto più senso **scalare i range di valori** in modo tale da far sì che la procedura di apprendimento possa lavorare con valori che hanno, in primis, tutti la stessa importanza.

Esistono diversi tipi di normalizzazione di dati. La **min-max** normalization normalizza i dati come segue:

$$\frac{x_j - \min(x_j)}{\max(x_j) - \min(x_j)}$$

Ogni feature viene quindi normalizzata nel range $[0, 1]$. L'obiettivo in questo caso è mantenere la proporzione: valori molto alti saranno prossimi a 1, valori molto bassi a 0.

La min-max normalization, tuttavia, non è ideale quando il range è molto ampio e la distribuzione è fortemente asimmetrica: ne seguirebbe che, con alcuni valori estremi, molti valori sarebbero prossimi allo zero. Quando la distribuzione dei dati è sbilanciata, conviene esprimere tutto in virtù di **media** e **deviazione standard**. Ciò è quello che fa la **z-score normalization**:

$$\frac{x_j - \mu_j}{\sigma_j}$$

Dove σ è la deviazione standard e μ è la media. Questa trasformazione:

- sposta lo spazio dei dati di un valore pari alla media, centrandolo in zero;
- modifica ogni valore in modo tale da ottenere le informazioni relative alla **distanza dalla media espressa in unità di deviazione standard**. In questo modo i valori estremi sono comunque modellati, ma vengono espressi in termini di distanza dalla media.

Queste normalizzazioni sono spesso necessarie in virtù dell'algoritmo di machine learning impiegato.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Iperparametri

Alcuni modelli potrebbero avere degli **iperparametri**. Un iperparametro è un parametro che non dipende solamente da un modello, ma anche dal tipo di problema che stiamo risolvendo. Per tale motivo si utilizza il **validation set**. Una volta stabiliti i parametri del modello, un ulteriore apprendimento viene eseguito sul validation set per stabilire gli iperparametri. Un esempio è il numero di layer nascosti di una rete neurale.

$k-$fold cross validation

Talvolta è possibile che i dati che abbiamo a disposizione non sono molti. Pertanto per ottimizzare l'apprendimento si può adottare una strategia nota come $k-$**fold cross validation**.
Prendiamo i dati e dividiamoli in due gruppi: training e test. Il set di training viene diviso in k fold (gruppi), in modo tale che $k-1$ fold siano utilizzati come training e il fold rimanente sia utilizzato come validation set. Si ottiene ad ogni iterazione (in totale k) un risultato per iperparametro, parametri e loss function. Si sceglie allora l'iperparametro che minimizza la loss function. Infine, si esegue il testing su tutto il training set, ottenendo i parametri. Si può allora passare alla fase di test.

Overfitting, underfitting

Con **overfitting** si intende la situazione in cui il modello è così complesso che fa ciò che vuole sui dati: sui dati di training ha errore basso, mentre sui dati di test ha errore alto (errore alto = valutazione alta della loss function). Significa che il modello è troppo capacitivo.

Con **underfitting** si intende il caso in cui il modello non è molto capacitivo. Abbiamo un modello che durante il training ha errore alto. Il modello scelto quindi non riesce allora a esprimere bene i risultati ottenuti.
Abbiamo allora un range di modelli dal meno complesso al più complesso. Se il modello è troppo poco complesso andiamo in underfitting, mentre se è troppo complesso andiamo in overfitting.

Bias, variance, tradeoff

Ci sono due sorgenti di errore che impediscono ad un algoritmo di apprendimento di ottenere risultati corretti al di fuori della fase di testing: il **bias** error e il **variance** error.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

1. Il **bias** error è un errore dovuto ad una assunzione erronea nell'algoritmo di apprendimento. Tale errore impedisce il riconoscimento di importanti relazioni da parte dell'algoritmo di machine learning.
2. Il **variance** error è un errore dovuto alla sensibilità alle piccole variazioni nel training set. Un'alta varianza spinge un algoritmo di machine learning a modellare anche il rumore.

Essenzialmente un alto bias error è causa di **underfitting**, mentre un alto variance error è causa di **overfitting**. Idealmente vorremmo scegliere un modello che catturi le regolarità del training set, ma che riesca a generalizzare bene i dati non conosciuti (testing set). Questa situazione, che abbiamo introdotto precedentemente come "scegliere il modello di mezzo", è chiamata **bias-variance trade-off**.

Primi tentativi di Machine Learning: McCulloch-Pitts

I primi tentativi di machine learning simulavano il funzionamento di un neurone. Il nostro cervello è infatti una rete neurale che compie scelte in virtù dell'attivazione di determinati neuroni.

Nel 1943 McCulloch-Pitts realizzarono il primo neurone computazionale. Si hanno d input $x_1, ..., x_d$ binari, una **threshold** θ e un output che dirama verso altri neuroni. I nostri input possono essere **eccitatori** o **inibitori**. Se sono inibitori, vengono indicati con un pallino.

Siano $x_1, ..., x_j$ eccitatori e $x_{j+1}, ..., x_d$ inibitori. Se $j \geq 1$ e almeno un input inibitore è uguale a 1, allora il neurone restituisce 0. Altrimenti, si calcola:

$$z = \sum_{i=1}^{d} x_i$$

e se $z \geq \theta$ l'output è 1, altrimenti è 0.

Questo modello riesce a computare funzioni, come le funzioni logiche AND e OR. Tuttavia porte logiche come lo XOR non possono essere realizzate: le funzioni che possiamo computare sono solo quelle **linearmente separabili**, ossia che tramite una retta dividono i risultati per problemi di classificazione. Inoltre, avere input binari può essere limitante.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Percettrone

Non abbiamo inoltre procedure automatiche per stabilire il valore θ di threshold, e non è detto che tutte le feature abbiano lo stesso peso.

Il **percettrone** presuppone la seguente architettura. Siano $x_1, ..., x_d$ gli input. Ad ogni input è associato un peso θ. Sia θ_0 una threshold. Il modello è un modello di classificazione con output nel set $\{0, 1\}$. Ogni input viene normalizzato nel range $[0, 1]$.

Il modello risolve il problema dell'input discreto, permettendo di computare un set più complesso di funzioni. Viene anche inclusa una **procedura di learning automatica**, ossia si ha un modo per stabilire i parametri $\theta_0, ..., \theta_d$. La funzione che computa il percettrone è:

$$f(x_1, ..., x_d) = \begin{cases} 1 & \text{se } \sum_{i=1}^{d} x_i \cdot \theta_i \geq \theta_0 \\ 0 & \text{altrimenti} \end{cases}$$

Per comodità introduciamo la variabile $x_0 = 1$ (SEMPRE uguale ad 1) il cui peso è la threshold θ_0. La funzione quindi sarà uguale a uno se:

$$\sum_{i=0}^{d} x_i \cdot \theta_i \geq 0$$

Questo permette di scrivere la funzione in maniera matriciale:

$$f([\theta^T \vec{x} > 0])$$

θ^T è il vettore di parametri $[\theta_0, ..., \theta_d]$, ottenendo quindi un prodotto riga per colonna. A livello di notazione, possiamo anche scrivere $\theta \times \vec{x}^T$: l'idea è quella di utilizzare il **calcolo matriciale** per esprimere le combinazioni lineari. Inoltre, il calcolo matriciale permette di eseguire computazioni in parallelo. Infine, possiamo sostituire $\geq$ anche con $>$ per evitare valori sul **decision boundary**, ossia sul bordo decisionale.

Supponiamo di avere le feature x_1, x_2 e i parametri $\theta_0, \theta_1, \theta_2$. La soluzione che ci dà l'algoritmo non sarà altro che costruire una linea. Infatti basta:

$$\theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 \geq 0$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

per ottenere 1. Questo significa che:

$$x_2 = -\frac{\theta_1}{\theta_2}x_1 - \frac{\theta_0}{\theta_2}$$

Pertanto il modello del percettrone è un modello **lineare**: ragioniamo solamente su **spazi linearmente separabili**.

Algoritmo di apprendimento

Possiamo partire inizializzando i pesi $\theta_0, ..., \theta_d$ randomicamente. Computiamo allora per ogni $x^{(i)}$ il valore $\hat{y}^{(i)}$ (input e output). Abbiamo due casi:

- $y^{(i)} = \hat{y}^{(i)}$;
- $y^{(i)} \neq \hat{y}^{(i)}$;

Nel primo caso non abbiamo necessità di aggiornare i pesi. Ma nella seconda condizione l'input viene classificato male. Resta da stabilire come aggiornare i pesi.
Se $y^{(i)} = 1$, allora l'output che abbiamo ottenuto è pari a 0. Pertanto **bisogna aumentare i pesi**. Si ottiene infatti che la threshold non può essere superata. Se invece $y^{(i)} = 0$, l'output che abbiamo ottenuto è pari ad 1 e dobbiamo decrementare i pesi. Possiamo allora ripetere questi passi fino a che non ci sono errori.

Aggiorniamo i pesi allora in modo tale che il nuovo termine θ_j è uguale a:

$$\theta_j \leftarrow \theta_j + (y^{(i)} - \hat{y}^{(i)}) \cdot x_j$$

I pesi vengono quindi decrementati quando abbiamo che il risultato atteso è 0, e aumentati nel caso contrario.

La versione successiva dell'algoritmo sostituisce il risultato ottenuto $\hat{y}^{(i)}$ con il valore $\sum_i \theta_i \cdot x_i$, modulando la modifica del peso in virtù del risultato ottenuto. Questo modello prende il nome di **modello ADALINE** ed è usato per la **regressione lineare**.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

02 - Problema dello XOR, regressione

Abbiamo visto che tramite percettrone siamo in grado di risolvere problemi i cui risultati sono linearmente separabili. Analizziamo ora il problema dello XOR. Infatti lo XOR è l'operazione il cui risultato è pari ad uno se solo uno dei due input è uguale ad 1 e zero altrimenti.

Date le due variabili x_1, x_2, abbiamo allora che:

$$x_1\theta_1 + x_2\theta_2 > \theta_0$$

Vediamo subito che il problema non può essere risolto. Avremo infatti che:

$$\begin{cases} 1\theta_1 + 0\theta_2 > \theta_0 \\ 0\theta_1 + 1\theta_2 > \theta_0 \\ 0\theta_1 + 0\theta_2 < \theta_0 \\ 1\theta_1 + 1\theta_2 < \theta_0 \end{cases}$$

Dalla penultima deduciamo che $\theta_0 > 0$. Dalle prime due deduciamo che $\theta_1, \theta_2 > \theta_0$. Per cui:

$$\theta_1 + \theta_2 > 2\theta_0$$

Ma dall'ultima si ha:

$$\theta_1 + \theta_2 < \theta_0$$

E quindi otteniamo che:

$$2\theta_0 < \theta_1 + \theta_2 < \theta_0$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Il che dimostra formalmente che il modello del percettrone non permette la computazione dello XOR. Tuttavia, possiamo modificare la struttura in modo tale che le feature x_1, x_2 siano pesati positivamente per un percettrone e negativamente per un altro. Ossia, abbiamo 3 percettroni:

1. il primo percettrone computa la funzione $f_1[x_1 - x_2 > 0.5]$;
2. il secondo percettrone computa la funzione $f_2[-x_1 + x_2 > 0.5]$;
3. il terzo percettrone computa la funzione $f[f_1 + f_2 > 0.5]$.

I due percettroni trasformano lo spazio di input, modificando il punto $(0, 0)$ in $(0, 0)$, $(1, 1)$ in $(0, 0)$ e i restanti invariati. Pertanto il punto $(1, 1)$ viene collassato nel punto $(0, 0)$.

Regressione

La regressione cerca una funzione $h_\theta : R^{d_1} \to R^{d_2}$, dove $d_1, d_2 \geq 1$. Assume tre versioni:

1. **regressione lineare**. h_θ mappa R in R, quindi mappa punti in uno spazio monodimensionale in un altro punto in uno spazio monodimensionale;
2. **regressione multipla**. Cerchiamo $h_\theta : R^{d_1} \to R$, con $d_1 > 1$;
3. **regressione multivariata**. Apprendiamo $h_\theta : R^{d_1} \to R^{d_2}$, con $d_1, d_2 > 1$. Nella regressione multivariata, ogni componente di output è il risultato di una multiple regression.

Regressione lineare

Consideriamo la feature x_1 e i corrispondenti y. Vogliamo associare ad ogni punto x_1 il corretto valore y. La funzione h_θ è del tipo:

$$h_\theta(x) = \theta_0 + x_1\theta_1$$

L'obiettivo è trovare i valori opportuni per θ_0, θ_1. Abbiamo bisogno di una funzione di loss; più precisamente, possiamo utilizzare come funzione di loss una funzione che tenga conto della distanza tra l'output stimato e l'output desiderato. Pertanto vogliamo minimizzare:

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

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

In modo tale da considerare solo il valore della distanza. Questa è una leggera modifica alla formula dello **scarto quadratico medio**. Inoltre, la differenza ricorda il calcolo per la modifica del peso nel modello Adaline. I valori x, y sono dei **dati fissi** e non parametri di funzione.

Il grafico di $\mathcal{J}(\theta)$ è una parabola convessa. Vogliamo trovare il minimo di questa parabola iterativamente. Possiamo allora randomicamente scegliere (per ora non consideriamo θ_0) un valore θ_1 al tempo 1, poi un altro al tempo 2 e così via. Stabiliamo pertanto un numero di iterazioni, osserviamo il grafico della funzione di errore e se non abbiamo ancora trovato il minimo, possiamo ricominciare dall'ultimo parametro trovato.
Vogliamo allora capire quando incrementare il parametro e quando diminuirlo. Chiaramente notiamo che preso un punto θ_1^1, possiamo stabilire il valore della tangente al grafico nel punto e osservare il segno della derivata:

- se la derivata è negativa, dobbiamo incrementare il valore;
- se la derivata è positiva, dobbiamo decrementare il valore.

Allora possiamo dire che ad ogni passo dobbiamo:

$$\theta_1 \leftarrow \theta_1 - \alpha \cdot \frac{d\mathcal{J}(\theta)}{d\theta_1}$$

In linea teorica potremmo calcolare la derivata e azzerarla, ma questo algoritmo permette di ragionare anche con n variabili. Questo algoritmo prende il nome di algoritmo della **discesa del gradiente**.

Multiple regression

Dato un input di dimensione d_1, dobbiamo istanziare un modello di $d_1 + 1$ parametri:

$$h_\theta(x_1, x_2, ..., x_d) = x_0\theta_0 + x_1\theta_1 + ... + x_d\theta_d$$

La loss function $\mathcal{J}(\theta)$ è sempre utilizzabile. In particolare, per l'algoritmo della discesa del gradiente eseguiamo i seguenti passi:

1. imposta in maniera randomica il vettore dei parametri $\theta = (\theta_0, ..., \theta_d)^T$;
2. calcola la derivata parziale rispetto ai vari parametri di $\mathcal{J}(\theta)$;
3. aggiorna i parametri, in modo tale che $\theta_j \leftarrow \theta_j - \alpha\frac{\delta\mathcal{J}(\theta)}{\delta\theta}$;
4. ripeti fino a quando il criterio di stop non è raggiunto.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

L'aggiornamento dei parametri va fatto simultaneamente. Al tempo t si utilizzano tutti i parametri.

Calcoliamo adesso la derivata prima per la funzione costo. Lo faremo per una variabile θ_j. La funzione costo è modificata in modo tale che $\frac{1}{m}$ non sia più presente. Lasceremo tuttavia $\frac{1}{2}$.
Abbiamo che:

$$\mathcal{J}'(\theta) = \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j$$

x_j è la derivata della funzione interna h_θ per il fatto che stiamo facendo la derivata rispetto alla variabile θ_j.

Da notare come il feature scaling che abbiamo visto precedentemente (ossia le normalizzazioni) permetta di velocizzare il procedimento della discesa del gradiente: lo scaling delle feature, infatti, impedisce ad un peso di essere aggiornato in maniera differente rispetto ad un altro a causa dell'input di diverso ordine di grandezza.

Output non lineari: regressione polinomiale

Può succedere che l'output non sia lineare. Il modello che abbiamo visto per ora è un modello lineare. Per output quadratici, il modello lineare va in underfitting a causa di un errore di bias, ossia l'assunzione che l'output sia lineare. Anche nel caso di d_1 dimensioni si presenta il problema: se l'output non è lineare rispetto alle feature, abbiamo underfitting.

Possiamo cambiare il modello tenendolo lineare nella trasformazione delle feature. Ossia, trasformando le feature, possiamo riutilizzare quanto visto. Cerchiamo allora una funzione $\phi(x)$ che dia in output (x') in modo tale che il modello non faccia underfitting. Generalmente si utilizzano i **polinomi** e si parla di **regressione polinomiale**. Estendiamo lo spazio delle feature utilizzando le feature stesse:

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

L'esempio precedente estende il polinomio di grado 1 in un polinomio di grado 2, includendo le combinazioni tra le feature.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

A questo punto lavoriamo linearmente con le nuove variabili e quindi computiamo $h_\theta(\phi(x))$.

Se dopo una trasformazione di spazio andiamo in overfitting, è possibile semplificare il modello azzerando alcuni parametri θ. Per fare questo è necessario una funzione costo che penalizza avere parametri grandi. Ossia si cerca un elemento matematico che mantenga i parametri il più piccolo possibile.
Nella espressione $\mathcal{J}(\theta)$ allora aggiungiamo il seguente termine:

$$\sum_{j=1}^{d_1} \theta_j^2$$

In modo tale che la funzione da minimizzare consista anche nel minimizzare i parametri. Possiamo inoltre aggiungere un **iperparametro** λ che pesa questa sommatoria. Tale termine allora diventa:

$$\frac{1}{2}\lambda \sum_{j=1}^{d_1} \theta_j^2$$

La derivata di questo termine sarà:

$$\lambda \cdot \theta_j$$

Pertanto dobbiamo includere dei validation set per:

- scegliere il grado di trasformazione;
- stabilire correttamente λ.

Una domanda potrebbe riguardare l'assenza di θ_0 nel termine di regolarizzazione. La risposta risiede nel fatto che l'interesse principale è la penalizzazione dei parametri che determinano la complessità del modello, non la bias.

Valutazione

Come funzione per la valutazione del nostro modello, possiamo utilizzare sempre uno scarto quadratico medio o semplicemente la media del valore assoluto delle differenze.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Comparazione di due modelli

Due modelli vengono comparati su due performance. Tuttavia le stiamo valutando su un solo valore. Esiste una misura chiamata **roc curve**. Dati x_m input ed ϵ_m errori per ogni input, possiamo ordinarli ottenendo la lista ordinata $\epsilon'_1, ..., \epsilon'_m$. Ci interessa generalmente osservare errori il più piccolo possibile. Costruiamo un grafico tale che ad ogni valore ϵ' associamo la quantità di input per cui l'errore è minore o uguale a tale valore ϵ'. Tale curva sarà chiaramente monotona crescente, ma ci interessa di questa curva la **ripidità**. Vogliamo infatti che questa curva cresca molto rapidamente: ciò significa che i test hanno orientativamente lo stesso livello di errore, nonché anche un errore basso.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

03 - Overfitting, classificazione

Overfitting

Abbiamo detto che l'overfitting si verifica quando si ha un errore basso nel training e un errore alto nel testing. Uno dei problemi per cui si verifica l'overfitting risiede nell'utilizzo di un modello troppo capacitivo.

Abbiamo visto che la formulazione corretta della funzione $\mathcal{J}(\theta)$:

$$\mathcal{J}(\theta) = \frac{1}{2}(\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{d_1}\theta_j^2)$$

In questa formula abbiamo una prima parte nota come **dataterm** e una seconda parte nota come **regolarizzazione**, che penalizza l'utilizzo di parametri molto alti.

Potrebbe capitare che il numero di campioni è molto più piccolo rispetto al numero di feature. Quando ciò accade, si può ricadere in overfitting: il motivo è dovuto al fatto che il modello si abitua ad un set di dati che potrebbe comparire come rumoroso.

Classificazione

La classificazione può essere:

- **binaria**. Si ottiene un output $y \in \{0, 1\}$;
- **multiclasse** o $k-$**classe**. Si ottiene un output $y \in \{0, ..., k-1\}$.

Vogliamo allora trovare un modello che ci permette di stabilire le **decision boundaries**, ossia i "confini" di decisione.

Abbiamo bisogno di diversi strumenti:

- modello parametrico;

- funzione di loss;
- algoritmo di learning;
- misure di valutazione.

Chiaramente non vogliamo overfitting.

Regressione e classificazione

Supponiamo di avere un problema in cui abbiamo la profondità di un neo, per cui più un neo è profondo maggiore è la pericolosità.

Si potrebbe allora fittare un regressore di questi dati, e porre una soglia 0.5 in modo tale che tutto ciò che è minore di 0.5 viene filtrato come un neo normale, mentre se è superiore viene filtrato come neo pericoloso.
Il problema è che osservato un certo pattern, la retta può assumere una determinata pendenza. Ma introducendo un nuovo pattern, è possibile che la pendenza di tale retta venga modificata, classificando come normali pattern filtrati precedentemente come pericolosi.

Modello parametrico per la classificazione: regressore logistico

Il modello di classificazione che utilizzeremo è il **regressore logistico**. Il regressore logsitico considera la relazione tra una o più feature in input per determinare un output dicotomico.
Ci serve una funzione che sia **derivabile**, in modo tale da poter applicare nuovamente l'algoritmo di discesa del gradiente. Più precisamente, la funzione dev'essere limitata:

$$0 \leq h_\theta(x) \leq 1$$

Una funzione possibile è la **funzione sigmoidea**:

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

dove z è una combinazione lineare delle feature:

$$z = \theta^T x$$

Tale funzione è l'**inversa** della funzione logit:

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

$$\text{logit}(p) = \ln(p) - \ln(1-p) = \ln \frac{p}{1-p}$$

p è la probabilità che un campione sia di classe 1, mentre $1-p$ la probabilità che sia di classe 0 (o viceversa). Questa funzione assume valori nei reali, ma possiamo invertirla per ottenere una funzione che ha come dominio $\mathbb{R}$ e come codominio $[0,1] \subset \mathbb{R}$. Il rapporto $\frac{p}{1-p}$ è noto come **odds**, ed è un valore reale che esprime una misura di probabilità. L'idea allora è quella di utilizzare un regressore per predire i valori di odds e applicare l'inversa della funzione per ottenere una stima di probabilità.

La funzione sigmodea è derivabile. Il nostro modello $h_\theta(x)$ allora è:

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}} : R^d \to [0,1] \subset \mathbb{R}$$

La derivata parziale rispetto a z della funzione $\sigma(z)$ è:

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

La funzione sigmodea è correlato alla tangente iperbolica, in quanto l'espressione della tangente iperbolica è:

$$\tanh x = \frac{1-e^{-2x}}{1+e^{-2x}}$$

Si ha che la funzione sigmodea è in relazione con la tangente iperbolica:

$$2\frac{1}{1-e^{-x}} - 1 = \tanh \frac{x}{2}$$

Pertanto in alternativa alla funzione sigmodea è possibile utilizzare la seguente funzione:

$$\sigma(z) = \frac{1}{2} + \frac{1}{2}\tanh(\frac{z}{2})$$

Criterio di classificazione

Siamo in grado di classificare seguendo la condizione:

$$h_\theta(x) \geq 0.5 \iff \theta^T x \geq 0$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

E viceversa:

$$h_\theta(x) < 0.5 \iff \theta^T x < 0$$

In questo modo siamo in grado di definire le classi $0, 1$, rispettivamente i casi $h_\theta(x) < 0.5$ e $h_\theta(x) \geq 0.5$.

Una volta appresi i parametri, possiamo ignorare la funzione sigmoide e utilizzare il prodotto $\theta^T x$ per classificare, in quanto valgono le relazioni precedenti.

Chiaramente se non è possibile dividere linearmente, possiamo operare una trasformazione delle feature $\phi(x)$. Ad esempio, se la decision boundary dev'essere una circonferenza, basta considerare ad esempio:

$$-1 + x_1^2 + x_2^2 = 0 \iff x_1^2 + x_2^2 = 1$$

e il vettore di parametri θ è $[-1, 0, 0, 1, 1, 0]$. Il boundary è una circonferenza di raggio 1.

Funzione costo e valutazione

Consideriamo due classi $0, 1$. Plottiamo su y la funzione $\mathcal{J}(\theta)$ e sull'asse x la funzione $h_\theta(x)$. Consideriamo un campione e valutiamo il classificatore. Se il classificatore ha operato bene, ci aspettiamo un errore nullo, quindi $\mathcal{J}(\theta) = 0$. Se invece il classificatore ha classificato erroneamente, ci aspettiamo un errore molto alto.
Il grafico che ci aspettiamo allora è una curva simile a $-\log z$. La nostra loss function deve tenere in considerazione la funzione $h_\theta(x)$ e l'etichetta y. Per cui:

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

Pertanto su ogni campione possiamo stabilire se stiamo sbagliando o meno in base alla sua etichetta. Possiamo mettere tutto insieme in una formula:

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

Poiché dobbiamo computare ciò con tutti i campioni, la nostra funzione costo diventa:

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

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

Questa funzione di loss è **derivabile** e prende il nome di **binary cross-entropy loss**, o più precisamente **Bernoulli cross-entropy loss** in quanto deriva dal modello di probabilità di Bernoulli.
Come per Adaline, utilizziamo l'aggiornamento:

$$\theta_j \leftarrow \theta_j - \alpha \frac{d\mathcal{J}(\theta)}{d\theta_j} \quad \forall \theta_j$$

La derivata della funzione di loss è:

$$\frac{d\mathcal{J}(\theta)}{d\theta_j} = \frac{1}{m}\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

La funzione prende il nome di "cross entropy loss" in quanto può essere considerata come una densità di probabilità, più precisamente la probabilità di appartenere ad una classe o l'altra. L'entropia infatti:

$$H(x) = -\sum_{k=1}^{K} p_k(x) \log(p_k)$$

dove $p_k(x)$ è la probabilità di x di essere di classe k. Nella loss function $y \log(h_\theta(x))$ è il prodotto tra il valore numerico corrispondente alla classe (y) e il grado di sicurezza che il dato in input x sia di tale classe ($h_\theta(x)$).
Anche in questo caso aggiungiamo una parte di regolarizzazione alla loss function:

$$\frac{\lambda}{2}\sum_{j=1}^{d} \theta_j^2$$

Il classificatore può anche essere modificato in modo tale da includere una **area grigia** per cui determinati valori non vengano considerati per la classificazione.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Classificatori a k classi

Il regressore logistico è il modello ideale quando abbiamo due classi da identificare. Analizziamo ora il problema di classificaazione supponendo di avere k classi.
Possiamo classificare k classi utilizzando k regressori logistici e analizzando i gradi di sicurezza di tali regressori. Ogni regressore infatti computa la funzione $h_\theta(x)$, funzione che restituisce un valore nell'intervallo $[0, 1]$ e che esprime il grado di sicurezza di appartenenza alla classe 0 o 1 in virtù del valore soglia scelto. Allora l'idea è quella di utilizzare un classificatore per classe, e vedere quale classificatore fornisce un valore di sicurezza più alto. Questa metodologia prende il nome di **one versus all**.

Poiché ogni classificatore restituisce un valore compreso tra zero ed uno, la somma:

$$\sum_{i=1}^{k} h_\theta(i) \neq 1$$

indica che **non si ha una distribuzione di probabilità**. Infatti si potrebbero avere 3 classi e ottenere come stime di probabilità per le tre classi rispettivamente $[0.7, 0.4, 0.3]$. La classificazione viene allora stabilita in base all'argomento più alto:

$$k = \text{argmax}_k h_\theta(x)$$

Tuttavia, è anche possibile ottenere $[0.4, 0.2, 0.1]$, ossia trovarsi in **aree di massima incertezza**. Quando si considera un output in un'area di massima incertezza, consideriamo la classe tale che il decision boundary è più vicino al punto considerato, ossia sempre argmax.

Un altro modello è il **one versus one**. Ossia, si vanno a utilizzare $\frac{k(k-1)}{2}$ classificatori che stabiliscono il grado di confidenza del confronto c_1 versus c_2: valori sopra 0.5 indicano che il classificatore considera il campione appartenente alla classe c_2, mentre valori inferiori il contrario. A questo punto si va a stabilire qual è la classe che ha più classificazioni. Ad esempio, con 3 classi si hanno le comparison $(c_1, c_2), (c_2, c_3), (c_1, c_3)$; se otteniamo come output le classificazioni c_2, c_2, c_1, allora la classe è c_2.
Il vantaggio nell'utilizzo dell'approccio one-versus-one risiede nel fatto che **non si hanno aree di incertezza**. Se ad esempio abbiamo una classificazione a tre classi, con l'approccio one-vs-all utilizziamo quattro classificatori che risolvono il problema di classificazione individuando le classi k e $\neg k$, dove $k \in \{0, 1, 2\}$. Utilizzando questo approccio, un pattern potrebbe essere classificato come "non di classe 1", "non di classe 2" e "non di classe 3". Se utilizziamo l'approccio one-vs-one, i campioni vengono classificati tramite confronto tra singole classi: la comparison "1 vs 2" potrebbe classificare

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

"1", "2 vs 3" classifica "3" e "1 vs 3" classifica "1". Poiché il confronto avviene a coppie, le aree di incertezza di one-vs-all non si presentano. Poiché, però, è sempre necessario avere $\frac{k(k-1)}{2}$ classificatori nel caso one-vs-one, solitamente si utilizza one-vs-all.

Non è detto, inoltre, che due o più classi non possano presentare lo stesso numero di voti; in tal caso si verifica un **conflitto di decisioni**. In questi casi:

- il sistema può scegliere la classe con indice più basso. Si tratta di una soluzione semplice ma non desiderabile;
- si utilizza la somma delle probabilità dei classificatori binari coinvolti;
- si rifiuta di classificare l'input. Questo è tipico di contesti dove l'errore è costoso, come nel campo della medicina;
- si delega la scelta ad un esperto umano.

Discesa del gradiente stocastica

Il problema dell'algoritmo di discesa del gradiente risiede nel fatto che la valutazione della funzione $\mathcal{J}(\theta)$ potrebbe richiedere molto tempo. Se sono m i campioni e d le feature, abbiamo bisogno di almeno $m \times d$ calcoli. Se consideriamo anche n epoche, allora il numero di calcoli da svolgere diventa insostenibile.

Possiamo risolvere il problema utilizzando la **discesa del gradiente stocastica**. Si scelgono delle **batch** che hanno un numero di campioni pari a $b \leq m$. In ogni epoca troviamo b campioni e abbiamo un numero di epoche pari ad e. Il numero di operazioni è ridotto a $e \cdot b \cdot d$.
Se abbiamo più memoria e aumentiamo la dimensione del batch è necessario aumentare anche la learning rate. Questo aumento può essere lineare (se il batch è aumentato di un fattore k, allora anche il learning rate aumenta di tale fattore) oppure alternativo (ossia $\sqrt{k}$).

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

04 - Ottimizzazione, valutazione, algorithm design, Softmax

Momento e gradiente

Una variante della discesa del gradiente è quella del momentum. Negli aggiornamenti dei nostri parametri possiamo introdurre un parametro β in modo tale che, invece di modificare solamente con la derivata della funzione di loss, si utilizza il parametro z tale che:

$$\theta_j \leftarrow \theta_j - \alpha z_k$$

$$z_{k+1} = \beta z_k + \frac{d\mathcal{J}(\theta)}{d\theta_j}$$

Questo meccanismo prende il nome di **momentum**, e semplicemente aggiunge un' "inerzia" che permette di accelerare la convergenza al minimo. Più precisamente, il valore $\beta = 0.90$ è il migliore e tale risultato deriva da osservazioni statistiche.

Learning rate decay

Il problema del learning rate risiede nel fatto che se esso è troppo alto, si può divergere durante la discesa del gradiente. Se ad ogni epoca facessimo un plot del valore della funzione di loss, si potrebbero osservare dei picchi. L'idea allora è di partire con un valore α e man mano diminuirlo per evitare problemi di "salto" del minimo. Lo scaling del learning rate può essere eseguito in diversi modi:

- **costante**, ossia il learning rate è fisso;
- **time-based**, ossia si ha che $\alpha_t = \alpha_0 \cdot \frac{1}{1+kt}$, dove t è l'epoca e k è un parametro;
- **exponential decay**, ossia il learning rate diminuisce in maniera esponenziale, quindi $\alpha_t = \alpha_0 - e^{kt}$, sempre con k parametro;

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

- **adaptive methods**. Vi sono diversi metodi:
  - **adagrad**. Prevede grandi update per parametri sparsi e piccoli update per parametri non sparsi. Un parametro si dice sparso se i valori in input del parametro sono solitamente vicini a zero.
  - **adam**. Utilizza un momentum e un learning rate differente per ognuno dei pesi. Si tratta di un metodo più aggressivo.

Valutazione

Possiamo valutare un classificatore utilizzando la funzione $\mathcal{J}(\theta)$ senza regolarizzazione.
Si può anche vedere quali sono i campioni mal classificati, contarli e calcolare una media sul dataset. Questa è la **misclassification error** ed è la frazione di pattern erroneamente classificati. Questo tuttavia non ci spiega bene come sta avvenendo la classificazione: l'accuratezza infatti è globale, non per singole classi, pertanto non sappiamo se l'errore è distribuito su una determinata classe per la classificazione o meno.

Possiamo anche utilizzare una **confusion matrix**. Utilizziamo una tabella dove le colonne sono le etichette predette dal nostro modello, e le righe sono le etichette reali. Si ha più precisamente la tabella con due righe e due colonne, ognuna con etichetta "positivo" o "negativo". Abbiamo allora quattro valori: positivo-positivo, positivo-negativo, negativo-positivo, negativo-negativo.

1. positivo-positivo nella tabella indica il numero di campioni classificati come positivi e che sono veri positivi;
2. negativo-negativo indica il nuimero di campioni classificati come veri negativi;
3. positivo-negativo indica i falsi negativi. Sono campioni positivi erroneamente classificati come negativi;
4. negativo-positivo indica i falsi positivi. Sono campioni negativi classificati erroneamente come positivi.

Vogliamo allora che la diagonale principale di questa tabella sia **massima**, ossia senza errori. L'accuratezza del modello è calcolata con la frazione il cui numeratore è la somma di campioni classificati correttamente (positivo-positivo e negativo-negativo) e il denominatore è il numero di tutte le classificazioni.
Il ragionamento vale chiaramente per due classi. Estendiamo questo a k classi. Abbiamo una tabella con colonne di etichetta $0, 1, ..., k-1$. L'oggetto di posto (i, j) è il numero di campioni di classe i classificati come classe j. Notiamo che questa tabella ci permette anche di capire dove sta l'errore con più precisione, ad esempio capire che campioni di classe i sono maggiormente classificati

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

come di classe j. Possiamo sempre calcolare l'accuratezza del modello: basta fare la frazione tra la somma degli elementi sulla diagonale principale e la somma di tutti gli elementi della tabella di confusione.

Un parametro che viene anche utilizzato è la **precision**. La precision mette in relazione i positivo-positivo e i negativo-positivo. Più precisamente, si calcola il rapporto:

$$\frac{\text{true positives}}{\text{true positives + false positives}}$$

Ossia di tutti i positivi individuati dal modello, quale frazione sono realmente positivi.

La **recall** invece è il rapporto:

$$\frac{\text{true positives}}{\text{true positives + false negatives}}$$

Ossia di tutti i positivi reali, quale frazione sono correttamente etichettati come positivi.

Abbiamo visto precedentemente che come threshold abbiamo utilizzato 0.5. Aumentando la threshold, si può osservare un aumento di precisione, fino a quando con threshold 0.99 si osserva precisione 1, mentre con un threshold di 0.01 si osserva un recall di 1. Il valore di threshold viene stabilito in base al nostro obiettivo:

- se vogliamo essere molto sicuri, vogliamo alta precisione e basso recall;
- se vogliamo invece essere conservativi, vogliamo alto recall e bassa precisione.

In ultimo, un'altra misura che viene utilizzata è quella che prende il nome di F_1 **score**. Mette insieme precision e recall. F_1 score è il punteggio:

$$F_1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Questo funziona molto bene come metrica di precisione.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Algorithm design

Supponiamo di implementare una regressione lineare con regolarizzazione, e osservando il modello notiamo che gli errori sono comunque grandi. Cerchiamo di capire allora quale strategia applicare. Esistono diversi approcci:

1. aumentare i campioni di training;
2. ridurre il set di features, al fine di eliminare ridondanze e ridurre il rischio di overfitting;
3. cercare nuove feature per risolvere il problema, in quanto la rappresentazione attuale di questo non è idonea;
4. provare ad utilizzare feature polinomiali, x_1^2, x_2^2...;
5. provare a modificare il parametro di regolarizzazione λ.

Solitamente si cerca di effettuare test diagnostici per cercare di ottenere informazioni dettagliate su cosa funziona o meno, in modo tale da ottenere indicazioni per migliorare le prestazioni.

Valutazione incrociata

Possiamo utilizzare la $k-$ fold cross-validation per una valutazione incrociata. Possiamo utilizzare la cross-validation per selezionare gli iperparametri del modello, ad esempio il parametro di regolarizzazione λ. Dividiamo il training set in k fold, usando un fold per validation e gli altri per testing. L'idea è quella di ottenere diverse stime delle prestazioni del modello in base agli iperparametri scelti tramite i fold. Ossia, se dividiamo il nostro set in 5 fold, allora possiamo considerare 5 split, dove abbiamo 4 fold utilizzati per training e 1 fold utilizzato per la validation. Allora per un dato iperparametro $h \in H$, otteniamo 5 valori di $\mathcal{J}_{val}$, dove $\mathcal{J}_{val}$ è il valore di loss ottenuto nel validation set: li mediamo per ottenere $\overline{\mathcal{J}_{val}}$. Scegliendo un altro candidato in H, ripetiamo il processo fino ad ottenere un valore h^* che minimizza la media $\overline{\mathcal{J}_{val}}$.

Model selection

Supponiamo per semplicità che il nostro modello abbia un solo parametro. Possiamo scegliere il modello migliore modificando i gradi del nostro modello, ossia invece di utilizzare un modello lineare utilizziamo un modello quadratico, oppure un modello cubico e così via. Valutiamo il modello migliore in base all'errore che si ottiene sul modello tramite $\mathcal{J}(\theta)$. Scegliamo allora il modello con i parametri che minimizzano $\mathcal{J}(\theta)$ e testiamo sul testset il modello con tali parametri.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Bias e variance

Stabilito il grado del polinomio, possiamo valutare l'andamento della funzione $\mathcal{J}(\theta)$ sul training set e sul validation set. Osserviamo che:

- se gli errori sono alti in entrambi i casi, abbiamo **underfitting**. C'è alto bias dovvuto ad un'errata assunzione del modello (**high bias**);
- se gli errori sono bassi nel training ed alti in validation, abbiamo **overfitting** e **high variance**;
- se gli errori sono all'incirca uguali ed abbastanza "bassi", il modello è adatto.

Parametro di regolarizzazione

Per il parametro di regolarizzazione possiamo stabilire sperimentalmente quello più opportuno, considerando λ a partire da 0 e aumentando progresisvamente. Si seleziona il set di parametri tale che il valore di $\mathcal{J}_{Val}(\theta)$ è minimo.

Possiamo anche in questo caso analizzare il valore più opportuno λ in virtù dell'andamento delle curve $\mathcal{J}_{Val}$ e $\mathcal{J}_{Train}$. Al crescere di λ l'errore $\mathcal{J}_{Train}$ sale, in quanto stiamo cercando in questo modo di rendere il modello il più semplice possibile. Questo perché al crescere di λ pesa maggiormente la parte di regolarizzazione dell'espressione di $\mathcal{J}$. Se invece è molto piccolo λ, l'errore può essere basso sul training set, ma più alto sul validation set. Siamo in qeusto caso in overfitting, e quindi si ha high variance. Cerchiamo allora un λ intermedio.

Altre curve utili

Possiamo analizzare l'errore rispetto alla dimensione del dataset di training. Aumentare la dimensione del dataset non implica semplicemente che l'errore diminuisce, ma ci sono anche alti costi per l'acquisizione e l'etichettatura dei nuovi dati.
Ci aspettiamo che al crescere del numero di valori la curva di $\mathcal{J}_{Train}$ cresca. Chiaramente se il numero di dati $m = 1$, qualsiasi curva fitta e l'errore facilmente è minimizzato. All'aumentare dei dati è inevitabile che l'errore di training aumenti, perché è sempre più difficile trovare una curva che fitti tutti i dati.

Ci aspettiamo invece che $\mathcal{J}_{Val}$ diminuisca. Tuttavia se siamo in presenza di alta bias, aumentare i dati non aiuta molto. Se invece siamo in presenza di alta varianza, aumentare i dati potrebbe aiutare. Pertanto aumentare il numero di dati di training è utile a risolvere problemi di high variance.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Softmax

Estendiamo il classificatore visto fino ad ora. Abbiamo visto che n classificatori possono essere utilizzati per identificare n classi, separando gli spazi.

Cambiamo ora il nostro modello. Introduciamo il modello **Softmax**. Si tratta di un'estensione naturale del classificatore logistico per il caso multi-classe.

Modello

Il modello della Softmax permette di stimare la probabilità che un determinato dato sia di classe k. Idealmente vorremmo che, dato un input x di n feature, il modello restituisse un vettore di k elementi, tale che la somma di questi elementi sia pari ad 1 (ossia tale che si abbia una **distribuzione di probabilità**). Possiamo considerare la funzione:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}}$$

Il vettore z è un vettore di k elementi ottenuto trasformando le n feature in input del vettore x tramite:

$$z = A^T x + b$$

Più precisamente A^T è la matrice dei pesi di dimensione $k \times n$ e x è la colonna di feature. Ogni riga di questa matrice corrisponde ai pesi di una classe. In alternativa si scrive:

La funzione $\sigma(z)_j$ esprime pertanto la probabilità:

$$P(y = j|x)$$

Notiamo in questo caso che abbiamo una **distribuzione di probabilità**. Si ha infatti che:

- ogni elemento è sempre maggiore o uguale a zero, in quanto l'esponenziale è una funzione non negativa;
- la somma di tutti gli elementi del vettore z è uguale a 1. Infatti computando la somma $\sum_{j=1}^{k} \sigma(z)_j$ si semplificano numeratore e denominatore, ottenendo 1.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Il nostro modello parametrizzato pertanto corrisponde alla funzione:

$$h_\theta(x) = \frac{1}{\sum_{j=1}^{k} e^{\theta^{(j)\top} \cdot x}} \cdot \begin{bmatrix} e^{\theta^{(1)\top} x} \\ e^{\theta^{(2)\top} x} \\ \vdots \\ e^{\theta^{(k)\top} x} \end{bmatrix}$$

E restituisce una matrice appartenente a $\mathbb{R}^{k,1}$, dove ogni valore corrisponde alla probabilità $P(y = j|x)$ per $j \in \{0, ..., k-1\}$.

Cost function

Modifichiamo leggermente la cross-entropy loss function per il nostro modello:

$$\mathcal{J}(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{c=0}^{k-1} 1 \cdot \{y^{(i)} = c\} \log(h_\theta^{(c)}(x^{(i)}))] + \frac{\lambda}{2n}\sum_{c=0}^{k-1}\sum_{j=1}^{n} \theta_j^{(c)2}$$

dove la notazione:

$$1 \cdot \{\text{proposition}\}$$

indica che si ha 1 quando proposition è vera. In questo modo si calcola la loss solamente per la classe considerata e si misura il grado di sicurezza relativo alla predizione per aggiornarne i pesi. La funzione:

$$h_\theta^{(c)}(x) = P(y = c|x) = \frac{e^{\theta^{(c)T} x}}{\sum_{i=1}^{k} e^{(\theta^{(i)T} x)}}$$

Coincide con la funzione $\sigma(x)_c$. Con $\theta^{(c)}$ intendiamo il vettore di pesi associato alla classe c. Infatti, come visto precedentemente, il modello parametrico softmax considera una matrice di pesi, dove ogni riga è associata ad una classe.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Learning

Per trovare i parametri θ, utilizziamo la discesa del gradiente, tenendo conto tuttavia delle classi. Utilizziamo sempre la regolarizzazione per evitare overfitting. Più precisamente, la matrice θ per i nostri parametri corrisponde alla matrice A di dimensione $k \times n$ per trasformare il vettore x di n features in un vettore di dimensione $1 \times k$ tramite la trasformazione:

$$z = A \cdot x^T + b$$

Si ha che:

$$\theta_j^{(c)} \leftarrow \theta_j^{(c)} - \alpha \frac{\delta \mathcal{J}(\theta)}{\delta \theta_j^{(c)}}$$

per $j = 0, ..., n, c = 0, ..., k-1$. La derivata della funzione di loss diventa:

$$\frac{\delta \mathcal{J}(\theta)}{\delta \theta_j^{(c)}} = -\frac{1}{m} \sum_{i=1}^{m} [(1 \cdot \{y^{(i)} = c\} - h_\theta^{(c)}(x^{(i)}))x_j^{(i)}] + \frac{\lambda}{n}\theta_j^{(c)}$$

Proprietà di Softmax

Softmax è **superparametrizzato**, ossia per qualsiasi funzione h_θ che fitta i dati, ci sono molti parametri che ci permettono di ottenere la stessa soluzione per mappare i dati alle predizioni $0, ..., k-1$. Questo si dimostra sottraendo un arbitrario parametro ψ agli esponenziali $e^{\theta^{(c)T}x}$ e osservando che si ottiene nuovamente la funzione di partenza:

$$P(y = c|x) = \frac{e^{(\theta^{(c)} - \psi)^\top x}}{\sum_{j=1}^{K} e^{(\theta^{(j)} - \psi)^\top x}} =$$

$$= \frac{e^{\theta^{(c)\top} x} \cdot e^{-\psi^\top x}}{\sum_j = 1^K e^{(\theta^{(j)\top} x)} e^{-\psi^\top x}}$$

La proprietà di superparametrizzazione, pertanto, indica che **i modelli del parametro sono ridondanti**. Inoltre, è possibile minimizzare la funzione di loss $\mathcal{J}(\theta)$ sia considerando la matrice di pesi di righe $(\theta_1, \theta_2, ..., \theta_k)$ che la matrice $(\theta_1, ..., \theta_k)$. Osserviamo, inoltre, che impostando $\psi = \theta_k$ è anche

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

possibile annullare l'intero vettore di pesi θ_k senza influenzare l'ipotesi. Questo permette di ottimizzare il modello in maniera più rapida lavorando con un vettore in meno.

Se $k = 2$ si può dimostrare che la regressione logistica è un caso particolare di Softmax. Ricordiamo che il modello parametrizzato della regressione logistica è l'ipotesi:

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

Se $k = 2$ infatti si osserva:

$$h_\theta(x) = \frac{1}{e^{\theta^{(1)\top} x} + e^{\theta^{(2)\top} x}} \cdot \begin{bmatrix} e^{\theta^{(1)\top} x} \\ e^{\theta^{(2)\top} x} \end{bmatrix}$$

Dato $\psi = \theta^{(2)}$, possiamo utilizzare la proprietà di superparametrizzazione di Softmax e considerare la matrice dove ogni elemento è del tipo $\theta^{(i)} - \psi$. Si ha allora che:

$$h_\theta(x) = \frac{1}{e^{(\theta^{(1)} - \theta^{(2)})\top x} + e^{\vec{0} x}} \cdot \begin{bmatrix} e^{(\theta^{(1)} - \theta^{(2)})\top x} \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \dfrac{e^{(\theta^{(1)} - \theta^{(2)})\top x}}{1 + e^{(\theta^{(1)} - \theta^{(2)})\top x}} \\ \dfrac{1}{1 + e^{(\theta^{(1)} - \theta^{(2)})\top x}} \end{bmatrix} = \begin{bmatrix} 1 - \dfrac{1}{1 + e^{(\theta^{(1)} - \theta^{(2)})\top x}} \\ \dfrac{1}{1 + e^{(\theta^{(1)} - \theta^{(2)})\top x}} \end{bmatrix}$$

Indicando $\theta^{(2)} - \theta^{(1)}$ con θ', si ottiene il calcolo di probabilità della classe con $\dfrac{1}{1 + e^{-\theta'\top x}}$, che è identico alla regressione logistica.

Rappresentazione di Softmax come rete, divisione di spazi

Dati gli n input, possiamo vedere Softmax come una rete di percettroni tali che abbiamo n neuroni di input e k neuroni in output, dove k sono le classi. Un layer interno è costituito da k neuroni che computano le funzioni:

$$z_i = \sum_{j=1}^{n} w_{i,j} x_j + b_i$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Tali output vengono poi inviati a k neuroni di output che computano la funzione $\sigma(z)_i$ analizzata precedentemente.

Softmax divide gli spazi utilizzando boundaries lineari. Cambiando le feature e trasformandole in feature polinomiali, ci saranno curve a partizionare lo spazio.

Risulta possibile infine cambiare la funzione di loss:

$$L = \frac{1}{n} \sum_i \sum_{j \neq y_i} \max(0, f_j - f_{y_i} + 1) + \lambda \sum_k \sum_l \theta_{k,l}^2$$

Questa funzione prende il nome di **hinge loss**. In questo caso abbiamo un modello che usa una matrice $\theta \in R^{k \times n}$. Tale matrice viene moltiplicata per il vettore in input $x_i^T \in R^{n \times 1}$. Il risultato viene sommato ad un offset di bias $b \in R^{k \times 1}$. La funzione L penalizza le predizioni in cui il punteggio della classe corretta non è almeno più grande di 1 unità rispetto alla classe scorretta con il punteggio più alto. La formula garantisce un margine di almeno 1 unità tra la classe corretta e le altre classi. Questa loss function è utilizzata per i framework SVM, che ottimizzano su tutte le classi simultaneamente.

Problema

Abbiamo adesso un ottimo classificatore, ma i problemi possono essere molto complessi. Possiamo:

- avere trasformazioni delle feature per generale partizioni dello spazio;
- estendere il modello ad uno più complesso per trasformare feature e risolvere il problema di classificazione.

Questo può essere risolto utilizzando delle **reti neurali**.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

05 - Reti neurali, decision trees

Reti neurali

Abbiamo visto come estendere l'architettura del regressore logistico in softmax, in cui una singola macchina in output dà una distribuzione di probabilità sulle classi. Un problema visto è quello di avere una buona rappresentazione. Se il nostro problema non è lineare, allora i classificatori hanno difficoltà a risolvere il problema. Una strategia che abbiamo visto è la trasformazione dello spazio $\phi(x)$ estendendo le feature in maniera polinomiale. Successivamente, dopo aver trasformato lo spazio possiamo utilizzare la strategia del modelllo softmax.

Quello che possiamo costruire allora è un **multi-layer perceptron** (**MLP**) per risolvere il problema. Una prima sezione si occupa di trasformare l'input, e una seconda parte va a risolvere il problema di classificazione tramite il modello softmax. Possiamo anche affrontare il problema della regressione lineare usando una MLP ma sostituendo la parte di softmax con una parte che restituisce semplicemente la combinazione lineare.

Definizione del modello

Da un lato abbiamo l'input $a_0, ...a_n$. Da ogni nodo di input ci sono collegamenti verso altri neuroni. Ogni input viene moltiplicato per un peso θ_i. Il primo passo che facciamo pertanto è combinare linearmente l'input con il vettore di pesi e dare in pasto ai neuroni tale combinazione lineare. Ogni neurone che riceve tale combinazione computa una funzione detta **funzione di attivazione** fino al layer di output, che restituisce il risultato finale della computazione.

Si costruisce ad ogni passo una riga partendo dalla riga di input a del livello $l-1$ moltiplicandola per la matrice di pesi θ, dove ogni colonna θ_j corrisponde ai pesi del $j-$esimo neurone. Ossia, se n_{l-1} è il numero di neuroni del layer precedente, associamo un numero n_{l-1} di righe di pesi. La lunghezza di ciascuna riga è n_l: in questo modo associamo ad ogni neurone del layer l una colonna di pesi da moltiplicare all'input ricevuto. La riga $z^{(l)}$ che si ottiene:

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

$$z^{(l)} = a^{(l-1)} \cdot \theta^l$$

ha dimensione n_l. Ad esempio, se abbiamo 2 nodi nel layer $l-1$ e 3 nodi nel layer l, si ha che $a^{(l-1)}$ è di dimensione 1×2, mentre θ^l è di dimensione 2×3. Il prodotto tra le due matrici è la matrice risultante $z^{(l)}$ di dimensione 1×3. Il $j-$esimo valore del vettore è la combinazione lineare:

$$z_j^{(l)} = a^{(l-1)} \cdot \theta_j^l = \sum_{i=1}^{n} a_i^{(l-1)} \cdot \theta_{i,j}^l$$

Il valore a^l è denotato come **valore di attivazione** e corrisponde all'output del layer l:

$$a^l = f(z^{(l)})$$

Una rete neurale pertanto è una **combinazione di funzioni** f che prendono il nome di **funzioni di attivazione**. Ci interessa che la funzione di attivazione sia derivabile. Il trucco in questo caso per poter allenare la rete è quello di **computare la derivata interna e moltiplicarla per la derivata della funzione esterna**, sfruttando la proprietà della derivata composta:

$$f(g(x))' = f'(g(x)) \cdot g'(x)$$

A seconda del numero di output che diamo alla rete, nonché delle funzioni di attivazioni utilizzate, è possibile avere un classificatore multi-classe o un semplice regressore logistico, o addirittura un regressore lineare se la rete è veramente semplice.

Cost function

Possiamo sempre utilizzare la cross-entropy loss per la classificazione, e nel caso della regressione una funzione di loss come la MSE. Il problema principale sarà il calcolo efficiente della derivata per la discesa del gradiente.

La tecnica utilizzata per addestrare le reti neurali prende il nome di **retropropagazione dell'errore**, meglio nota in inglese con il termine **backpropagation**. Il valore della loss function è computato in virtù del valore che restituisce la rete neurale, e la rete neurale è una combinazione di funzioni.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

FACOLTATIVO: Backpropagation (non è in programma!)

In generale abbiamo visto:

$$z^l = \theta^l \cdot a^{l-1} \qquad a^l = f^l(z^l)$$

La backpropagation calcola il gradiente della loss $\mathcal{J}$ rispetto ai parametri utilizzando la regola della catena.
Definiamo l'errore allo strato l come la quantità:

$$\delta^l = \frac{\delta \mathcal{J}}{\delta a^l}$$

All'ultimo strato tale errore è:

$$\delta^L = \frac{\delta \mathcal{J}}{\delta a^L} \circ (f^L)'(z^L)$$

dove $\circ$ indica il **prodotto puntuale**, ossia elemento per elemento. Per gli strati precedenti:

$$\delta^l = ((\theta^{l+1})^\top \delta^{l+1}) \circ (f^l)'(z^l)$$

La backpropagation consiste allora nel calcolare l'errore δ^l ad ogni strato l e computare il **gradiente dei pesi**:

$$\nabla_{\theta^l} \mathcal{J} = \delta^l (a^{l-1})^\top$$

E ogni matrice di pesi è aggiornata con la discesa del gradiente:

$$\theta^l \leftarrow \theta^l - \alpha \nabla_{\theta^l} \mathcal{J}$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Overfitting e underfitting nelle reti neurali

Se le reti neurali sono semplici, ossia con pochi parametri e pochi layer, abbiamo problemi di underfitting. Se invece il numero di neuroni è molto alto, possiamo andare incontro a problemi di overfitting. Il problema che ora ci poniamo è di tipo **architetturale**, in quanto riguarda direttamente la struttura della rete. I parametri di regolarizzazione nella funzione di costo ci aiutano a capire quale architettura è più adatta.

A partire dalla base della MLP è possibile formare reti neurali anche di diverso tipo, come le **reti neurali ricorrenti**, in cui i neuroni possono avere dei cappi per prendere informazioni da computazioni precedenti.

Decision trees

I **decision trees** sono dei modelli non-parametrici predittivi utilizzati per trarre conclusioni da un set di osservazioni. I decision trees si dividono in:

- **classification trees** se la feature target ha un set discreto di valori. In questo caso si risolve quindi un task di classificazione: ad una tupla in input viene associata un'etichetta;
- **regression trees** se il valore target è un valore continuo, per cui si considera un valore medio in un intervallo o un valore di previsione.

Il vantaggio principale dei decision trees rispetto alle reti neurali è la **spiegabilità**. Allenato il decision tree, possiamo considerare un path per analizzare come un determinato input è stato classificato. Nelle reti neurali ciò non avviene, in quanto si computano funzioni internamente. Inoltre, i decision trees sono molto veloci in quanto possono essere computati dalla CPU direttamente, senza la necessità di utilizzare necessariamente una GPU per il calcolo matriciale.

Nodi interni

I nodi interni decisionali possono essere monovariati o multivariati.
Nel primo caso si utilizza un singolo attributo x_i. In questo caso l'attributo può essere numerico o discreto. Nel caso numerico si opera un binary split: si sceglie il nodo in base alla condizione $x_i > w_m$. Per valori discreti invece si può operare un $n-$way split per gli n possibili valori che può assumere la variabile.
Nel caso multivariato, si utilizzano tutti gli attributi x. In questo caso si deve stabilire il set di parametri per cui la combinazione lineare è maggiore di una certa soglia.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Foglie

Le foglie corrispondono a classi per problemi di classificazione o valori numerici (media dei campioni presenti, interpolazione) per task di regressione.

Dato un nodo m, possiamo avere N_m istanze che raggiungono tale nodo e N_m^i che appartengono ad una classe C. Indichiamo allora la probabilità che una di queste istanze sia di classe C con la seguente notazione:

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

Il nodo si dice **puro** se p_m^i è pari a zero o uno. La purezza di un nodo è calcolabile mediante l'entropia:

$$I_m = -\sum_{i=1}^{K} p_m^i \cdot \log_2(p_m^i)$$

In questo caso K indica il numero di classi nel problema di classificazione. Quello che ci interessa è, ad ogni passo, eseguire i migliori split. Se abbiamo un nodo puro, allora possiamo direttamente generare una foglia nel nostro albero decisionale e terminare il processo. Altrimenti si continua ricorsivamente.

Dopo lo split, l'impurità tiene conto del fatto che di tutte le N_m tuple che sono arrivate al nodo, solo N_{mj} passeranno al ramo j. Tra queste N_{mj} tuple, solo N_{mj}^i saranno appartenenti alla classe C_i. Come prima, possiamo indicare allora la probabilità di appartenere alla classe C_i nel seguente modo:

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

Avremo allora che l'impurità dopo lo split è data da:

$$I'_m = -\sum_{j=1}^{n} \frac{N_{mj}}{N_m} \sum_{i=1}^{K} p_{mj}^i \log_2 p_{mj}^i$$

L'impurità pertanto è misurata in virtù dei figli che abbiamo dopo lo split. Ogni figlio j ha il proprio grado di impurità pesato in virtù del numero di tuple che attraversano il branch e giungono in quel nodo. Quindi l'impurità del nodo è la somma pesata delle impurità dei figli.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Scelta dell'attributo migliore

Per la costruzione dell'albero cerchiamo ad ogni passo di scegliere il migliore attributo per effettuare lo split. Sia dato quindi un insieme di N tuple. Ogni tupla è dotata di un certo numero di feature. Per ogni feature, calcoliamo l'entropia I_m. Si sceglie l'attributo a per cui I_m è minimo:

- se a è un valore discreto, si effettuano n split dove n è il numero di possibli valori per a;
- se a è continuo, si ordinano i valori osservati di a e si considerano i **punti di mezzo** tra valori consecutivi come possibili soglie. Ossia, se si osservano i valori $2, 3, 5$, come soglie si considerano $\frac{2+3}{2}, \frac{3+5}{2}$.

A questo punto si considera la purezza dei nodi: se uno dei nodi figli è puro, allora diventa foglia. In alternativa si procede in maniera ricorsiva. L'obiettivo è, ad ogni passo, scegliere l'attributo che minimizza l'impurità dei figli. Dire che l'impurità dei figli dev'essere minima significa dire che **la differenza tra l'impurità del padre e l'impurità dei figli dev'essere massima**, o che semplicemente minimizza l'entropia del nodo dopo lo split.

Può darsi che determinati attributi non vengano utilizzati in quanto superflui nella decisione.

Procedura di costruzione

La procedura di costruzione è ricorsiva. Scelto una soglia θ_I per l'entropia di un nodo:

- se il nodo ha entropia inferiore alla soglia, diventa foglia;
- se il nodo ha entropia superiore alla soglia, si sceglie l'attributo che minimizza l'entropia dei figli;
- per ogni ramo creato, richiama la procedura sui nodi figli.

Il metodo `GeneraAlbero` prende in input l'insieme di esempi del nodo.

```
GeneraAlbero(X):
    se EntropiaNodo(X) < Theta_I:
        Crea foglia con etichetta pari alla classe di maggioranza in X
        Restituisci
    i = MiglioreSplit(X)
    per ogni ramo di x_i:
        X_i = Esempi del ramo
        GeneraAlbero(X_i)
```


Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

```
MiglioreSplit(X):
    MinEntropia = +Infinito
    per i = 1 fino al numero di feature:
        se x_i è discreto:
            dividi X in X_1,...,X_n in base a x_i
            entropia = EntropiaPostSplit(X_1,...,X_n)
            se entropia < MinEntropia:
                IndiceMiglioreFeature = i
                MinEntropia = entropia
        se x_i è continuo:
            per ogni threshold t per x_i:
                dividi X in X_1, X_2 in base a t
                entropia = EntropiaPostSplit(X_1, X_2)
                se entropia < MinEntropia:
                    IndiceMiglioreFeature = i
                    MinEntropia = entropia
    Restituisci IndiceMiglioreFeature
```

Regression trees

Come anticipato, un **regression tree** è un modello predittivo per stimare valori continui in virtù di un insieme di caratteristiche. I dati vengono suddivisi in intervalli continui.

A differenza dei classification trees, uno split avviene partizionando l'insieme di valori continui in intervalli. Se scelto un valore soglia, allora lo split è binario.
La qualità di uno split è determinata tramite la somma degli scarti quadratici. Dato un nodo m, se N_m è il numero di tuple che arrivano al nodo, si ha che l'errore è dato da:

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

dove y^t è il valore da predire per l'esempio t e y_m è la predizione del nodo m, cioè la **media del valore da predire calcolata con gli esempi che giungono nel nodo**. $b_m(x^{(t)})$ è la funzione che restituisce 1 se $x^{(t)}$ raggiunge il nodo m, altrimenti è pari a zero. Dopo lo split l'errore è calcolato come segue:

$$E'_m = \frac{1}{N_m} \sum_j \sum_t (y^{(t)} - \hat{y}_{mj})^2 \cdot b_{mj}(x^{(t)})$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Se l'errore è al di sotto di una opportuna soglia θ, allora si crea una foglia. Altrimenti, si procede con ulteriori split fino a quando non si ottiene un errore accettabile.

Pruning

Intrinsecamente il decision tree è soggetto ad overfitting per la minimizzazione dell'entropia. Infatti l'algoritmo base, noto come ID3, continuerà a dividere i dati fino a quando tutte le foglie sono completamente pure. Questo può creare un albero che è influenzato dalla varianza dei dati. Ciò implica che **piccole perturbazioni nei dati mutano profondamente la struttura dell'albero**. Naturalmente questo comporta anche **overfitting**.

Per evitare l'overfitting, si può operare del **pruning**, ossia la rimozione di sottoalberi per una migliore generalizzazione e una riduzione della varianza. Si hanno due approcci principali:

1. **pre-pruning**. Nel pre-pruning si interrompe la crescita dell'albero prima che raggiunga la massima profondità, ad esempio considerando la soglia di riduzione dell'impurità o addirittura un massimo valore di profondità.
2. **post-pruning**. Nel post-pruning si costruisce prima l'albero nella sua interezza, e poi si rimuovono i branch che riducono la precisione del modello. Tali sottoalberi vengono sostituiti con una foglia che rappresenta la classe più frequente delle tuple che giungono in tale branch.

Estrazione delle regole, induzione delle regole

Da un albero di decisione possiamo estrarre facilmente delle regole partendo dalla radice e considerando i path verso le foglie. Questo può essere utile in quanto **fare il pruning delle regole è molto più semplice che farlo sull'albero**. Una volta che le regole sono ridotte, è anche possibile ridurre la dimensione dell'albero, riordinando le regole per ottenere il miglior albero. Si tratta di un metodo frequentemente utilizzato per la costruzione di alberi decisionali e di regressione.

Si può anche costruire un albero imparando prima le regole dai dati. Questa procedura prende il nome di **rule induction**, espressa sempre nella forma:

IF condition THEN conclusion

dove condition indica un insieme di test su attributi e conclusion la classe. Le condizioni possono essere combinate con operatori logici per formare regole più complesse. Tali condizioni vengono aggiunte una per volta per ottimizzare una determinata euristica fino al raggiungimento di un criterio

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

di arresto. Appresa la regola, le tuple **coperte** da tale regola vengono rimosse dal dataset e si ripete il processo con le tuple rimanenti. Si dice che una regola **copre** una tupla se la tupla soddisfa le condizioni della regola. L'approccio che costruisce le regole in questo modo prende il nome di **sequential covering**.

Un algoritmo noto di sequential covering è **RIPPER** (**Repeated Incremental Pruning to Produce Error Reduction**), che costruisce l'insieme di regole e lo ottimizza in virtù della **description length**:

```
Ripper(Pos, Neg, k):
    SetRegole = ImparaSetRegole(Pos, Neg)
    Per k volte:
        OttimizzaSetRegole(SetRegole, Pos, Neg)

ImparaSetRegole(Pos, Neg):
    SetRegole = Vuoto
    DL = DescLen(SetRegole, Pos, Neg)
    Ripeti:
        Regola = ImparaRegola(Pos, Neg)
        SetRegole = SetRegole + Regola
        DL' = DescLen(SetRegole, Pos, Neg)
        Se DL' > DL + 64:
            PruneSetRegole(SetRegole, Pos, Neg)
            Restituisci SetRegole
        Altrimenti:
            DL = DL'
            Rimuovi esempi coperti in Pos e Neg
    Fino a Pos = Vuoto
    Restituisci SetRegole

PruneSetRegole(SetRegole, Pos, Neg):
    per ogni Regola in SetRegole a partire dalla fine:
        DL = DescLen(SetRegole, Pos, Neg)
        DL' = DescLen(SetRegole - Regola, Pos, Neg)
        Se DL' < DL:
            Rimuovi Regola da SetRegole
    Restituisci SetRegole

OttimizzaSetRegole(SetRegole, Pos, Neg):
```


Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

```
Per ogni Regola in SetRegole:
    DL0 = DescLen(SetRegola, Pos, Neg)
 DL1 = DescLen(SetRegola – Regola + RimpiazzaRegola(SetRegole, Pos, Neg), Pos,
 DL2 = DescLen(SetRegola – Regola + RivediRegola(SetRegole, Pos, Neg), Pos, Neg
    Se DL1 = min{DL0, DL1, DL2}:
    Rimuovi Regola e inserisci RimpiazzaRegola(SetRegole, Pos, Neg)
    Altrimenti Se DL2 = min{DL0, DL1, DL2}:
      Rimuovi Regola e inserisci RivediRegola(SetRegole, Pos, Neg)
restituisci SetRegole
```

In primis si impara il set di regole, rimuovendo di volta in volta gli esempi dall'insieme di esempi di classe positiva quando una regola migliora la description length. Dopo aver appreso le regole, il set viene ottimizzato in due possibili modi:

- rimpiazzando la regola con un'altra;
- revisionando la regola, rifinendo le condizioni.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

06 - Decision trees, random forest

Può succedere che determinati valori tra gli attributi manchino in quanto non osservati. Se ad esempio un attributo A presenta valori mancanti in alcuni examples, allora è possibile assegnare il valore più comunemente osservato per esempi che hanno la stessa classe. Ossia, se per gli esempi di classe 2 il valore più come di A è t, allora ad esempi di classe 2 per cui non si ha una misurazione di A verrà assegnato t.

I nostri dati potrebbero anche essere rumorosi:

- due esempi possono avere le stesse coppie attributo-valore, ma classificazioni differenti;
- alcuni attributi sono errati per errori nell'acquisizione dei dati;
- i dati sono stati etichettati erroneamente.

Questo causa **overfitting**. L'accuracy nel training set è molto più alta del testing set. Nel caso dell'overfitting, vogliamo fermare la discesa nell'albero, acquisire nuovi dati, rimuovere attributi irrilevanti o completare l'albero e poi fare un post-prune.

Per selezionare il miglior albero, possiamo utilizzare un validation set oltre al training set.

Chiaramente i decision trees non sono sempre il modello migliore, ma per una pool di dati mista si tratta del modello più facilmente interpretabile.

Cenni di ensemble learning

Abbiamo visto come il decision tree possa essere soggetto a overfitting in virtù del fatto che si adatta molto facilmente ai dati. In questo caso può essere utile applicare dei **meta-algoritmi** di **ensemble learning**.

Ensemble learning

I modelli di apprendimento supervisionati che abbiamo osservato fino ad ora lavorano tramite algoritmi che formulano un'ipotesi in un determinato spazio. Anche se in tale spazio possa esistere un'ipotesi che funge da miglior predizione per il problema analizzato, spesso il singolo modello può fare fatica a trovare tale soluzione.
L'**ensemble learning** consiste nell'allenare due o più modelli di machine learning sullo stesso problema. L'idea è quella di allenare un insieme di **weak learner** sullo stesso task, in modo tale che ogni weak learner abbia un **livello alto di bias**. L'idea di avere alta polarizzazione su ogni learner è quella di avere valori di output con **alta varianza**. I risultati dei weak learner sono combinati per produrre un unico risultato accurato.

Due tecniche molto comuni di ensemble learning sono il **bootstrap aggregating** (o **bagging**) e il **boosting**, che vedremo in futuro.

Prior knowledge

Se abbiamo dei dati in cui alcune classi sono più rappresentate rispetto alle altre, è necessario normalizzare rispetto all'inverso della frequenza dei dati. Nel considerare l'impurità di un nodo, essa deve considerare quanto pesa questa classe. Classi più numerose pesano di meno e non incidono maggiormente nella computazione dell'entropia. La conoscenza della frequenza dei dati appartenenti ad una data classe, e in generale una qualunque **prior knowledge** sul training set, può essere utilizzata per l'allenamento del modello.

Randomized trees

Un **random tree** è un decision tree costruito in maniera del tutto casuale. Si decide in maniera casuale sia la funzione sia la threshold per il nodo.

Come esempio per capire la procedura, supponiamo di lavorare con un modello lineare. La nostra $f(x)$ è:

$$f(x) = wx^T + b$$

La threshold è l'intercetta:

$$wx^T > b$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Assegnati randomicamente w, b si ottengono linee nello spazio di dati da dividere. Valutiamo la retta migliore utilizzando un'informazione quale l'entropia. Si sceglie allora la migliore retta generata con i valori casuali.
L'approccio casuale permette una costruzione greedy di modelli anche più complessi. Si può anche operare tramite bagging per ottenere un modello più performante e in maniera più rapida.

Il learning di un randomized tree segue un approccio randomizzato. Il set I_n di esempi del training che raggiunge il nodo n può essere diviso nei due insiemi I_l e I_r complementari:

$$I_l = \{i \in I_n | f(v_i) < t\} \qquad I_r = I_n \setminus I_l$$

$f(v_i)$ è la funzione che estrae una feature dal vettore v_i di feature dell'esempio i dall'insieme F di possibili feature. La threshold è un valore scelto nel range di valori di $f(v)$. In un extremely randomized tree, anche abbreviato come **ExtraTree**, la scelta di queste feature è totalmente casuale. In alternativa, si ottimizza l'information gain generando un certo numero di coppie f, t casuali e calcolando l'information gain. Si sceglie in maniera greedy la migliore coppia.

Random forest

L'idea della scelta casuale può essere applicata per ottenere **foreste** che aumentano la stabilità della classificazione. La classificazione si ottiene considerando la media delle distribuzioni computate sulla distribuzione delle foglie. Infatti ogni albero restituisce una differente distribuzione di probabilità, rappresentabile visivamente con un istogramma normalizzato. La foresta che si ottiene prende il nome di **random forest**.

L'idea è quella di utilizzare la tecnica di bagging per la costruzione dei diversi alberi, e utilizzare l'approccio di learning visto in precedenza. Per far sì che i vari alberi non siano correlati tra loro in virtù della **scelta della migliore feature**, si può anche ridurre il set di feature possibili da considerare quando si decide il miglior split. Il problema che si può presentare, infatti, è che la scelta del miglior split può causare alta correlazione, che si traduce quindi in più alberi polarizzati allo stesso modo. Si adotta solitamente la seguente strategia:

- per un task di classificazione, ogni albero ha accesso ad un numero di feature pari a $\sqrt{n}$;
- per un task di regressione, ogni albero ha accesso ad un numero di feature pari a $\frac{n}{3}$.

Risulta interessante come le random forest possano essere utilizzate per clustering. Possiamo costruire un modo su come guardare lo spazio e assegnare ad un pattern una stringa binaria in base alle

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

classificazioni dei nodi. Questo può essere utilizzato per il clustering: ad ogni classificazione possiamo associare una stringa binaria.

Non esiste un magic number per feature e threshold per gli alberi:

- poche feature causano underfitting;
- tante feature causano training più lento;
- troppe feature causano training molto lento e overfitting.

Per la profondità degli alberi, possiamo utilizzare un valore minimo di gain di entropia. Stabilire la soglia di entropia è difficile tuttavia, pertanto si definisce un limite fisso di profondità e si modula tale valore.

Random ferns per keypoint recognition

Abbiamo visto come gli alberi possano essere utilizzati come classificatori. Un task ricorrente in computer vision è la **keypoint recognition**, utilizzato per l'estrazione di informazioni di forme tridimensionali e per il tracciamento di oggetti nei frame video. Il problema viene risolto tramite un approccio standard che consiste nella costruzione di descriptors di parti di immagini note come **patch**. Il processo tuttavia può richiedere operazioni quali rotazioni e normalizzazioni di intensità, per cui si ha un alto costo computazionale. Un approccio più semplice consiste nel risolvere il task come un problema di classificazione, per cui si può costruire una random forest per ridurre i costi computazionali.

Un approccio ancora più semplice consiste nello **sfruttare la relazione del teorema di Bayes per risolvere il problema senza strutture gerarchiche**. Le strutture che utilizziamo per la keypoint recognition prendono il nome di **ferns**; ogni fern è un piccolo set di test binari che restituisce la probabilità che una patch appartenga ad una determinata classe. Combiniamo i risultati tramite il teorema di bayes.

Un esempio può essere il riconoscimento di segnali stradali. Un oracolo può stabilire se è presente un segnale stradale. La patch può essere pertanto classificata tramite il modello.

La classificazione consiste nel considerare la classe che soddisfa:

$$\text{argmax } P(C = c_i|\mathbf{patch})$$

$\mathbf{patch}$ può essere identificata dal set di test binari randomici $f_1, f_2, ..., f_N$:

$$P(C = c_i|\mathbf{patch}) = P(C = c_i|f_1, f_2, ..., f_n, f_{n+1}, ..., f_N)$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

La probabilità è proporzionale alla:

$$P(f_1, ..., f_N|C = c_i)$$

Infatti ciò deriva dal **teorema di Bayes**:

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

Quindi a meno del fattore di scala $P(B)$, possiamo dire che:

$$P(A|B) \simeq P(B|A) \cdot P(A)$$

Quindi diciamo che:

$$P(C = c_i|f_1, ..., f_N) \simeq P(f_1, ..., f_N|C = c_i) \cdot P(C = c_i)$$

Assumiamo che la probabilità $P(C = c_i)$ sia **uniforme**. Tale assunzione è legittima in quanto per ciascun keypoint si raccoglie lo stesso numero di esempi, per cui non vi è motivo nel considerare una probabilità più alta di un'altra: ossia, **il dataset si considera bilanciato a priori**. Il problema adesso consiste nello stimare $P(f_1, ..., f_N|C = c_i)$.

Consideriamo le feature f banali, quali il confronto di intensità tra due pixel:

$$f_i = \begin{cases} 1 & \text{se } I(m_{i,1}) \leq I(m_{i,2}) \\ 0 & \text{altrimenti} \end{cases}$$

Se abbiamo N test binari, associamo ad ogni immagine una sequenza binaria lunga N. Le sequenze binarie che possiamo avere, pertanto, sono 2^N. In teoria, per ogni classe, dovremmo avere un numero rappresentativo di esempi che ci permette di descrivere la probabilità di osservare i 2^N pattern binari nota la classe. Questo è **insostenibile**.

Ignorando totalmente la correlazione delle feature, ossia considerando **ogni feature indipendente**:

$$P(f_1, ..., f_N|C = c_i) = \prod_{j=1}^{N} P(f_j|C = c_i)$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Il problema è molto più approcciabile, visto che dobbiamo semplicemente stimare probabilità per $2 \cdot N$ pattern. Ma chiaramente è un'**estrema semplificazione**. Per cui si può tentare un approccio intermedio, costruendo dei ferns di dimensione $n < N$ e calcolando:

$$P(f_1, ..., f_n | C = c_i) \cdot P(f_{n+1}, ..., f_{2n} | C = c_i) \cdot ...$$

L'idea è quindi di prendere gruppi di feature e di ottenere un risultato proporzionale a quello atteso dal teorema di Bayes. I gruppi di feature inoltre sono **casuali**.

Margine, Support Vector Machine (SVM)

Riprendiamo la classificazione binaria per introdurre le **SVM** (Support Vector Machine). SVM è un modello che permette di risolvere un problema di ottimizzazione vincolato che restituisce una soluzione ottima per problemi di classificazione binari.

Per problemi di classificazioni binari, andremo da problemi più semplici, ossia problemi in cui gli spazi sono separabili linearmente. Un problema che abbiamo visto riguarda la scelta dell'iperpiano per separare lo spazio. Il problema che ci si pone quindi è quello di generare il miglior iperpiano per classificare lo spazio.

Introduciamo il **margine** come la distanza del pattern più vicino ad una linea (i.e. iperpiano. La linea è nel caso bidimensionale, più facilmente rappresentabile). Dalle feature x cerchiamo quindi la funzione:

$$f(x) = < w, x > + b$$

$< w, x >$ denota il prodotto scalare. w è il vettore ortogonale alla retta, nonché **vettore di pesi** del nostro modello, e b è l'intercetta. Ossia, w determina l'inclinazione della retta e b lo spostamento dall'origine. Vogliamo soddisfatti i constraints:

- $< w, x > + b \geq 0 \quad \forall x | y = +1$;
- $< w, x > + b < 0 \quad \forall x | y = -1$.

Vogliamo quindi trovare una linea in modo tale che i constraint siano verificati. I constraint possono essere messi assieme nell'espressione:

$$y \cdot (< w, x > + b) \geq 0$$

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

In quanto y assume i valori nell'insieme $\{-1, +1\}$.

Supponiamo x_A sia il punto più vicino alla retta di classificazione, e senza ledere generalità al discorso assumiamo che x_A sia dal lato positivo dell'iperpiano. Il valore x_A è ottenibile con la seguente somma:

$$x_A = x'_A + r\frac{w}{||w||}$$

dove r è la distanza tra il punto e la retta. Ci interessa allora determinare r.
Siamo interessati solamente alla direzione del vettore, e non alla sua lunghezza. Pertanto possiamo introdurre come constraint:

$$||w|| = 1$$

Poiché x_A è il più vicino alla retta, vogliamo **massimizzare** r. Infatti avendo un dataset linearmente separabile, abbiamo infiniti iperpiani candidati alla separazione, e quindi infiniti classificatori che possono risolvere il problema senza errori sul training. Ciò che noi vogliamo tuttavia è una soluzione che **massimizza il margine**, così da ridurre l'errore in fase di testing.

Quanto detto può essere, quindi, riassunto nella formulazione:

$$y \cdot (< w, x > +b) \geq r$$

Vogliamo allora trovare il massimo valore di r rispetto ai valori w, b, r soggetto al vincolo $y \cdot (< w, x > +b) \geq r$ e al vincolo $||w|| = 1$. Questo prende il nome di **problema del massimo margine** o **hard SVM**, ossia il problema della ricerca del miglior iperpiano. Stiamo quindi costringendo ad avere una distanza dal punto più vicino alla retta più alta possibile.

Per come è stato costruito il modello, ossia **per via geometrica**, abbiamo imposto il constraint $||w|| = 1$. Solitamente però, invece di scalare il vettore di parametri w, è lo spazio ad essere scalato. Ossia, trasformiamo lo spazio in modo tale che non si scala il vettore w, ma scaliamo le feature in maniera tale che la distanza del punto più vicino sia pari ad 1. I nostri constraint saranno:

$$y \cdot (< w, x > +b) \geq 1$$

Consideriamo pertanto il punto sul margine x_A. Si ha che:

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

$$< w, x_A > +b = 1$$

La proiezione del punto x'_A sulla retta è tale che:

$$x'_A = x_A - r\frac{w}{||w||}$$

Quindi:

$$< w, x_A - r\frac{w}{||w||} > +b = 0$$

Si ottiene allora:

$$< w, x_A > +b - r\frac{< w, w >}{||w||} = 0$$

Chiaramente notiamo che $< w, x_A > +b = 1$. Pertanto:

$$1 - r||w|| = 0 \implies r = \frac{1}{||w||}$$

Il risultato osservato sembra privo di senso: cerchiamo infatti di derivare la distanza r partendo dalla lunghezza del vettore w, che però non conosciamo. La chiave di lettura di tale risultato è la seguente: r è una "variabile temporanea" che utilizziamo esclusivamente per questa derivazione.

Il problema allora diventa:

$$\max_{w,b} \frac{1}{||w||}$$

sotto il vincolo:

$$y(< w, x > +b) \geq 1$$

Solitamente il problema della massimizzazione è risolto tramite la minimizzazione della norma al quadrato:

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

$$\min_{w,b} \frac{1}{2}||w||^2$$

Quest'equazione è meglio nota come **hard margin SVM** in quanto la formulazione non permette violazioni della condizione di margine.

Si può dimostrare che le due formulazioni, ossia la massimizzazione del margine geometrico r e la minimizzazione della norma del vettore w, sono identiche. Nella prima formulazione, ossia $||w|| = 1$, la massimizzazione di r può essere vista come la massimizzazione del quadrato di r, in quanto r è una quantità non negativa. Se $||w|| = 1$ allora possiamo considerare un nuovo vettore peso w' non normalizzato. Si ottiene:

$$\max_{b,r,w'} r^2$$

sotto la condizione:

$$y \cdot (< \frac{w'}{||w'||}, x > +b) \geq r$$

La massimizzazione di r^2 vincola r ad essere una quantità positiva, per cui possiamo dividere i termini $\frac{w'}{||w'||}$ e b per r e ottenere i termini w'' e b''. Si deriva:

$$w'' = \frac{w'}{||w'||r} \implies ||w''|| = \frac{1}{r}||\frac{w'}{||w'||}|| = \frac{1}{r}$$

Allora si ottiene che la massimizzazione di r^2 coincide con la massimizzazione:

$$\max_{w'',b''} \frac{1}{||w''||^2}$$

Ma la massimizzazione della frazione corrisponde con la minimizzazione di $||w''||^2$, che coincide con la minimizzazione di $\frac{1}{2}||w''||^2$. Questo quindi ci permette di dire che risolvere il problema di massimizzazione di margine coincide con risolvere il problema della minimizzazione della norma di w quando il margine è fissato a 1.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

07 - SVM

Soft Margin

Il problema che analizziamo solitamente può includere soluzioni che non sono linearmente separabili; tuttavia possiamo comunque trovare una linea tale che si minimizza l'errore di classificazione per ottenere una classificazione accettabile. Dobbiamo permettere quindi a qualche campione di potersi spostare nello spazio del margine, e questo spostamento è un **costo**. Vogliamo che questo "spostamento" sia il minimo possibile.

Passiamo allora alla versione **soft** della SVM. La formulazione è identica, ma per ogni campione è aggiunta una **slack variable**. Al campione $n-$esimo è associata una variabile ϵ_n che restituisce la distanza dal margine. La distanza è sempre positiva come constraint, e rilassiamo il problema in modo tale che:

$$y(< w, x_n > +b) \geq 1 - \epsilon_n \quad \epsilon_n \geq 0$$

Permettiamo ad ogni campione di trovarsi all'interno dello spazio del margine, volendo comunque ϵ piccoli.
Bisogna pertanto minimizzare:

$$\frac{1}{2}||w||^2 + C\sum_{n=1}^{N}\epsilon_n$$

considera una costante C per la regolarizzazione degli ϵ. A differenza dei problemi visti, più piccolo è C maggiore è il rilassamento del problema. Se C è più piccolo, ϵ può anche essere più grande, quindi il problema è più rilassato. C può essere pensato come un coefficiente di rilassamento.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

Loss function

Possiamo contare gli errori tramite una funzione che conta 1 se c'è un errore e 0 se non ci sono. La funzione è una **zero-one loss**. Tuttavia questo è un problema per l'ottimizzazione.

In alternativa mappiamo gli errori utilizzando una **hinge loss**. La hinge loss è:

$$l(t) = \max\{0, 1 - t\}$$

Dove:

$$t_a = y \cdot f(x_a) = y(< w, x_a > +b)$$

ossia è la distanza tra il punto x_a e l'iperpiano.

Notiamo che t può:

1. essere maggiore di 1. A noi sta bene che sia maggiore di uno: vogliamo infatti che il margine sia 1 e che i pattern positivi o negativi siano a distanza maggiore di 1. Ne segue allora che $1 - t < 0$, e $\max$ restituisce 0;
2. essere nel range $[0, 1]$. In tal caso c'è un errore di valore $1 - t$ identificato dalla funzione $\max$. L'errore sarà maggiore se t è più basso, ossia se il punto è molto vicino all'iperpiano.
3. essere minore di 0. $1 - t$ allora sarà maggiore di uno e vi è una **misclassification**: il prodotto tra y e $< w, x_a > +b$ è di segno negativo in quanto i termini sono discordi e la penalità è tanto più alta quanto maggiore è la distanza dall'iperpiano.

Dall'equazione del margine di Soft SVM, se isoliamo ϵ_n otteniamo:

$$\epsilon_n = 1 - y(< w, x_n) + b$$

e osserviamo anche che per ipotesi:

$$\epsilon_n \geq 0$$

Ossia, la hinge loss coincide con la slack variable. Minimizziamo pertanto la funzione:

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

$$\mathcal{L} = \frac{1}{2}||w||^2 + C\sum_{i=1}^{N}\max\{0, 1 - y(< w, x_i > +b)\}$$

Similmente alla logistic regression cerchiamo una linea, ma in questo caso stiamo introducendo un'idea di margine che ci permette di massimizzare la separazione degli spazi. Questo problema di ottimizzazione, inoltre, ha dei principi teorici che dimostrano l'ottenimento dell'ottimo teorico per problemi che non sono totalmente separabili.

L'ottimizzazione del problema può sempre essere attuata con la **discesa del gradiente**. Solitamente, inoltre, si preferisce l'utilizzo di una **versione smussata** della hinge loss.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

08 - Ensemble learning

L'**ensemble learning** è un approccio che consiste nella costruzione di un modello tramite la combinazione di modelli più semplici. Il modello risultante H deriva dalla combinazioni delle decisioni di modelli più semplici $h_1, ...h_L$. I diversi classificatori vengono allenati sui dati in maniera indipendente. Ogni singolo classificatore può anche avere natura differente (ad esempio possono essere tutti decision stump).

Combinazione dei modelli

Ci sono diversi modi per combinare l'output dei singoli learner:

- **averaging** tra le predizioni dei modelli, ossia tramite un sistema di voting;
- **media pesata** tra le classificazioni dei modelli, attribuendo un peso più alto in virtù dei risultati ottenuti da questi o tramite grid search;
- **mediana** dei classificatori;
- **minimo e massimo**;
- **produttoria** delle distribuzioni di probabilità, sempre scegliendo la classe più alta;
- **gating**. Il gating consiste sempre nel pesare in maniera diversa i classificatori, ma il peso è appreso tramite una funzione dipendente dall'input. Ossia, il gating **calcola pesi diversi per ogni input** e li normalizza tramite softmax. In questo modo se, ad esempio, un modello è bravo con dati lineari e un altro con dati non lineari, la gating function sarà in grado di attribuire un peso maggiore in virtù dell'input ricevuto;
- **stacking**. Si crea un nuovo feature vector dato dall'output dei classificatori. Un secondo layer, detto **meta-modello**, funge da classificatore sui parametri forniti dal layer precedente. Nella pratica si utilizza un validation set per generare le predizioni dei modelli base e si addestra il meta-modello per imparare i pesi ottimali per combinare gli output dei modelli.

Manipolazione del training data

Ci sono diverse tecniche per manipolare il training data adottate per l'ensemble learning.

Bootstrap aggregating

Il **bootstrap aggregating (bagging)** è un meta-algoritmo che permette di migliorare la stabilità e la precisione del learning del modello di classificazione tramite la divisione del training set. L'obiettivo è ridurre la varianza ma evitare l'overfitting.

Il bagging consiste nel prendere l'insieme di training I di dimensione n e generare m nuovi training set I_i dove ogni insieme ha un numero di campioni $n' \leq n$. Un campione si può ritrovare in tanti set, mentre è possibile che determinati campioni non appaiano mai; questi ultimi prendono il nome di **out of bag elements** (**OOB elements**). Matematicamente, se abbiamo n elementi e vogliamo costruire un set di n elementi che ammette ripetizioni la probabilità che un elemento non sia selezionato è $(\frac{n-1}{n})^n$. Infatti ad ogni passo la probabilità che un elemento sia selezionato è $\frac{1}{n}$, quindi l'inverso di questa probabilità è $1 - \frac{1}{n}$. Questa probabilità è ripetuta per gli n elementi che stiamo considerando. Per $n \to \infty$ il valore tende a 0.368, ossia il $36,8\%$ dei dati può essere utilizzato per il validation set.

Utilizziamo allora m modelli, allenando ciascun modello su uno degli I_i set di training. Si adotta infine una tecnica a votazione per combinare i risultati dei modelli. Ad esempio, se due modelli su tre classificano un determinato campione nella classe 3, allora il risultato di classificazione è 3.

Boosting

Il **boosting** è un altro meta algoritmo utilizzato per la creazione di un modello forte a partire da un set di weak learner. La differenza tuttavia risiede nel fatto che i modelli non sono costruiti in maniera parallela, come nel bagging, ma sono costruiti in maniera **sequenziale**. Ogni modello è allenato in modo tale da correggere gli errori commessi dai modelli precedenti. L'obiettivo in questo caso è migliorare la precisione riducendo la bias.

In generale l'idea che segue ogni algoritmo di boosting è quella di pesare i dati dopo l'aggiunta di un nuovo weak learner, dando maggiore peso ai **dati erroneamente classificati** e riducendo il peso di dati correttamente classificati. In questo modo i weak learner futuri potranno concentrarsi maggiormente sui dati erroneamente classificati.

L'algoritmo concepito da Freund e Schapire prende il nome di **AdaBoost**. L'algoritmo viene concepito per problemi di classificazione binaria, ma in realtà basta semplicemente modificare la funzione

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it

ipotesi $H(x)$ che viene restituita per risolvere problemi multiclasse. Inoltre, AdaBoost è efficiente ma non performa bene quando la relazione tra le feature è di natura polinomiale: in tal caso si ricorre al **GradientBoosting**.

AdaBoost consta dei seguenti passi:

1. si genera un vettore di pesi inizialmente pari a $[\frac{1}{n}, \frac{1}{n}, ..., \frac{1}{n}]$, dove n è il numero di esempi nel training set;
2. per $t = 1$ fino a T, dove T è il numero di iterazioni, si allena il modello $t-$esimo sul training set $X, y = \{(x_i, y_i)\}_{i=1}^{n}$ e considerando il vettore di pesi w_t. I pesi sono relativi agli errori che commette il classificatore $t-1$. Ossia, **per generare il classificatore h_t pesiamo i dati su cui il classificatore precedente ha commesso errore**;
3. si computa l'errore ϵ_t del modello h_t in virtù del vettore di pesi w_t. Tale errore è calcolato empiricamente tramite $\epsilon_t = \sum_{i|y_i \neq h_t(x_i)} w_{t,i}$
4. si sceglie $\beta_t = \frac{1}{2}\ln(\frac{1-\epsilon_t}{\epsilon_t})$ come peso da assegnare al classificatore;
5. si aggiornano i pesi in modo tale che $w_{t+1,i} = w_{t,i} \cdot e^{-\beta_t y_i h_t(x_i)}$ per $i = 1, ..., n$. Il peso $i-$esimo associato all'$i-$esimo dato (x_i, y_i) sarà uguale a $e^{-\beta}$ se il dato è classificato correttamente, mentre sarà pari a e^{β} se il dato è classificato erroneamente. Il segno è negativo in quanto $h_t(x_i) \cdot y_i$ sarà maggiore di zero quando il campione è classificato correttamente, infatti si ottiene y_i^2, che è un valore positivo. Il valore nuovo di w_i sarà quindi più grande quando si commette un errore;
6. si normalizza il vettore in modo tale che sia una distribuzione, ossia $w_{t+1,i} = \frac{w_{t+1,i}}{\sum_{j=1}^{n} w_{t+1,j}}$ per $i = 1, ..., n$;
7. si ripete il processo per tutte le iterazioni, fino a quando non si restituisce la funzione $H(x) = \text{sign}(\sum_{t=1}^{T} \beta_t h_t(x))$.

Ci chiediamo adesso come poter allenare il nostro classificatore per tenere conto del vettore di pesi w. Poiché la funzione di loss è scritta come una media pura, possiamo sostituire tale media con una **media pesata** utilizzando il vettore di pesi w e moltiplicando il termine di loss per il termine w_i. Se non si hanno funzioni di loss, ad esempio con i decision trees, possiamo tener conto di questa distribuzione tramite un **ricampionamento dei pattern**. Si replica il campionamento tenendo conto del fatto che, **maggiore è l'errore commesso sul pattern, maggiore è la probabilità di ripescarlo**: ciò è ottenibile seguendo come distribuzione il vettore w.

Il boosting è molto semplice da programmare, non ci sono iperparametri ad eccezione di T e non ci sono assunzioni sui weak learner. Una limitazione, tuttavia, risiede nella **parallelizzazione** del processo di allenamento dei weak learner: il boosting, infatti, richiede che i modelli siano allenati in maniera sequenziale.

Materiale fornito da noistudiamo.it | Contatti: info@noistudiamo.it