# Statistical methods for machine learning

Mauro Tellaroli

## Indice

# 1 Introduzione

## 1.1 Definizioni fondamentali

La *data inference* è lo studio dei metodi che utilizzano i dati per predirre il futuro. Il *Machine Learning* è uno strumento potente che può essere usato per risolvere una grossa parte dei problemi di *data inference*, inclusi i seguenti:

- **Clustering**: raggruppare i *data points* in base alle loro similarità;
- **Prediction**: assegnare delle etichette (*label*) ai *data points*;
- **Generation**: generare nuovi *data points*;
- **Control**: eseguire una sequenza di azioni in un ambiente con l'obiettivo di massimizzare una nozione di utilità.

Con *data point* si intende una serie di informazioni legate ad un unico elemento; un'analogia può essere un *record* in un database.

Gli algoritmi che risolvono una *learning task* in base a dei dati già semanticamente etichettati lavorano in modalità ***supervised learning***. A etichettare i dati saranno delle persone o la natura. Un esempio dell'ultimo caso sono le previsioni del meteo. D'altra parte, gli algoritmi che utilizzano i dati senza la presenza di etichette lavorano in modalità ***unsupervised learning***.

In questo corso ci si focalizzerà sul *supervised learning* e la progettazione di sistemi di *machine learning* il cui obiettivo è apprendere dei **predittori**, ovvero funzioni che mappano i *data points* alla loro etichetta.

### 1.1.1 *Label set* $\mathcal{Y}$

Verrà usata $\mathcal{Y}$ per indicare il *label set*, ovvero l'insieme di tutte le possibili etichette di un *data point*. Le etichette potranno essere di due tipi differenti:

1. **Categoriche** ($\mathcal{Y} = \{\text{sport}, \text{politica}, \text{economia}\}$): si parlerà di problemi di **classificazione**;
2. **Numeriche** ($\mathcal{Y} \subseteq \mathbb{R}$): si parlerà di problemi di **regressione**.

È importante sottolineare come la reale differenza tra le due tipologie di etichetta sia il significato e non la sua rappresentazione in quanto, si potrà sempre codificare un'etichetta categorica in un numero.

A sottolineare ciò è il fatto che nella regressione l'errore è tipicamente una funzione della differenza $|y - \hat{y}|$, dove $\hat{y}$ è la predizione di $y$. Nella classificazione, invece, l'errore è tipicamente binario: predizione corretta ($\hat{y} = y$) o errata ($\hat{y} \neq y$).

Quando ci sono solo due possibili etichette ($|\mathcal{Y}| = 2$), si ha un **problema di classificazione binario** e, convenzionalmente, verrà usata una codifica numerica $\mathcal{Y} = \{-1, 1\}$.

### 1.1.2 *Loss function* $\ell$

Come già visto precedentemente, si vuole misurare l'errore che un predittore commette su una determinata predizione. Per farlo si userà una **funzione di loss** $\ell$ non negativa che misurerà la discrepanza $\ell(y, \hat{y})$ tra l'etichetta predetta $\hat{y}$ e quella corretta $y$. Si assumerà sempre $\ell(y, \hat{y}) = 0$ quando $\hat{y} = y$.

La funzione di loss più semplice per la classificazione è la ***zero-one loss***:

$$\ell(y, \hat{y}) = \begin{cases} 0 & y = \hat{y} \\ 1 & \text{altrimenti} \end{cases}$$

Nella regressione, le tipiche funzioni di loss sono:

- la ***absolute loss***: $\ell(y, \hat{y}) = |y - \hat{y}|$

- la ***quadratic loss***: $\ell(y, \hat{y}) = (y - \hat{y})^2$

In alcuni casi può essere conveniente scegliere l'etichetta predetta da un insieme $\mathcal{Z}$ diverso da $\mathcal{Y}$. Per esempio, si consideri il problema di assegnare una probabilità $\hat{y} \in (0, 1)$ all'evento $y =$ "pioverà domani". In questo caso, $\mathcal{Y} = \{$"piove", "non piove"$\}$ e $\mathcal{Z} = (0, 1)$. Indicando questi due eventi con 1 (piove) e 0 (non piove), si può usare una funzione di loss per la regressione, come la *absolute loss*:

$$\ell(y, \hat{y}) = |y - \hat{y}| = \begin{cases} 1 - \hat{y} & y = 1 \quad \text{(piove)} \\ \hat{y} & y = 0 \quad \text{(non piove)} \end{cases}$$

Per penalizzare maggiormente le predizioni che distano troppo dalla realtà, si può usare una ***logarithmic loss***:

$$\ell(y, \hat{y}) = \begin{cases} \ln \frac{1}{\hat{y}} & y = 1 \quad \text{(piove)} \\ \ln \frac{1}{1-\hat{y}} & y = 0 \quad \text{(non piove)} \end{cases}$$

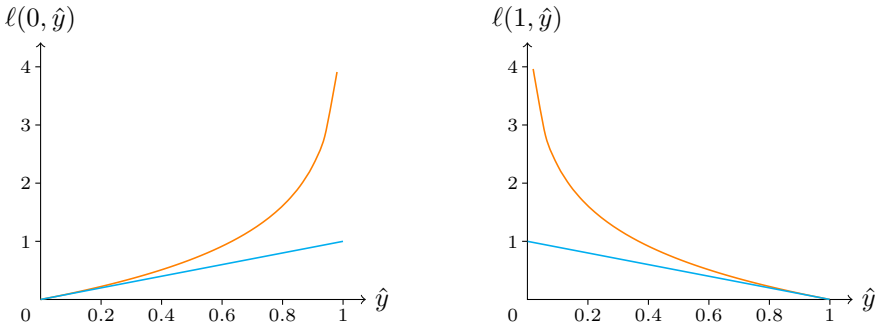



Figura 1: Confronto tra *absolute loss* e *logarithmic loss*; a sinistra il caso $y = 0$, a destra $y = 1$.

Si noti in figura 1 come la *logarithmic loss* tenda ad infinito quando la predizione è opposta all'etichetta reale:

$$\lim_{\hat{y} \to 1^-} \ell(0, \hat{y}) = \lim_{\hat{y} \to 0^+} \ell(1, \hat{y}) = +\infty$$

In pratica questo previene l'utilizzo di predizioni $\hat{y}$ troppo sicure, quindi troppo vicine a zero o uno.

### 1.1.3 *Data domain* $\mathcal{X}$

Verrà usata $\mathcal{X}$ per indicare l'insieme dei *data points*; ogni suo punto $x \in \mathcal{X}$ è tipicamente un record di un database formato da *feature*:

$$x = (x_1, \ldots, x_d)$$

Spesso un *data point* può essere codificato come un vettore i cui elementi sono le sue *feature*. Questa codifica risulta naturale in presenza di quantità omogenee, come i pixel di un'immagine o una lista di occorrenze di parole in un testo. Quando invece i dati presenti utilizzano unità di misura differenti, come "età" e "altezza", la codifica non risulta più immediata. Ci sarà bisogno di una procedura che codifichi i dati in modo da ottenere uno spazio vettoriale omogeneo e coerente con i dati iniziali.

In questo corso si assumerà che i dati possano essere rappresentati da vettori di numeri:

$$\mathcal{X} \equiv \mathbb{R}^d$$

### 1.1.4 Predittori $f$

Un **predittore** è una funzione $f : \mathcal{X} \to \mathcal{Y}$ che mappa i *data points* alle etichette (o $f : \mathcal{X} \to \mathcal{Z}$). Sì può quindi dire che in un problema di predizione l'obiettivo è ottenere una funzione $f$ che genera delle predizioni $\hat{y} = f(x)$ tali che $\ell(y, \hat{y})$ sia basso per il maggior numero di punti $x \in \mathcal{X}$ osservati. In pratica, **la funzione $f$ è definita da un certo numero di parametri in un dato modello**. Un esempio sono i parametri di una rete neurale.

### 1.1.5 Esempi

Nel *supervised learning* un **esempio** è una coppia $(x, y)$ dove $x$ è un *data point* e $y$ la sua reale etichetta.

In alcuni casi $x$ ha un'unica $y$, come nel caso in cui $y$ rappresenta una proprietà oggettiva di $x$; in altri casi, invece, $x$ può avere diverse $y$ associate, come quando le $y$ sono soggettivamente assegnate da persone.

### 1.1.6 *Test set* e *test error*

Per poter stimare la qualità di un predittore si usa un insieme di esempi detto ***test set***:

$$\{(x'_1, y'_1), \ldots, (x'_n, y'_n)\}$$

Data una *loss function* $\ell$, il *test set* viene usato per calcolare il ***test error*** $\ell_{S'}$ di un predittore $f$:

$$\ell_{S'}(f) = \frac{1}{n} \sum_{t=1}^{n} \ell(\underbrace{y'_t}_{\text{reale}}, \overbrace{f(x'_t)}^{\text{predetta}})$$

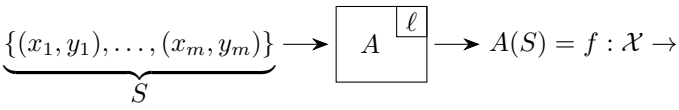
Il *test error* ha quindi lo scopo di calcolare la prestazione media del predittore su dei dati reali.

### 1.1.7 *Learning algorithm* $A$

Si definisce *training set* $S$ un insieme di esempi:

$$S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$$

che viene usato dal ***learning algorithm*** $A$ per produrre un predittore $A(S)$. Informalmente, il *learning algorithm* "impara" dal *training set*.




Il *test set* e il *training set* vengono solitamente prodotti assieme attraverso un processo di collezione dati e etichettamento. Dato l'insieme di esempi preparati, questo verrà partizionato in *test set* e *training set*, tipicamente tramite una divisione casuale. **Obiettivo del corso è lo sviluppo di una teoria che ci guidi nella progettazione di *learning algorithm* che generano predittori con un basso *test error*.**

### 1.1.8 *Training error* $\ell_S$

Sia $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ il *training set*; viene definito, equivalentemente al *test error*, il ***training error***:

$$\ell_S(f) = \frac{1}{m} \sum_{t=1}^{m} \ell(y_t, f(x_t))$$

**Un approccio intuitivo alla progettazione di *learning algorithm* è quello di assumere che il *training error* $\ell_S(f)$ del predittore $f$ sia correlato con il suo *test error*.**

## 1.2 Empirical Risk Minimization (ERM)

### 1.2.1 Definizione

Sia $\mathcal{F}$ un insieme di predittori e $\ell$ una *loss function*. L'*empirical risk minimizer* (ERM) è il *learning algorithm* $A$ che restituisce un predittore in $\mathcal{F}$ che **minimizza il *training error***:

$$A(S) \in \underset{f \in \mathcal{F}}{\operatorname{argmin}}\, \ell_S(f)$$

Si noti come $A(S)$ appartenga e non uguagli il minimo; questo perchè ci potrebbero essere più $f \in \mathcal{F}$ che minimizzano $\ell_S(f)$.

### 1.2.2 Predittori con *test error* elevato

Quando in $\mathcal{F}$ tutti i predittori hanno un *test error* alto, ERM produrrà un pessimo predittore. **Per trovare un buon predittore, ovvero un predittore con un *test error* basso, ci sarà quindi bisogno che $\mathcal{F}$ sia sufficientemente grande**.

**Tuttavia, se $\mathcal{F}$ è troppo grande, anche in questo caso verrà prodotto un pessimo predittore**. Un esempio è il seguente.

Si consideri il seguente problema "giocattolo":

$$\mathcal{Y} = \{-1, 1\} \qquad \mathcal{X} = \{x_1, x_2, x_3, x_4, x_5\}$$

Si prenda l'insieme $\mathcal{F}$ contenente un classificatore $f : \mathcal{X} \to \mathcal{Y}$ per ognuna delle possibili combinazioni di etichettamento dei cinque *data points*. $\mathcal{F}$ sarà quindi formata da $2^5 = 32$ classificatori:

$$\mathcal{F} = \{f_1, \ldots, f_{32}\}$$

| $\mathcal{F}$ | $f(x_1)$ | $f(x_2)$ | $f(x_3)$ | $f(x_4)$ | $f(x_5)$ |
|---|---|---|---|---|---|
| $f_1$ | 1 | 1 | 1 | 1 | 1 |
| $f_2$ | 1 | 1 | 1 | 1 | -1 |
| $f_3$ | 1 | 1 | 1 | -1 | 1 |
| $f_4$ | 1 | 1 | 1 | -1 | -1 |
| $f_5$ | 1 | 1 | -1 | 1 | 1 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $f_{31}$ | -1 | -1 | -1 | -1 | 1 |
| $f_{32}$ | -1 | -1 | -1 | -1 | -1 |

Si supponga che il *training set* $S$ contenga solo tre *data points* qualsiasi e il *test set* contenga gli altri due. Sia $f^*$ il predittore usato per etichettare i dati che quindi avrà zero *test* e *training error*; ogni etichetta $y_t$ sarà quindi ottenuta da $f^*$:

$$y_t = f^*(x_t) \quad \forall t = 1, \ldots, 5$$

Per rendere l'idea, si prenda come esempio:

$$f^* = f_3$$

$$\begin{aligned} S &= \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\} \\ &= \{(x_1, 1), (x_2, 1), (x_3, 1)\} \end{aligned}$$

Nonostante ad avere *test error* nullo sia solo $f_3$, ad avere il *training error* nullo sono i quattro classificatori che hanno $y_1, y_2, y_3 = 1$ ovvero $f_1, f_2, f_3, f_4$. Questo perchè il *training set* $S$ contiene solo i primi 3 *data points*.

Siamo quindi nella situazione in cui ERM trova più predittori con $\ell_S$ minimo e non ha abbastanza informazioni per capire quale di questi sia migliore a livello di *test error*.

**Il problema dell'esempio appena visto è che $\mathcal{F}$ è troppo grande rispetto al *training set*.** La domanda che sorge spontanea è quindi: Quanto deve essere grande $\mathcal{F}$ per poter ottenere un buon predittore tramite ERM?

**La teoria dell'informazione ci suggerisce che $S$ debba avere cardinalità** $\log_2 |\mathcal{F}|$ o, viceversa, $\mathcal{F}$ debba avere cardinalità $2^m$. Quindi, nell'esempio di prima, il *training set* avrebbe dovuto contenere almeno $\log_2 |\mathcal{F}| = 5$ *data points*.

### 1.2.3 *Overfitting* e *underfitting*

I due eventi visti nella sezione precedente, che portano alla generazione di un predittore con *test set* elevato, vengono chiamati:

- ***Underfitting***: si verifica quando il *training error* è elevato;
- ***Overfitting***: si verifica quando il *training error* è basso ma il *test error* è alto.

Quando $A$ è ERM e $S$ ha dimensione fissata $|S| = m$:

- Ci si aspetta *overfitting* quando $\log_2 |\mathcal{F}| \gg m$;
- Ci si aspetta *underfitting* quando $\log_2 |\mathcal{F}| \ll m$.

### 1.2.4 Etichette rumorose

Il fenomeno dell'*overfitting* spesso accade quando le etichette sono rumorose, ovvero quando le etichette $y$ non sono deterministicamente associate con i *data points* $x$. Questo può accadere per i seguenti motivi (non mutuamente esclusivi tra loro):

1. **Incertezza umana**: se ad etichettare $S$ sono delle persone, ci sarà dell' incertezza in quanto persone diverse potrebbero avere opinioni diverse;
2. **Incertezza epistemica**: ogni *data point* è rappresentato da un vettore delle *feature* che non contiene abbastanza informazioni per determinare univocamente l'etichetta;
3. **Incertezza aleatoria**: il vettore delle *feature* che rappresenta il *data point* è ottenuto attraverso delle misurazioni rumorose.

Le etichette rumorose portano all'*overfitting* perchè possono ingannare l'algoritmo su quale sia la "vera" etichetta di una certo *data point*.

# 2 Gli algoritmi *Nearest Neighbor*

## 2.1 *Nearest Neighbor* (NN)

### 2.1.1 Definizione

Verrà introdotto ora l'algoritmo di ***Nearest Neighbor* (NN)** per la classificazione binaria con *feature* numeriche:

$$\mathcal{X} = \mathbb{R}^d \qquad \mathcal{Y} = \{-1, 1\}$$

NN non è un'istanza di ERM in quanto non punta a minimizzare $\ell_S$.

**L'idea di NN è la sueguente:**

- **Predici ogni punto del *training set* con la propria etichetta;**
- **Predici gli altri punti con l'etichetta del punto del *training set* che è più vicino al punto interessato.**

Più formalmente, dato un *training set*:

$$S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$$

l'algoritmo $A_{\text{NN}}$ genera un classificatore $h_{\text{NN}} : \mathbb{R} \to \{-1, 1\}$ definito come segue:

$$h_{\text{NN}}(x) = \text{etichetta } y_t \text{ del punto } x_t \in S \text{ più vicino a x}$$

Se a minimizzare la distanza con $x$ sono più punti, si predirrà l'etichetta più presente tra i putni vicini. Se non c'è una maggioranza di etichette tra i punti più vicini si predirrà un valore di default $\in \{-1, 1\}$.

Presi due punti $x = (x_1, \ldots, x_d)$ e $x_t = (x_{t,1}, \ldots, x_{t,d})$, la distanza $||x - x_t||$ verrà calcolata tramite la distanza euclidea:

$$||x - x_t|| = \sqrt{\sum_{i=1}^{d} (x_i - x_{t,i})^2}$$

Ogni classificatore binario $f : \mathbb{R}^d \to \{-1, 1\}$ partiziona $\mathbb{R}^d$ in due regioni (come mostrato in figura 2):

$$\{x \in \mathbb{R}^d : f(x) = 1\} \quad , \quad \{x \in \mathbb{R}^d : f(x) = -1\}$$

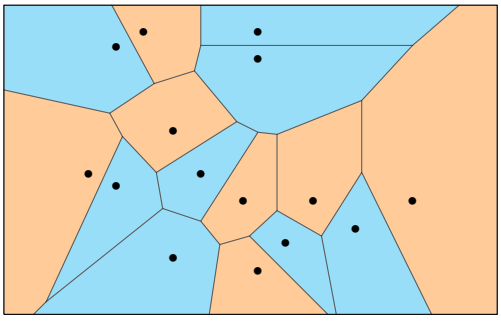


Figura 2: Diagramma di Voronoi in $\mathbb{R}^2$; tutti i punti $x$ interni a una cella con centro $\bullet\, x_t$ sono tali che $h_{\text{NN}}(x) = y_t$

### 2.1.2 Efficienza ed efficacia

Siccome il funzionamento di NN implica la memorizzazione di tutto il *training set*, **l'algoritmo non scala bene con il numero di $|S| = m$ di *training point***. Inoltre, calcolare un qualsiasi $h_{\text{NN}}(x)$ è costoso, in quanto richiede di calcolare la distanza tra $x$ e tutti gli altri punti di $S$; questo in $\mathbb{R}^d$ comporta un costo di $\Theta(dm)$.

Infine, si noti come, vista la completa memorizzazione di $S$, **NN generi sempre un classificatore $h_{\text{NN}}$ con *training error* nullo**:

$$\ell_S(h_{\text{NN}}) = 0$$

## 2.2 $k-$Nearest Neighbor ($k-$NN)

### 2.2.1 Definizione

Partendo dagli algoritmi NN, si può ottenere una famiglia di algoritmi detta $k-$NN; il parametro $k$ assume tipicamente i valori $k = 1, 3, 5, \ldots$ con $k < |S|$.

Questi algoritmi sono definiti come segue: dato un *training set* $S$ e un punto $x \in \mathcal{X}$, $k-$NN genererà un predittore $h_{k\text{-NN}}$ tale che:

$$h_{k\text{-NN}}(x) = \text{etichetta } y_t \text{ appartenente alla maggioranza dei } k \text{ punti più vicini a } x$$

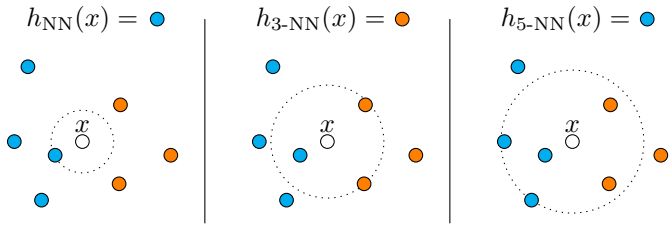



Figura 3: Esempi di $h_{k\text{-NN}}$ con $\mathcal{X} = \mathbb{R}^2$; si noti come, con lo stesso *training set*, la predizione cambia al variare di $k$.

### 2.2.2 Efficienza ed efficacia

A livello di efficienza $k-$NN soffre degli stessi problemi di NN vista la memorizzazione dell'intero *training set*.

Per quanto riguarda la sua efficacia invece, $k-$NN non ha sempre un *training error* nullo:

$$\ell_S(h_{k\text{-NN}}) \geq 0$$

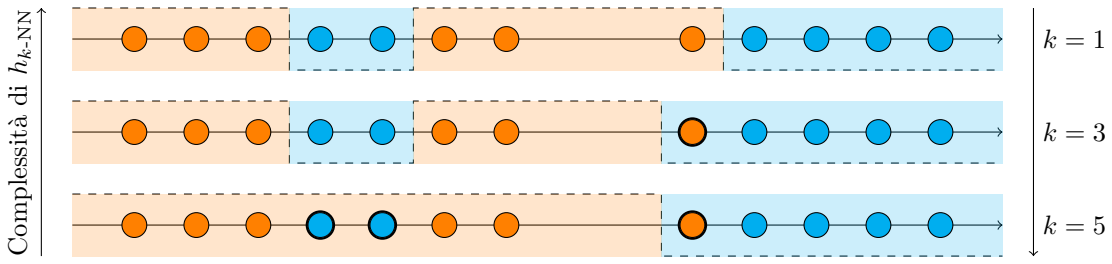



Figura 4: Esempi di $h_{k\text{-NN}}$ con $\mathcal{X} = \mathbb{R}$.

Come si può infatti notare dalla figura 4, nei casi con $k = 3$ e $k = 5$ sono presenti punti errati (evidenziati in grassetto) considerati dal classificatore come *outlier*. Inoltre **al crescere di $k$**

**cresce anche la "semplicità" del classificatore così come il numero di punti errati**. L'estremo di ciò è quando $k = |S|$; in questo caso infatti $h_{k\text{-NN}}$ diventa un classificatore costante che predice sempre l'etichetta più presente in tutto $S$.

In un generico classificatore $h_{k\text{-NN}}$ tipicamente succede che:

- **Se $k$ è troppo basso** si ottiene un classificatore che si "fida" troppo del *training set*, ottenendo quindi ***overfitting***;
- **Se $k$ è troppo alto**, si ottiene un classificatore troppo semplice, ottenendo ***underfitting***.

Tutti i classificatori introdotti fino ad'ora sono classificatori binari ($|\mathcal{Y}| = 2$). Tuttavia $k-$NNpuò essere usato anche per:

- problemi di classificazione multiclasse ($|\mathcal{Y}| > 2$): si opera come nel caso binario, predicendo quindi l'etichetta più presente nei $k$ punti più vicini;
- problemi di regressione ($\mathcal{Y} = \mathbb{R}$): si predice la media aritmetice delle etichette dei $k$ punti più vicini.

# 3 *Tree Predictors*

## 3.1 Definizione

Come già visto, mentre alcuni tipi di dato hanno una naturale rappresentazione vettoriale $x \in \mathbb{R}^d$, altri non ce l'hanno. Un esempio possono essere dei *record* medici, dove i dati contengono i seguenti campi:

$$\texttt{età} \in \{12, \dots, 90\}$$
$$\texttt{fumatore} \in \{\text{sì}, \text{no}, \text{ex}\}$$
$$\texttt{peso} \in [10, 200]$$
$$\texttt{sesso} \in \{\text{M}, \text{F}\}$$
$$\texttt{terapia} \in \{\text{antibiotici}, \text{cortisone}, \text{nessuna}\}$$

Anche convertendo questi tipi di dato in dati numerici, gli algoritmi basati sulla distanza euclidea, come il $k-$NN, potrebbero non andare molto bene.

**Per poter applicare la *data inference* su dati le cui *feature* variano in insiemi eterogenei $\mathcal{X}_1, \dots, \mathcal{X}_d$, verrà introdotta una nuova famiglia di predittori: i *tree predictors*.**

Un *tree predictor* è un albero ordinato e radicato dove ogni nodo può essere una **foglia** o un **nodo interno**. È importante sottolineare che in un albero ordinato i figli di ogni nodo sono anch'essi ordinati e quindi numerabili consecutivamente. In figura 5 viene mostrato un esempio di *tree predictor* binario le cui *feature* sono:

$$\texttt{previsione} \in \{\text{sole}, \text{nuvole}, \text{pioggia}\}$$
$$\texttt{umidità} \in [0, 100]$$
$$\texttt{vento} \in \{\text{sì}, \text{no}\}$$

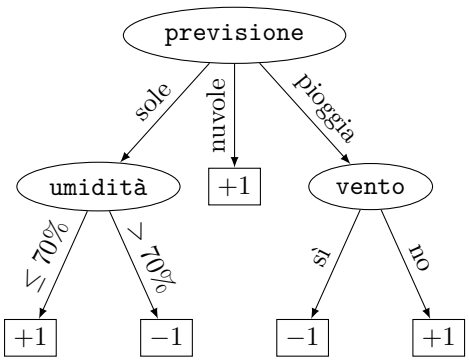



Figura 5: Esempio classico di *tree classifier* per una classificazione binaria.

Sia $\mathcal{X} = \mathcal{X}_1, \dots, \mathcal{X}_d$, dove ogni $\mathcal{X}_i$ rappresenta il dominio dell'$i$-esimo attributo (o *feature*) $x_i$. **Il *tree predictor* $h_T : \mathcal{X} \to \mathcal{Y}$ è un predittore definito da un albero $T$ i cui nodi interni corrispondono a dei test e le cui foglie corrispondono a delle etichette $y \in \mathcal{Y}$.**

Un test su un attributo $i$ su un nodo interno con $k$ figli è una funzione $f : \mathcal{X} \to \{1, \dots, k\}$. $f$ mappa ogni elemento di $\mathcal{X}_i$ a un nodo figlio. Due esempi possono essere i seguenti:

$$\mathcal{X}_i = \{a, b, c, d\} \qquad k = 3$$
$$f(x_i) = \begin{cases} 1 & x_i = c \\ 2 & x_i = d \\ 3 & x_i \in \{a, b\} \end{cases}$$

$$\mathcal{X}_i = [0, 100] \qquad k = 2$$
$$f(x_i) = \begin{cases} 1 & x_i \in [0, 70] \\ 2 & x_i \in (70, 100] \end{cases}$$

L'esempio di destra è riferito all'attributo `umidità` di figura 5.

La predizione $h_T(x)$ è calcolata come segue:

1. $v \leftarrow r$ ($r$ è la radice di $T$)
2. se $v$ è una foglia $\ell$, si restituisce l'etichetta $y \in \mathcal{Y}$ associata a $\ell$;
3. altrimenti, sia $f : \mathcal{X}_i \to \{1, \ldots, k\}$ il test associato a $v$, **assegna** $v \leftarrow v_j$ dove $j = f(x_i)$ e $v_j$ indica il $j$-esimo figlio di $v$;
4. vai al punto 2.

Se $h_T(x)$ restituisce la foglia $\ell$, si dirà che l'esempio $x$ è indirizzato a $\ell$.

## 3.2 Costruzione di un *tree predictor*

### 3.2.1 Idea generale

Dato un *training set* $S$, si vedrà ora come costruire un *tree predictor*. Per semplicità, si guarderà solo ad una classificazione binaria $\mathcal{Y} = \{-1, 1\}$ e verranno usati solo alberi binari completi, cioè alberi dove ogni nodo interno ha due figli.

**L'idea è quella di far crescere l'albero partendo da un singolo nodo (che dovrà essere una foglia)**. L'etichetta di quest'unica foglia sarà l'etichetta $\hat{y} \in \mathcal{Y}$, ovvero l'etichetta più presente nel *training set*. Si avrà quindi inizialmente, un classificatore che assegna a tutti i *data point* l'etichetta $\hat{y}$. **L'albero sarà fatto crescere scegliendo una foglia e rimpiazzandola con un nodo interno e due nuove foglie**.

### 3.2.2 *Training error*

Si chiami $T$ l'albero cresciuto fino a un certo punto e $h_T$ il classificatore corrispondente. **Obiettivo è calcolare il contributo che ogni foglia dà al *training error* $\ell_S(h_T)$.**

**Presa una foglia $\ell$, si vuole capire che etichetta assegnarle per minimizzare $\ell_S$.**

Si definisca:

$$S_\ell = \{(x_t, y_t) \in S : x_t \text{ è indirizzato a } \ell\}$$

$S_\ell$ è quindi l'insieme degli esempi di *training* che sono indirizzati alla foglia $\ell$. Si divida ora $S_\ell$ in due sottoinsiemi:

$$S_\ell^+ = \{(x_t, y_t) \in S_\ell : y_t = +1\}$$
$$S_\ell^- = \{(x_t, y_t) \in S_\ell : y_t = -1\}$$

Il primo conterrà tutti gli esempi di *training* che vengono indirizzati a $\ell$ con etichetta positiva mentre il secondo con etichetta negativa. Di questi insiemi si prenda il loro numero di elementi:

$$N_\ell^+ = |S_\ell^+| \qquad N_\ell^- = |S_\ell^-| \qquad N_\ell = |S_\ell|$$

È facile capire che se la maggior parte degli esempi di *training* che vengono indirizzati alla foglia $\ell$ hanno etichetta positiva, allora l'etichetta che bisognerà dare a $\ell$, per minimizzare il suo errore $\ell_S$, sarà l'etichetta positiva (chiaramente lo stesso discorso vale per l'etichetta negativa); questa intuizione può essere quindi usata per assegnare l'etichetta $y_\ell$ alla foglia $\ell$ nel seguente modo:

$$y_\ell = \begin{cases} +1 & N_\ell^+ \geq N_\ell^- \\ -1 & \text{altrimenti} \end{cases}$$

**Di conseguenza la foglia $\ell$ sbaglierà la sua previsione su** $\min\{N_\ell^+, N_\ell^-\}$ esempi di *training*. Per facilitare delle successive osservazioni moltiplichiamo e dividiamo per $N_\ell$:

$$\min\{N_\ell^+, N_\ell^-\} = \min\left\{\frac{N_\ell^+}{N_\ell}, \frac{N_\ell^-}{N_\ell}\right\} N_\ell$$

Quindi se il valore appena scritto è l'errore che una singola foglia $\ell$ fa, il *training error* sarà:

$$\ell_S(h_T) = \frac{1}{m}\sum_{\ell} \min\left\{\frac{N_\ell^+}{N_\ell}, \frac{N_\ell^-}{N_\ell}\right\} N_\ell$$
$$= \frac{1}{m}\sum_{\ell} \psi\left(\frac{N_\ell^+}{N_\ell}\right) N_\ell$$

Dove viene introdotta la funzione $\psi$, definita in $[0,1]$:

$$\psi(a) = \min\{a, 1-a\}$$

Si può facilmente intuire come $N_\ell^+/N_\ell$ e $N_\ell^-/N_\ell$ siano sempre compresi tra 0 e 1 in quanto rappresentano la percentuale di esempi positivi/negativi che raggiungono $\ell$ rispetto al totale degli esempi (sempre che raggiungono $\ell$).

**Esempio**

Sia $T$ l'albero di figura 6 e $S$ il *training set* mostrato in tabella 1 (vengono mostrati solo gli esempi di $S$ che sono indirizzati a $\ell'$ e $\ell''$). Si deve decidere che etichette assegnare alle foglie $\ell'$ e $\ell''$;

| $x_t$ | previsione | umidità | vento | $y_t$ |
|---|---|---|---|---|
| $x_1$ | sole | 85 | no | $+1$ |
| $x_2$ | sole | 76 | sì | $-1$ |
| $x_3$ | sole | 55 | sì | $+1$ |
| $x_4$ | sole | 65 | sì | $-1$ |
| $x_5$ | sole | 82 | sì | $-1$ |
| $x_6$ | sole | 35 | no | $+1$ |
| $x_7$ | sole | 94 | no | $-1$ |
| $x_8$ | sole | 66 | no | $+1$ |
| $x_9$ | sole | 48 | sì | $+1$ |

Tabella 1: Esempio di *training set*

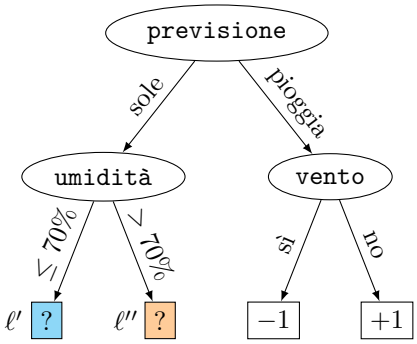



Figura 6: Esempio di *tree classifier* "in costruzione".

Si prenda $\ell'$:

$$S_{\ell'} = \{(x_3,+1),(x_4,-1),(x_6,+1),(x_8,+1),(x_9,+1)\} \quad N_{\ell'} = 5$$
$$S_{\ell'}^+ = \{(x_3,+1)(x_6,+1),(x_8,+1),(x_9,+1)\} \quad N_{\ell'}^+ = 4 \quad \frac{N_{\ell'}^+}{N_{\ell'}} = 0.8$$
$$S_{\ell'}^- = \{(x_4,-1)\} \quad N_{\ell'}^- = 1 \quad \frac{N_{\ell'}^-}{N_{\ell'}} = 0.2$$

L'ottanta percento degli esempi che raggiungono $\ell'$ ha etichetta positiva, si può quindi affermare che l'etichetta $y_{\ell'} = +1$.

Si prenda infine $\ell''$:

$$S_{\ell''} = \{(x_1,+1),(x_2,-1),(x_5,-1),(x_7,-1)\} \quad N_{\ell''} = 4$$
$$S_{\ell''}^+ = \{(x_1,+1)\} \quad N_{\ell''}^+ = 1 \quad \frac{N_{\ell''}^+}{N_{\ell''}} = 0.25$$
$$S_{\ell''}^- = \{(x_2,-1),(x_5,-1),(x_7,-1)\} \quad N_{\ell''}^- = 3 \quad \frac{N_{\ell''}^-}{N_{\ell''}} = 0.75$$

Il settantacinque percento degli esempi che raggiungono $\ell''$ ha etichetta negativa, si può quindi affermare che l'etichetta $y_{\ell''} = -1$.

### 3.2.3 Crescita dell'albero e *training error*

Si supponga di sostituire una foglia $\ell$ con un nodo interno e due nuove foglie $\ell'$ e $\ell''$, come mostrato in figura 7. **Può il *training error* del nuovo albero espanso crescere rispetto a quello originale?**

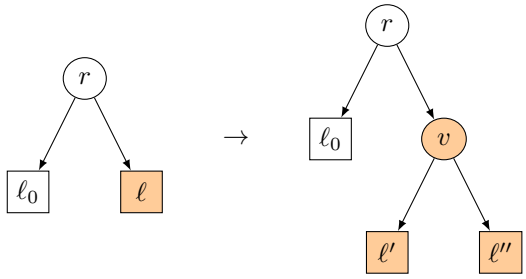



Figura 7: Un passaggio della crescita dell'albero: la foglia $\ell$ viene rimpiazzata da un nodo interno $v$ con due nuove foglie $\ell'$ e $\ell''$.

L'apporto che la foglia $\ell$ dà al *training error* è:

$$\psi\left(\frac{N_\ell^+}{N_\ell}\right) N_\ell \tag{1}$$

Gli esempi con etichetta positiva che vengono indirizzati a $\ell$ saranno ora divisi tra $\ell'$ e $\ell''$:

$$N_\ell^+ = N_{\ell'}^+ + N_{\ell''}^+ \tag{2}$$

Si può quindi ottenere che:

$$\begin{aligned}
\frac{N_\ell^+}{N_\ell} &= \frac{N_{\ell'}^+ + N_{\ell''}^+}{N_\ell} \\
&= \frac{N_{\ell'}^+}{N_\ell} + \frac{N_{\ell''}^+}{N_\ell} \\
&= \frac{N_{\ell'}^+}{N_\ell} \cdot \frac{N_{\ell'}}{N_{\ell'}} + \frac{N_{\ell''}^+}{N_\ell} \cdot \frac{N_{\ell''}}{N_{\ell''}} \\
&= \frac{N_{\ell'}^+}{N_{\ell'}} \cdot \frac{N_{\ell'}}{N_\ell} + \frac{N_{\ell''}^+}{N_{\ell''}} \cdot \frac{N_{\ell''}}{N_\ell}
\end{aligned} \tag{3}$$

Dati (1) e (3) si può dire che l'apporto di $\ell$ è:

$$\psi\left(\frac{N_\ell^+}{N_\ell}\right) N_\ell = \psi\left(\frac{N_{\ell'}^+}{N_{\ell'}} \cdot \frac{N_{\ell'}}{N_\ell} + \frac{N_{\ell''}^+}{N_{\ell''}} \cdot \frac{N_{\ell''}}{N_\ell}\right) N_\ell \tag{4}$$

Si noti che $\psi$ è una funzione concava. Questo permette di poter applicare la disuguaglianza di Jensen (valida con $a, b \in \mathbb{R} \wedge \mu \in [0, 1]$):

$$\psi(a\mu + b(1-\mu)) \geq \mu\psi(a) + (1-\mu)\psi(b) \tag{Jensen}$$

$$\begin{aligned}
\psi\left(\frac{N_{\ell'}^+}{N_{\ell'}} \cdot \frac{N_{\ell'}}{N_\ell} + \frac{N_{\ell''}^+}{N_{\ell''}} \cdot \frac{N_{\ell''}}{N_\ell}\right) N_\ell &\geq \cancel{N_\ell} \frac{N_{\ell'}}{\cancel{N_\ell}} \psi\left(\frac{N_{\ell'}^+}{N_{\ell'}}\right) + \cancel{N_\ell} \frac{N_{\ell''}}{\cancel{N_\ell}} \psi\left(\frac{N_{\ell''}^+}{N_{\ell''}}\right) \\
&\geq \psi\left(\frac{N_{\ell'}^+}{N_{\ell'}}\right) N_{\ell'} + \psi\left(\frac{N_{\ell''}^+}{N_{\ell''}}\right) N_{\ell''}
\end{aligned} \tag{5}$$

Chiaramente si dovrà avere che:

$$\frac{N_{\ell''}}{N_\ell} = 1 - \frac{N_{\ell'}}{N_\ell}$$

Questo è facilmente verificabile a partire da (2). Infine, dati (4) e (5) si ha:

$$\underbrace{\psi\left(\frac{N_\ell^+}{N_\ell}\right)N_\ell}_{\text{apporto di } \ell} \geq \underbrace{\psi\left(\frac{N_{\ell'}^+}{N_{\ell'}}\right)N_{\ell'}}_{\text{apporto di } \ell'} + \underbrace{\psi\left(\frac{N_{\ell''}^+}{N_{\ell''}}\right)N_{\ell''}}_{\text{apporto di } \ell''}$$

Questo dimostra che, **facendo crescere l'albero, il *training error* non aumenta**.

**Una foglia $\ell$ viene detta pura se non contribuisce ad aumentare il *training error***, ovvero:

$$N_\ell^+ \in \{0, N_\ell\}$$

In qualsiasi albero il *training error* è maggiore di zero ($\ell_S(h_T) > 0$) almeno che non abbia solo foglie pure.

### 3.2.4 Algoritmo generale

Verrà ora mostrato un algoritmo generale per la cotruzione di un albero binario a partire da un *training set* $S$:

1. **Inizializzazione:**
   (a) Crea un albero $T$ con solo la radice $\ell$
   (b) $S_\ell = S$
   (c) $y_\ell =$ etichetta più frequente in $S_\ell$
2. ***Main loop*:**
   (a) Scegli una foglia $\ell$ e sostituiscila con un nodo interno $v$ e due nuove foglie $\ell'$ e $\ell''$
   (b) Scegli un attributo $i$ e un test $f : \mathcal{X}_i \to \{1, 2\}$
   (c) Associa il test $f$ a $v$ e partiziona $S_\ell$ in due sottoinsiemi:

$$S_{\ell'} = \{(x_t, y_t) \in S_\ell : f(x_{t,i}) = 1\} \qquad \text{e} \qquad S_{\ell''} = \{(x_t, y_t) \in S_\ell : f(x_{t,i}) = 2\}$$

   (d) Associa a $\ell'$ l'etichetta più frequente in $S_{\ell'}$
   (e) Associa a $\ell''$ l'etichetta più frequente in $S_{\ell''}$

## 3.3 *Overfitting*

**Se il numero di nodi dell'albero è troppo alto rispetto alla cardinalità di $S$ si potrà avere dell'*overfitting*. Per questo motivo, la scelta della foglia da espandere dovrebbe garantire approssimativamente la riduzione massima del *training error***.

Nella pratica, per calcolare il *training error* si usano funzioni diverse da $\psi(p) = \min\{p, 1-p\}$. Questo perchè $\psi$ può essere problematica in alcuni casi. Un esempio è il seguente:

$$p = \frac{N_\ell^+}{N_\ell} = 0.8 \qquad q = \frac{N_{\ell'}^+}{N_{\ell'}} = 0.6 \qquad r = \frac{N_{\ell''}^+}{N_{\ell''}} = 1 \qquad \alpha = \frac{N_{\ell'}^+}{N_\ell} = 0.5$$

$$\underbrace{\psi(p)}_{\text{apporto di } \ell} - \overbrace{(\alpha\psi(q)}^{\text{apporto di } \ell'} + \underbrace{(1-\alpha)\psi(r))}_{\text{apporto di } \ell''} = 0.2 - (0.5 \cdot 0.4 + 0.5 \cdot 0) = 0$$

In questo passaggio il cambiamento che si avrebbe rimpiazzando la foglia $\ell$ sarebbe nullo e quindi non verrebbe scelto dall'algoritmo. Potrebbe inoltre succedere che tutte le foglie diano questo risultato facendo quindi bloccare l'algoritmo.

Per correggere questo problema vengono usate altre funzioni $\psi$. Queste funzioni sono simili a quella già vista in quanto simmetriche attorno a $1/2$ e nulle agli estremi ($\psi(0) = \psi(1) = 0$). Alcune funzioni usate sono:

- **Funzione di Gini:** $\psi_2(p) = 2p(1-p)$
- **Entropia scalata:** $\psi_3(p) = -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p)$
- $\psi_4(p) = \sqrt{p(1-p)}$

Come si può vedere in figura 8, valgono le seguenti disuguaglianze ($\psi_1(p) = \min\{p, 1-p\}$):

$$\psi_1(p) \leq \psi_2(p) \leq \psi_3(p) \leq \psi_4(p)$$

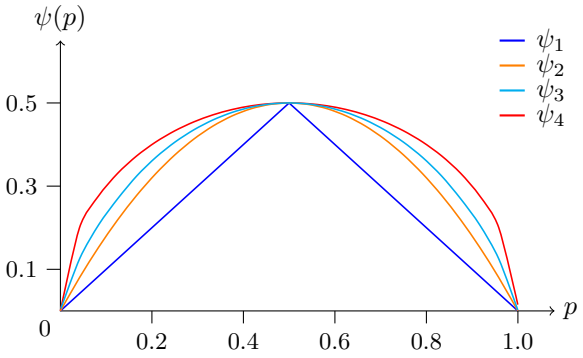



Figura 8: Grafici delle funzioni $\psi$

## 3.4 Interpretabilità

Una proprietà interessante dei *tree predictors* per la classificazione binaria è che possono essere rappresentati con una proposizione logica in forma normale disgiuntiva (DNF). Questa rappresentazione è ottenuta considerando le clausule (congiunzione di predicati) che risultano dai test che si trovano sui percorsi che portano ad un etichetta $+1$.

Un esempio è la seguente DNF ricavata dall'albero in figura 9:

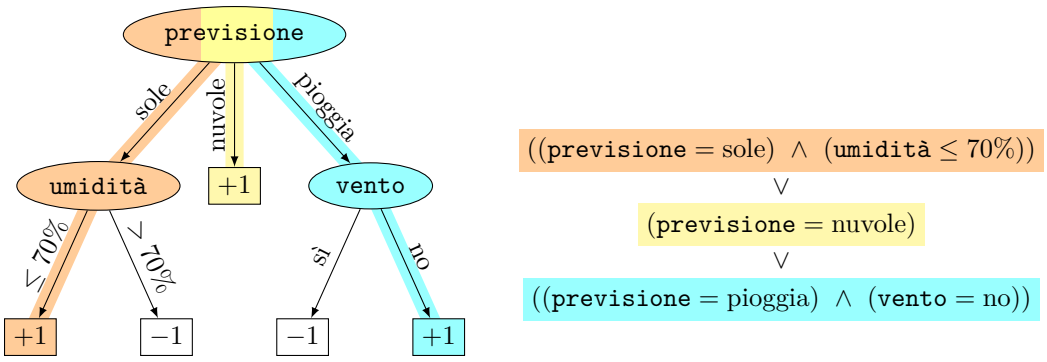



Figura 9: *Tree predictor*

Questa rappresentazione "logica" dell'albero è molto intuitiva e permette di essere manipolata attraverso le regole della logica preposizionale, permettendo quindi una possibile semplificazione del classificatore. Soprattutto, questa rappresentazione fornisce una descrizione interpretabile della conoscenza del *learning algorithm* estratta dal *training set*.

# 4 *Statistical Learning*

Per poter analizzare un *learning algorithm* c'è bisogno di definire un modello matematico di come gli esempi $(x, y)$ siano generati. Nel contesto della *statistical learning* si assumerà che ogni esempio sia ottenuto attraverso un'estrazione indipendente da una distribuzione di probabilità fissata su $\mathcal{X} \times \mathcal{Y}$. Si scriverà $(X, Y)$ per sottolineare come **le due componenti di un esempio siano due variabili aleatorie**.

Assumere che ogni esempio $(x, y)$ sia la realizzazione di un'estrazione casuale **indipendente** da un'unica distribuzione $\mathcal{D}$, implica che ogni *dataset* (come *test* e *training set*) sia un campione statistico. L'indipendenza dei dati è in realtà violata in alcuni domini pratici. Nonostante ciò, l'assunzione di indipendenza nei dati è estremamente utile dal punto di vista della tracciabilità analitica del problema e funziona sorpendentemente bene nella pratica.

## 4.1 Definizioni

Nel contesto della *statistical learning* un problema è specificato da una coppia $(\mathcal{D}, \ell)$, dove $\mathcal{D}$ è la distribuzione e $\ell$ la *loss function*.

### 4.1.1 Rischio statistico $\ell_{\mathcal{D}}$

Le prestazioni di un predittore $h : \mathcal{X} \to \mathcal{Y}$ rispetto a $(\mathcal{D}, \ell)$ è valutata dal **rischio statistico**:

$$\ell_{\mathcal{D}}(h) = \mathbb{E}[\ell(Y, h(X))]$$

che indica il valore atteso della *loss function* su un esempio casuale $(X, Y)$ estratto da $\mathcal{D}$.

### 4.1.2 Predittore ottimo di Bayes $f^*$

Data $\mathcal{D}$, il miglior predittore possibile $f^* : \mathcal{X} \to \mathcal{Y}$ è detto **predittore ottimo di Bayes**:

$$f^*(x) = \operatorname*{argmin}_{\hat{y} \in \mathcal{Y}} \mathbb{E}[\ell(Y, \hat{y}) \mid X = x]$$

### 4.1.3 Rischio condizionato

L'argomento di argmin di $f^*$, ovvero $\mathbb{E}[\ell(Y, \hat{y}) \mid X = x]$, è detto **rischio condizionato**. Il *Bayes optimal predictor* quindi, è la predizione che minimizza il rischio condizionato. Un altro modo per scrivere il rischio condizionato di un predittore $f$ è:

$$\mathbb{E}[\ell(Y, \hat{y}) \mid X = x] = \mathbb{E}[\ell(Y, f(X)) \mid X = x]$$

### 4.1.4 Rischio di Bayes $\ell_{\mathcal{D}}(f^*)$

Essendo $f^*$ il miglior predittore possibile per $\mathcal{D}$ è ragionevole pensare che esso abbia il rischio statistico migliore, ed è così; si ha infatti che il rischio $\ell_{\mathcal{D}}(f^*)$, detto **rischio di Bayes**, è il minore tra tutti i predittori:

$$\forall h \in \mathcal{F} \quad \ell_{\mathcal{D}}(f^*) \leq \ell_{\mathcal{D}}(h)$$

Tipicamente il rischio di Bayes è maggiore di zero vista la casualità delle etichette.

## 4.2 $f^*$ e $\ell_{\mathcal{D}}$ nelle varie *loss function*

Si valuteranno ora i predittori ottimi di Bayes per le varie *loss function*.

### 4.2.1 *Quadratic loss*

Sia $\ell(y, \hat{y}) = (y - \hat{y})^2$ con $\mathcal{Y} = \mathbb{R}$.

Si analizzi il predittore ottimo di Bayes:

$$\begin{aligned} f^*(x) &= \operatorname*{argmin}_{\hat{y} \in \mathbb{R}} \mathbb{E}[(Y - \hat{y})^2 \mid X = x] \\ &= \operatorname*{argmin}_{\hat{y} \in \mathbb{R}} \mathbb{E}[Y^2 + \hat{y}^2 - 2\hat{y}Y \mid X = x] \end{aligned}$$

Per le varie proprietà del valore atteso si ha:

$$= \operatorname*{argmin}_{\hat{y} \in \mathbb{R}} \big(\mathbb{E}[Y^2 \mid X = x] + \mathbb{E}[\hat{y}^2 \mid X = x] - \mathbb{E}[2\hat{y}Y \mid X = x]\big)$$

$$= \operatorname*{argmin}_{\hat{y} \in \mathbb{R}} \big(\mathbb{E}[Y^2 \mid X = x] + \mathbb{E}[\hat{y}^2 \mid X = x] - 2\hat{y}\mathbb{E}[Y \mid X = x]\big)$$

Siccome argmin varia su $\hat{y}$, tutti i fattori che non ne dipendono non incidono sul risultato; possono quindi essere tolti:

$$= \operatorname*{argmin}_{\hat{y} \in \mathbb{R}} \big(\mathbb{E}[\hat{y}^2 \mid X = x] - 2\hat{y}\mathbb{E}[Y \mid X = x]\big)$$

Non essendo $\hat{y}$ una variabile aleatoria:

$$= \operatorname*{argmin}_{\hat{y} \in \mathbb{R}} \big(\hat{y}^2 - 2\hat{y}\mathbb{E}[Y \mid X = x]\big)$$

L'argomento di argmin è una funzione del tipo:

$$F(\hat{y}) = \hat{y}^2 - 2\hat{y}q \qquad\qquad q = \mathbb{E}[Y \mid X = x]$$

Obiettivo di argmin è trovare il valore di $\hat{y}$ che minimizza $F$. Facendo un semplice studio di funzione si può trovare che $F$ è minimizzata quando:

$$\begin{aligned} &F'(\hat{y}) = 2\hat{y} - 2q \\ &\text{Cerco il minimo:} \\ &F'(\hat{y}) = 0 \qquad \Rightarrow \qquad \hat{y} = \mathbb{E}[Y \mid X = x] \\ &2\hat{y} - 2q = 0 \\ &\hat{y} = q \end{aligned}$$

**Si può quindi dire che**:

$$f^*(x) = \mathbb{E}[Y \mid X = x] = \mathbb{E}[Y \mid X]$$

Sostituendo il risultato appena mostrato nella formula del rischio condizionato si ha:

$$\begin{aligned} &\mathbb{E}[\ell(Y, \hat{y}) \mid X = x] \\ &= \mathbb{E}[(Y - f^*(X))^2 \mid X = x] \\ &= \mathbb{E}[(Y - \mathbb{E}[Y \mid X])^2 \mid X = x] \\ &= \operatorname{Var}[Y \mid X] \end{aligned}$$

**Il rischio di Bayes sarà quindi**:

$$\ell_{\mathcal{D}}(f^*) = \mathbb{E}[\operatorname{Var}[Y \mid X]] \neq \operatorname{Var}[Y]$$

Da notare che il valore atteso della varianza dato $X$ sia diverso dalla varianza; per la legge della varianza totale si ha infatti che:

$$\operatorname{Var}[Y] - \mathbb{E}[\operatorname{Var}[Y \mid X]] = \operatorname{Var}[\mathbb{E}[Y \mid X]]$$

### 4.2.2 *Zero-one loss*

Si guarderà ora la classificazione binaria dove $\mathcal{Y} = \{-1, 1\}$.

Si definiscano due funzioni:

$$\eta(x) = \mathbb{P}(Y = 1 \mid X = x) \qquad (\mathbb{P} \text{ è la funzione di probabilità})$$

$$\mathbb{I}\{A\} = \begin{cases} 1 & \text{si verifica } A \\ 0 & \text{altrimenti} \end{cases} \qquad (A \text{ è un evento})$$

La funziona *zero-one loss* si potrà ora definire:

$$\ell(y, \hat{y}) = \mathbb{I}\{\hat{y} \neq y\}$$

**Si analizzi il rischio statistico**:

$$\begin{aligned} \ell_{\mathcal{D}}(h) &= \mathbb{E}[\ell(Y, h(X))] \\ &= \mathbb{E}[\mathbb{I}\{h(X) \neq Y\}] \\ &= \mathbb{P}(h(X) \neq Y) \end{aligned}$$

Di conseguenza **il predittore ottimo di Bayes è**:

$$\begin{aligned} f^*(x) &= \operatorname*{argmin}_{\hat{y} \in \{-1,1\}} \mathbb{E}[(Y - \hat{y})^2 \mid X = x] \\ &= \operatorname*{argmin}_{\hat{y} \in \{-1,1\}} \mathbb{E}[\mathbb{I}\{Y = 1\}\mathbb{I}\{\hat{y} = -1\} + \mathbb{I}\{Y = -1\}\mathbb{I}\{\hat{y} = 1\} \mid X = x] \end{aligned}$$

Si applichi la definizione di valore atteso:

$$\begin{aligned} &= \operatorname*{argmin}_{\hat{y} \in \{-1,1\}} \Big( \mathbb{P}(Y = 1 \mid X = x)\mathbb{I}\{\hat{y} = -1\} + \mathbb{P}(Y = -1 \mid X = x)\mathbb{I}\{\hat{y} = 1\} \Big) \\ &= \operatorname*{argmin}_{\hat{y} \in \{-1,1\}} \Big( \eta(x)\mathbb{I}\{\hat{y} = -1\} + (1 - \eta(x))\mathbb{I}\{\hat{y} = 1\} \Big) \\ &= \begin{cases} -1 & \eta(x) < 1/2 \\ +1 & \eta(x) \geq 1/2 \end{cases} \end{aligned}$$

$f^*(x)$ predirrà quindi l'etichetta con la maggior probabilità quando condizionata dall'istanza.

Infine è facile verificare che **il rischio di Bayes è**:

$$\ell_{\mathcal{D}}(f^*) = \mathbb{E}[\min\{\eta(X), 1 - \eta(X)\}]$$

## 4.3 Limitare il rischio

Si vedrà ora come limitare il rischio statistico di un predittore.

### 4.3.1 Stima del rischio

Dato un predittore $h$ **non si può calcolare direttamente il suo rischio statistico** $\ell_{\mathcal{D}}(h)$; il motivo è semplice: **non si conosce** $\mathcal{D}$ (se si conoscesse si potrebbe calcolare direttamente il predittore ottimo di Bayes).

Si dovrà quindi procedere ad ottenere una stima del rischio di un predittore $h$. Per farlo **si userà il *test set*** $S' = \{(x'_1, y'_1), \ldots, (x'_n, y'_n)\}$; in particolare si assumerà che $S'$ sia stato generato attraverso estrazioni indipendenti da $\mathcal{D}$ e che quindi:

$$(X'_t, Y'_t) \sim \mathcal{D} \qquad t = 1, \ldots, n$$

Questo permette di affermare che:

$$\mathbb{E}[\ell(Y', h(X'))] = \ell_{\mathcal{D}}(h)$$

Infine, usando come stimatore la media campionaria del rischio, **si può avere una stima del rischio** attraverso il *test set*, **ottenendo di fatto il *test error***:

$$\ell_{S'}(h) = \frac{1}{n}\sum_{t=1}^{n} \ell(y'_t, h(x'_t))$$

È importante sottolineare che **la stima ha senso solo se $h$ non dipende in alcun modo dal *test set***.

### 4.3.2 Chernoff-Hoeffding

Per poter calcolare quanto "buona" è la stima del rischio si può usare il seguente risultato della legge dei grandi numeri.

**Lemma 1** (Chernoff-Hoeffding)**.** *Siano $Z_1, \ldots, Z_n$ delle variabili aleatorie indipendenti e identicamente distribuite; sia $\mu$ il valore atteso e sia $0 \leq Z_t \leq 1$ per ogni $t = 1, \ldots, n$. Allora per ogni $\varepsilon > 0$ si ha che:*

$$\mathbb{P}\left(\frac{1}{n}\sum_{t=1}^{n} Z_t > \mu + \varepsilon\right) \leq e^{-2\varepsilon^2 n} \qquad e \qquad \mathbb{P}\left(\frac{1}{n}\sum_{t=1}^{n} Z_t < \mu - \varepsilon\right) \leq e^{-2\varepsilon^2 n}$$

Applicando il precedente lemma con $Z_t = \ell(Y'_t, h(X'_t)) \in [0,1]$ si può trovare un intervallo di confidenza del rischio statistico:

$$\begin{aligned}
\mathbb{P}\Big(|\ell_{S'}(h) - \ell_{\mathcal{D}}(h)| > \varepsilon\Big) &= \mathbb{P}\Big((\ell_{S'}(h) > \ell_{\mathcal{D}}(h) + \varepsilon) \ \cup \ (\ell_{S'}(h) < \ell_{\mathcal{D}}(h) - \varepsilon)\Big) \\
&= \underbrace{\mathbb{P}\Big(\ell_{S'}(h) > \ell_{\mathcal{D}}(h) + \varepsilon\Big)}_{\leq e^{-2\varepsilon^2 n}} + \underbrace{\mathbb{P}\Big(\ell_{S'}(h) < \ell_{\mathcal{D}}(h) - \varepsilon\Big)}_{\leq e^{-2\varepsilon^2 n}} \\
&\Downarrow \\
\mathbb{P}\Big(|\ell_{S'}(h) - \ell_{\mathcal{D}}(h)| > \varepsilon\Big) &\leq \ 2e^{-2\varepsilon^2 n} \qquad (6)
\end{aligned}$$

Questo mostra che **la probabilità che il *test error* $\ell_{S'}(h)$ differisca dal rischio statistico $\ell_{\mathcal{D}}(h)$ per più di $\varepsilon$, diminuisce esponenzialmente con il crescere della dimensione $n$ del *test set***.

Andando più nello specifico, si prenda un valore $\delta \in (0,1)$, uguale a $2e^{-2\varepsilon^2 n}$, e si isoli la $\varepsilon$:

$$\begin{aligned}
2e^{-2\varepsilon^2 n} &= \delta \\
e^{2\varepsilon^2 n} &= \frac{2}{\delta} \\
2\varepsilon^2 n &= \ln\frac{2}{\delta} \\
\varepsilon^2 &= \frac{1}{2n}\ln\frac{2}{\delta} \\
\varepsilon &= \sqrt{\frac{1}{2n}\ln\frac{2}{\delta}}
\end{aligned}$$

Applicando i risultati appena mostrati su (6) si ottiene infine:

$$\mathbb{P}\left(|\ell_{\mathcal{D}}(h) - \ell_{S'}(h)| \leq \sqrt{\frac{1}{2n}\ln\frac{2}{\delta}}\right) \geq 1 - \delta$$

## 4.4 Decomposizione *bias-variance*

Si consideri un problema di *learning* $(\mathcal{D}, \ell)$ e un *learning algorithm* $A$. Si scriverà $A(S)$ per indicare il predittore generato da $A$ preso in input il *training set* $S$.

Sia $\mathcal{H}_m = \{h_1, \ldots, h_m\}$ l'insieme dei predittori generati da $A$ su *training set* di dimensione $m$; un predittore $h$ appartiene a $\mathcal{H}_m$ se e solo se esiste un *training set* $S$ di dimensione $m$ tale che $A(S) = h$:

$$h \in \mathcal{H}_m \;\Leftrightarrow\; \exists S : |S| = m \;\wedge\; A(S) = h$$

Sia $h^*$ il miglior predittore di $\mathcal{H}_m$, ovvero il predittore con il rischio minimo (ricordando che potrebbero essere più di uno):

$$h^* \in \operatorname*{argmin}_{h \in \mathcal{H}_m} \ell_{\mathcal{D}}(h)$$

Fissato un *training set* $S$ di dimensione $m$ si indichi $h_S = A(S) \in \mathcal{H}_m$. Di seguito verrà mostrata la decomposizione *bias-variance*:

$$\begin{aligned} \ell_{\mathcal{D}}(h_S) = \quad & \ell_{\mathcal{D}}(h_S) - \ell_{\mathcal{D}}(h^*) && \text{(errore di stima o di variazione)} \\ & + \ell_{\mathcal{D}}(h^*) - \ell_{\mathcal{D}}(f^*) && \text{(errore di approssimazione o di bias)} \\ & + \ell_{\mathcal{D}}(f^*) && \text{(rischio di Bayes)} \end{aligned}$$

Dove $f^*$ è il predittore ottimo di Bayes.

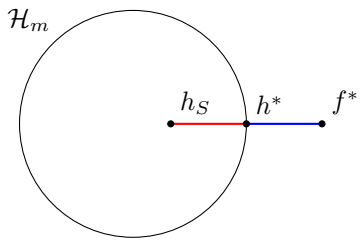



Figura 10: Rappresentazione grafica della decomposizione *bias-variance*

Si noti che:

- Il rischio di Bayes non dipende da $A$;
- L'errore di approssimazione è grande quando $\mathcal{H}_m$ non contiene una buona approssimazione di $f^*$;
- L'errore di stima è grande quando $S$ non contiene abbastanza informazioni necessarie ad identificare $h^*$.

## 4.5 *Overfitting* e *underfitting*

Obiettivo di questo paragrafo è stabilire una connessione tra la decomposizione *bias-variance* e *overfitting/underfitting*. Per farlo si prenda come $A$ il *learning algorithm* ERM su una classe di predittori arbitraria $\mathcal{H}$ :

$$A(S) = h_S = \operatorname*{argmin}_{h \in \mathcal{H}} \ell_S(h)$$

Si tenga poi conto del predittore ottimo $h^*$ di $\mathcal{H}$:

$$h^* \in \operatorname*{argmin}_{h \in \mathcal{H}} \ell_{\mathcal{D}}(h)$$

Si noti che $h_S$ è il miglior predittore che ERM riesce a trovare basando la stima del rischio su $S$, mentre $h^*$ è il miglior predittore possibile esistente in $\mathcal{H}$, **che non dipende quindi da** $S$.

Il lemma 1 non può essere applicato direttamente in quanto $\ell_S(h_S)$ e $\ell_D(h_S)$ sono entrambi variabili aleatorie il cui valore atteso non coincide necessariamente. Per poter quindi analizzare formalmente l'errore di variazione di ERM si proceda come segue.

Per ogni *training set* $S$ di dimensione $m$ si ha che:

$$\ell_D(h_S) - \ell_D(h^*) = \ \ell_D(h_S) - \ell_S(h_S) \ + \ \ell_S(h_S) - \ell_D(h^*)$$

Per definizione $h_S$ minimizza il rischio stimato su $S$, si ha quindi che $\ell_S(h_S) \leq \ell_S(h^*)$:

$$\begin{aligned} &\leq \ \ell_D(h_S) - \ell_S(h_S) \ + \ \ell_S(h^*) - \ell_D(h^*) \\ &\leq |\ell_D(h_S) - \ell_S(h_S)| + |\ell_S(h^*) - \ell_D(h^*)| \\ &\leq 2 \max_{h \in \mathcal{H}} |\ell_S(h) - \ell_D(h)| \end{aligned} \tag{7}$$

Concentrandosi ora sull'errore di variazione, si vuole capire quando, per ogni $\varepsilon > 0$ si ha:

$$\ell_D(h_S) - \ell_D(h^*) > \varepsilon \tag{8}$$

Unendo la disuguaglianza appena vista con (7) si ha:

$$2 \max_{h \in \mathcal{H}} |\ell_S(h) - \ell_D(h)| \geq \ell_D(h_S) - \ell_D(h^*) > \varepsilon$$

Da cui:

$$2 \max_{h \in \mathcal{H}} |\ell_S(h) - \ell_D(h)| > \varepsilon$$

$$\max_{h \in \mathcal{H}} |\ell_S(h) - \ell_D(h)| > \frac{\varepsilon}{2} \tag{9}$$

$$\Downarrow$$

$$\exists h \in \mathcal{H} : |\ell_S(h) - \ell_D(h)| > \frac{\varepsilon}{2} \tag{10}$$

Dai risultati (8) e (10) si evince che:

$$\begin{aligned} (8) \quad &\Rightarrow \quad (10) \\ \ell_D(h_S) - \ell_D(h^*) > \varepsilon \quad &\Rightarrow \quad \exists h \in \mathcal{H} : |\ell_S(h) - \ell_D(h)| > \frac{\varepsilon}{2} \end{aligned} \tag{11}$$

La figura 11 mostra come:

$$A \Rightarrow B \ \rightarrow \ \mathbb{P}(A) \leq \mathbb{P}(B) \tag{12}$$

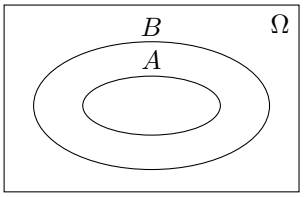



Figura 11

Da (11) e (12) si ottiene che:

$$\mathbb{P}(\ell_D(h_S) - \ell_D(h^*) > \varepsilon) \leq \mathbb{P}\left(\exists h \in \mathcal{H} : |\ell_S(h) - \ell_D(h)| > \frac{\varepsilon}{2}\right) \tag{13}$$

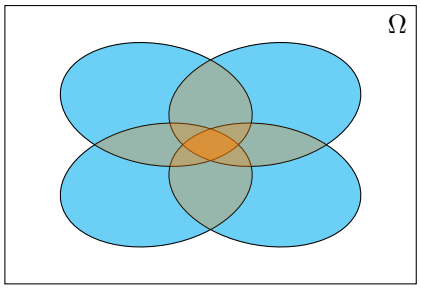



Figura 12

**Si guardi solo il caso in cui $\mathcal{H}$ è finito ($|\mathcal{H}| < \infty$).**

$$\mathbb{P}\left(\exists h \in \mathcal{H} : |\ell_S(h) - \ell_D(h)| > \frac{\varepsilon}{2}\right) = \mathbb{P}\left(\bigcup_{h\in\mathcal{H}} \left(|\ell_S(h) - \ell_D(h)| > \frac{\varepsilon}{2}\right)\right)$$

La probabilità dell'unione di più eventi non è uguale alla somma delle probabilità dei singoli eventi in quanto alcuni eventi si potrebbero sovrapporre come mostrato in figura 12. Tuttavia rappresenta un limite superiore:

$$\leq \sum_{h\in\mathcal{H}} \mathbb{P}\left(|\ell_S(h) - \ell_D(h)| > \frac{\varepsilon}{2}\right)$$

Si applichi il lemma 1:

$$\leq \sum_{h\in\mathcal{H}} 2e^{-2\frac{\varepsilon^2}{4}m}$$

$$\leq \sum_{h\in\mathcal{H}} 2e^{-\frac{\varepsilon^2 m}{2}}$$

$$\leq |\mathcal{H}|2e^{-\frac{\varepsilon^2 m}{2}} \tag{14}$$

Da (13) e (14) si ha che:

$$\mathbb{P}(\ell_D(h_S) - \ell_D(h^*) > \varepsilon) \leq |\mathcal{H}|2e^{-m\varepsilon^2/2} \tag{15}$$

Si chiami il lato destro $\delta$ e si isoli la $\varepsilon$:

$$2|\mathcal{H}|e^{-m\varepsilon^2/2} = \delta \tag{16}$$

$$e^{-m\varepsilon^2/2} = \frac{\delta}{2|\mathcal{H}|}$$

$$e^{m\varepsilon^2/2} = \frac{2|\mathcal{H}|}{\delta}$$

$$\frac{m\varepsilon^2}{2} = \ln\frac{2|\mathcal{H}|}{\delta}$$

$$\varepsilon^2 = \frac{2}{m}\ln\frac{2|\mathcal{H}|}{\delta}$$

$$\varepsilon = \sqrt{\frac{2}{m}\ln\frac{2|\mathcal{H}|}{\delta}} \tag{17}$$

Concludendo, da (15) si ha che:

$$\mathbb{P}(\ell_D(h_S) \leq \ell_D(h^*) + \varepsilon) \geq 1 - |\mathcal{H}| 2e^{-m\varepsilon^2/2}$$

Applicando (16) e (17) si ottiene:

$$\mathbb{P}\left(\ell_D(h_S) \leq \ell_D(h^*) + \sqrt{\frac{2}{m} \ln \frac{2|\mathcal{H}|}{\delta}}\right) \geq 1 - \delta$$

Si ricordi che $m$ è la dimensione del *training set*.

Fissato $m$, per poter diminuire lo scarto $\sqrt{\frac{2}{m} \ln \frac{2|\mathcal{H}|}{\delta}}$ dell'errore di variazione di ERM, **bisogna diminuire** $|\mathcal{H}|$. Diminuendo $|\mathcal{H}|$ però si incrementerebbe $\ell_D(h^*)$, il che farebbe aumentare l'errore di bias.

Alla luce di quest'analisi statistica, si può concludere che gli algoritmi ERM generano predittori con un alto rischio (rispetto al rischio di Bayes) quando c'è uno sbilanciamento tra l'errore di variazione e quello di bias. In particolare:

- Si verifica *overfitting* quando l'errore di variazione domina l'errore di bias;
- Si verifica *underfitting* quando l'errore di bias domina l'errore di variazione;

# 5 Analisi del rischio sui *tree predictor*

L'analisi del rischio per ERM su una classe finita $\mathcal{H}$ di predittori dice che, con probabilità di almeno $1-\delta$ rispetto all'estrazione casuale di un *training set* di dimensione $m$, si ha:

$$\ell_{\mathcal{D}}(h_S) \leq \ell_{\mathcal{D}}(h^*) + \sqrt{\frac{2}{m} \ln \frac{2|\mathcal{H}|}{\delta}}$$

Si proverà in questa sezione ad applicare questo risultato alla classe dei predittori calcolati da *tree classifier* binari su $\mathcal{X} = \{0,1\}^d$. **Si considereranno solo alberi binari completi ovvero alberi i cui nodi hanno o zero o due figli**. Un albero completo in cui le foglie si trovano tutte alla stessa profondità si dice albero binario perfetto. Un albero binario completo con $N$ nodi ha sempre $(N+1)/2$ foglie.

**Osservazione 1.** *Per ogni funzione $h : \{0,1\}^d \to \{-1,1\}$ esiste un tree classifier binario con almeno $2^{d+1}-1$ nodi che calcolano $h$.*

*Dimostrazione.* Si prenda l'albero binario perfetto che calcoli il predittore $h : \{0,1\}^d \to \{-1,1\}$. Per ogni possibile $x \in \{0,1\}^d$ ci dovrà essere una corrispettiva etichetta. Ci sono $2^d$ possibili punti. Di conseguenza l'albero avrà $2^d$ foglie. Essendo l'albero completo ed avendo $2^d$ foglie avrà $2^{d+1}-1$ nodi. □

Siccome ci sono $2^{2^d}$ funzioni binarie su $\{0,1\}^d$, si può eseguire ERM con una classe $\mathcal{H}$ contenente $2^{2^d}$ *tree classifier*. Contenendo tutte le possibili funzioni, si avrà che $f^* \in \mathcal{H}$ e quindi verrà scelto da ERM avendo così un errore di bias nullo. Analizzando il limite superiore dell'errore di variazione si ha:

$$\ell_{\mathcal{D}}(h_S) \leq \ell_{\mathcal{D}}(h^*) + \sqrt{\frac{2}{m} \ln \frac{2 \cdot 2^{2^d}}{\delta}}$$

$$\ell_{\mathcal{D}}(h_S) \leq \ell_{\mathcal{D}}(h^*) + \sqrt{\frac{2}{m} \left( \ln 2^{2^d} + \ln \frac{2}{\delta} \right)}$$

$$\ell_{\mathcal{D}}(h_S) \leq \ell_{\mathcal{D}}(h^*) + \sqrt{\frac{2}{m} \left( 2^d \ln 2 + \ln \frac{2}{\delta} \right)}$$

Questo mostra che per controllare l'errore di variazione di ERM, il *training set* deve contenere un numero $m$ di esempi dell'ordine di grandezza di $2^d$. Questo è un caso estremo di *overfitting* creato scegliendo $\mathcal{H}$ talmente grande da azzerare l'errore di bias.

## 5.1 Limitare il numero di nodi

Per cercare di ridurre l'*overfitting*, si può incrementare l'errore di bias riducendo la dimensione di $\mathcal{H}$, guardando quindi una classe di alberi ridotta.

Si consideri l'insieme $\mathcal{H}_N$ di tutti i classificatori calcolati da *tree predictor* binari completi con esattamente $N$ nodi, dove $N \ll 2^d$.

**Osservazione 2.** $|\mathcal{H}_N| \leq (2de)^N$

Dall'osservazione 2 si trova che:

$$\ell_{\mathcal{D}}(h_S) \leq \ell_{\mathcal{D}}(h^*) + \sqrt{\frac{2}{m} \left( N(1 + \ln(2d)) + \ln \frac{2}{\delta} \right)}$$

Da qui si può dedurre che un *training set* con delle dimensioni dell'ordine di $N \ln d$ è abbastanza a controllare il rischio dei predittori $h \in \mathcal{H}$.

## 5.2 Limite superiore più dettagliato

### 5.2.1 *Structural Risk Minimization* (SRM)

Dalle conclusioni fatte fino ad ora non si capisce bene che valore $N$ debba assumere nella pratica; si vedrà quindi un limite dell'errore più dettagliato.

Precedentemente si è controllato l'errore di variazione di ERM controllando quando il rischio di ogni predittore possa eccedere il suo *training error* di almeno $\varepsilon$. **Si vedrà ora un approccio differente: si limitera superiormente il rischio di un *tree predictor* $h$ dal suo *training error* più una quantità $\varepsilon_h$ che dipende dalla dimensione dell'albero**.

Si introdurrà una funzione $w : \mathcal{H} \to [0, 1]$ e sia $w(h)$ il peso del *tree predictor* $h$. Si assuma che:

$$\sum_{h \in \mathcal{H}} w(h) \leq 1 \tag{18}$$

Si può quindi scrivere che:

$$\mathbb{P}(\exists h \in \mathcal{H} : |\ell_S(h) - \ell_D(h)| > \varepsilon_h) \leq \sum_{h \in \mathcal{H}} \mathbb{P}(|\ell_S(h) - \ell_D(h)| > \varepsilon_h)$$

Si applichi il lemma 1 (Chernoff-Hoeffding):

$$\leq \sum_{h \in \mathcal{H}} 2e^{-2m\varepsilon_h^2}$$

Scegliendo:

$$\varepsilon_h = \sqrt{\frac{1}{2m}\left(\ln \frac{1}{w(h)} + \ln \frac{2}{\delta}\right)}$$

Si ottiene che:

$$\mathbb{P}(\exists h \in \mathcal{H} : |\ell_S(h) - \ell_D(h)| > \varepsilon_h) \leq \sum_{h \in \mathcal{H}} \delta w(h) \leq \delta$$

Dove nell'ultimo passaggio si è usata l'assunzione (18). Una coseguenza di ciò è che per ogni *tree predictor* $h \in \mathcal{H}$ si ha che:

$$\mathbb{P}\left(\ell_D(h) \leq \ell_S(h) + \sqrt{\frac{1}{2m}\left(\ln \frac{1}{w(h)} + \ln \frac{2}{\delta}\right)}\right) \geq 1 - \delta$$

Questo suggerisce un algoritmo alternativo a ERM per la minimizzazione del *training error*: **la *Structural Risk Minimization* (SRM):**

$$h_S = \operatorname*{argmin}_{h \in \mathcal{H}} \left(\ell_S(h) + \sqrt{\frac{1}{2m}\left(\ln \frac{1}{w(h)} + \ln \frac{2}{\delta}\right)}\right)$$

La funzione $w$ può essere vista come una misura della complessità del *tree predictor* $h$. Più il predittore è complesso più c'è il rischio che avvenga *overfitting*. **SRM punta ad ottenere un predittore che minimizza sia il *training error* che la complessità del predittore**.

### 5.2.2 Funzione peso $w$

Una possibile funzione $w$ si può ottenere come segue: utilizzando tecniche teoriche di codifica, si può codificare ogni *tree predictor* $h$ con $N_h$ nodi, utilizzando una stringa binaria $\sigma(h)$. La lunghezza della stringa sarà:

$$|\sigma(h)| = (N_h + 1)\lceil \log_2 (d + 3) \rceil + 2\lfloor \log_2 N_h \rfloor + 1 = O(N_h \log d)$$

La codifica deve garantire che non esistano due *tree predictor* tali che la codifica di uno sia prefisso della codifica dell'altro. Questo tipo di codifica è detta istantanea è garantisce che:

$$\sum_{h \in \mathcal{H}} 2^{-|\sigma(h)|} \leq 1$$

**Si può quindi assegnare alla funzione peso:**

$$w(h) = 2^{-|\sigma(h)|}$$

Si può quindi introdurre un *learning algorithm* per *tree predictor* che controlla l'*overfitting* generando predittori così definiti:

$$h_S = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \left( \ell_S(h) + \sqrt{\frac{1}{2m} \left( |\sigma(h)| + \ln \frac{2}{\delta} \right)} \right) \qquad (\text{con } |\sigma(h)| = O(N_h \log d))$$

Si noti come la scelta della funzione peso $w$ non sia determinata dall'analisi fatta. Si potrebbe scegliere qualsiasi altra funzione $w$ che soddisfi (18). La funzione $w$ andrebbe interpretata come un termine di bias, preferendo alcuni alberi rispetto ad altri. Un bias verso alberi più piccoli è un esempio del principio Occam Razor: se due spiegazioni concordano con una serie di osservazioni, la spiegazione più breve è quella con la migliore "potenza predittiva".

# 6 Iperparametri e stima del rischio

I *learning algorithm* spesso hanno degli iperparametri; un'esempio è il parametro $k$ degli algoritmi $k-$NN. Il loro valore deve essere determinato prima della fase di *training*. **I valori degli iperparametri sono di cruciale importanza per il controllo dell'*underfitting* e dell'*underfitting***.

Un *learning algorithm* con uno o più iperparametri non è un algoritmo, ma piuttosto una famiglia di algoritmi, uno per ogni possibile valore degli iperparametri. Sia $\{A_\theta : \theta \in \Theta\}$ una famiglia di *learning algorithm*, dove $\Theta$ è l'insieme di tutti i possibili valori degli iperparametri.

Si fissi un *learning problem* $(\mathcal{D}, \ell)$ e sia $A_\theta(S)$ il predittore generato da $A_\theta$ sul *training set* $S$.

## 6.1 Valutazione tramite *cross-validation* esterna

Si fissi l'iperparametro $\theta$; **si vogliono stimare le performance di $A_\theta$ stimando** $\mathbb{E}[\ell_{\mathcal{D}}(A_\theta(S))]$. Per farlo si userà una tecnica chiamata $K$-fold *cross-validation* (esterna).

Sia $S$ l'intero *data set*. Si partizioni $S$ in $K$ sottoinsiemi (*folds*) $S_1, \ldots, S_K$ ognuno di dimensione $m/K$ (assumendo per semplicità che $K$ divida $m$).

Sia $S_{-i} = S \setminus S_i$. Si chiamerà $S_i$ la **parte di *testing*** dell'$i$-esima *fold* mentre $S_{-i}$ la **parte di *training***.

Per esempio, si partizioni $S = \{(x_1, y_1), \ldots, (x_{20}, y_{20})\}$ in $K = 4$ sottoinsiemi:

$$\begin{aligned} S_1 &= \{(x_1, y_1), \ldots, (x_5, y_5)\} \\ S_2 &= \{(x_6, y_6), \ldots, (x_{10}, y_{10})\} \\ S_3 &= \{(x_{11}, y_{11}), \ldots, (x_{15}, y_{15})\} \\ S_4 &= \{(x_{16}, y_{16}), \ldots, (x_{20}, y_{20})\} \\ S_{-2} &= S_1 \cup S_3 \cup S_4 \end{aligned}$$

La stima di $A$ su $S$ ottenuta tramite $K$-fold $CV$, detta $\ell_S^{\text{cv}}(A)$, viene calcolata come segue:

- Si esegue $A$ su ogni $S_{-i}$ con $i = 1, \ldots, K$, ottenendo dei predittori $h_1, \ldots, h_K$;
- Si calcola l'errore medio sulla parte di *testing* di ogni *fold* ottenendo $\ell_{S_1}(h_1), \ldots, \ell_{S_K}(h_K)$ dove:
$$\ell_{S_i}(h_i) = \frac{K}{m} \sum_{(x,y) \in S_i} \ell(y, h_i(x))$$
- Infine, si calcola la stima facendo una media degli errori ottenuti nel precedente punto:
$$\ell_S^{\text{cv}}(A) = \frac{1}{K} \sum_{i=1}^{K} \ell_{S_i}(h_i)$$

## 6.2 Modifica degli iperparametri

Obiettivo di questa sezione è scegliere gli iperparametri in modo da ottenere un predittore con un rischio basso. Questo è tipicamente fatto minimizzando la stima del rischio sui dati di *training*.

Siccome $\Theta$ può essere molto grande, se non infinito, si cercherà di minimizzare su un sottoinsieme opportuno $\Theta_0 \subset \Theta$. Sia $S$ il *training set*, si vuole trovare $\theta^* \in \Theta_0$ tale che:

$$\ell_{\mathcal{D}}(A_{\theta^*}(S)) = \min_{\theta \in \Theta_0} \ell_D(A_\theta(S)) \tag{19}$$

Questa stima è fatta dividendo i dati di *training* in due sottoinsiemi $S_{\text{train}}$ e $S_{\text{dev}}$. $S_{\text{dev}}$ (detto insieme di validazione) è usato come sostituto del *test set*. L'algoritmo è eseguito su $S_{\text{train}}$ per ogni valore dell'iperparametro $\Theta_0$. I predittori risultanti sono testati su $S_{\text{dev}}$ per trovare il valore $\theta^*$ che minimizza l'errore. Una volta trovato $\theta^*$ si esegue $A_{\theta^*}$ sul *training set* originale, ottenendo il predittore finale.

## 6.3 Modifica dei parametri tramite *nested cross-validation*

Si vuole stimare (19):

$$\mathbb{E}\left[\min_{\theta\in\Theta_0} \ell_{\mathcal{D}}(A_\theta(S))\right] \tag{20}$$

o, in altre parole, si vogliono stimare le performance di $A_\theta$ su un *training set* di dimensione fissata quando $\theta$ è scelto in base al *training set*.

Dato $S$, un modo veloce di stimare (20) è usare il miglior stimatore CV su $\{A_\theta : \theta \in \Theta_0\}$:

$$\min_{\theta\in\Theta_0} \ell_S^{\text{CV}}(A_\theta)$$

Nonostante questa stima tenda a sottostimare (20), in pratica la differenza è tipicamente piccola.

Una miglior stima, seppur computazionalmente più pesante, è ottenuta tramite la *nested* CV:

**Data:** *dataset* $S$
Dividi $S$ in $S_1, \dots, S_K$
**for** $i = 1, \dots, K$ **do**
  $S_{-i} = S \setminus S_i$
  Esegui CV su $S_{-i}$ per ogni $\theta \in \Theta_0$ e trova $\theta_i = \operatorname*{argmin}_{\theta\in\Theta_0} \ell_{S_{-i}}^{\text{CV}}(A_\theta)$
  $h_i = A_{\theta_i}(S_{-i})$
  $\varepsilon_i = \ell_{S_i}(h_i)$
**end**
**Output:** $(\varepsilon_1 + \cdots + \varepsilon_K)/K$

**Algoritmo 1:** $K$-fold *nested cross-validation*

# 7 Consistenza e algoritmi non parametrici

## 7.1 Consistenza

La consistenza è una proprietà asintotica che certifica che il rischio statistico del predittore generato da un *learning algorithm* converga al rischio di Bayes all'aumentare della dimensione del *training set*. Ricordando che $A(S_m)$ è il predittore generato dal *learning algorithm* $A$ sul *training set* $S_m$ di dimensione $m$, **un *learning algorithm* $A$ è consistente rispetto alla funzione di loss $\ell$ se per ogni distribuzione $D$ si ha che:**

$$\lim_{m\to\infty} \mathbb{E}\left[\ell_{\mathcal{D}}(A(S_m))\right] = \ell_{\mathcal{D}}(f^*)$$

In alcuni casi si può definire la consistenza rispetto ad una classe limitata di distribuzioni $\mathcal{D}$. Per esempio, nella classificazione binaria ci si può limitare a tutte le distribuzioni $\mathcal{D}$ tali che la funzione $\eta(x) = \mathbb{P}(Y = 1 \mid X = x)$ sia una funzione di Lipschitz su $\mathcal{X}$. Questo avviene quando:

$$\exists c : 0 < c < \infty \wedge |\eta(x) - \eta(x')| \leq c\|x - x'\| \qquad \forall x, x' \in \mathcal{X}$$

## 7.2 Algoritmi non parametrici

Dato un *learning algorithm* $A$, sia $\mathcal{H}_m$ l'insieme dei predittori generati da $A$ su *training set* di dimensione $m$. $A$ **è non parametrico se il suo errore di approssimazione si annulla al crescere si $m$.** Formalmente:

$$\lim_{m\to\infty} \min_{h\in\mathcal{H}_m} \ell_{\mathcal{D}}(h) = \ell_{\mathcal{D}}(f^*)$$

Gli algoritmi non parametrici si riconoscono per:

- La dimensione in memoria dei loro predittori tende a crescere insieme alla dimensione del *training set*;
- Per valori sufficientemente grandi di $m$, ci sono predittori in $\mathcal{H}_m$ con un *training error* vicino alla zero.

Due esempi di algoritmi non parametrici sono $k-$NN e l'algoritmo greedy per i classificatori ad albero decisionale.

### 7.2.1 $k-$NN

L'algoritmo $k-$NNclassico è non parametrico ma non si conosce se sia consistente per valori fissati di $k$. L'unica cosa che si conosce è che, per ogni distribuzione $\mathcal{D}$ si ha:

$$\lim_{m\to\infty} \mathbb{E}[\ell_{\mathcal{D}}(k\text{-NN}(S_m))] \leq \ell_{\mathcal{D}}(f^*) + 2\sqrt{\frac{\ell_{\mathcal{D}}(f^*)}{k}}$$

Tuttavia, se si sceglie $k$ come una funzione $k_m$ della dimensione del *training set*, allora l'algoritmo diventerebbe consistente a patto che $k_m \to \infty$ e $k_m = o(m)$.

### 7.2.2 Algoritmo greedy per *tree classifier*

Siano:

$$\mathcal{X}_\ell = \{x \in \mathbb{R}^d : x \text{ è indirizzato a } \ell\}$$
$$N_\ell = |\{x \in S : x \text{ è indirizzato a } \ell\}|$$

L'algoritmo greedy per la costruzione di *tree classifier* è consistente quando, per $m \to \infty$, le seguenti due condizioni sono rispettate:

1. Il diametro di $\mathcal{X}_\ell$ va a zero;
2. $N_\ell \to \infty$.

In altre parole, l'albero deve crescere senza limiti ma non troppo velocemente.

## 7.3 Algoritmi non consistenti e non parametrici

Se un algoritmo è non parametrico non è detto sia consistente. Un esempio di algoritmo non parametrico e non consistente è l'algoritmo greedy per *tree classifier* binari che utilizza il rischio statistico come criterio di separazione. In altre parole, invece di sostituire la foglia che provoca il maggior calo del *training error*, si assume che l'algoritmo è in grado di scegliere la foglia che provoca il maggior calo nel rischio per la funzione di *zero-one loss*.

**La consistenza di un algoritmo è sempre preferibile? Nella pratica no.** Può infatti succedere che la velocità di convergenza di un algoritmo consistente sia più bassa di quella di un algoritmo non consistente.

### 7.3.1 Teorema *No Free Lunch*

**Teorema 1** (*No Free Lunch*)**.** *Per qualsiasi sequenza $a_1, a_2, \ldots$ di numeri positivi che converge a 0, che rispetta:*

$$\frac{1}{16} \geq a_1 \geq a_2 \geq \cdots \geq 0$$

*e per ogni learning algorithm $A$ consistente per la classificazione binaria con la zero-one loss, esiste una distribuzione $\mathcal{D}$ tale che:*

$$\forall m \geq 1 \qquad \ell_{\mathcal{D}}(f^*) = 0 \ \wedge \ \mathbb{E}[\ell_{\mathcal{D}}(A(S_m))] \geq a_m$$

Questo teorema non previene che un algoritmo consistente converga rapidamente al rischio di Bayes su una distribuzione $\mathcal{D}$. Quello che il teorema mostra è che se $A$ converge al rischio di Bayes per una qualsiasi distribuzione, allora convergerà in modo arbitrariamente lento per alcune di queste distribuzioni.

## 7.4 Maledizione della dimensionalità

Per la classificazione binaria, si può riassumere la situazione come segue:

- Senza assunzioni su $\eta$, non c'è garanzia di convergenza a $\ell_{\mathcal{D}}(f^*)$ e il tipico tasso di convergenza in una classe parametrica (o finita) $\mathcal{H}$ è $m^{-1/2}$;
- Con l'assunzione di Lipschitz su $\eta$, il tipico tasso di convergenza a $\ell_{\mathcal{D}}(f^*)$ è $m^{-1/(d+1)}$, esponenzialmente più lenta del caso parametrico del punto precedente;

Si noti che il tasso di convergenza $m^{-1/(d+1)}$ implica che per andare vicino al rischio di Bayes di massimo $\varepsilon$, c'è bisogno di un *training set* di dimensioni dell'ordine di $\varepsilon^{-(d+1)}$. **Questa dipendenza esponenziale dal numero di *feature* è conosciuta come maledizione della dimensionalità (*curse of dimensionality*) e si riferisce alla difficoltà di apprendimento in un contesto non parametrico quando i *data point* si trovano in uno spazio ad alta dimensione (con tante *feature*)**.

# 8 Analisi del rischio per gli algoritmi NN

In questa sezione si analizzerà il rischio della funzione *zero-one loss* su classificatori binari 1-NN rispetto al *training set*.

## 8.1 Rischio statistico di 1-NN

Con dovute assunzioni sulla distribuzione $\mathcal{D}$, si può dimostrare che:

$$\mathbb{E}[\ell_{\mathcal{D}}(A(S_m))] \leq 2\ell_{\mathcal{D}}(f^*) + \varepsilon_m$$

Dove $A$ denota l'algoritmo 1-NN, $S_m$ il *training set* di dimensione $m$ e $\varepsilon_m$ **è una quantità che si azzera per** $m \to \infty$. Il fatto che si riesca a confrontare $\mathbb{E}[\ell_{\mathcal{D}}(A(S_m))]$ direttamente con il rischio di Bayes mostra come, in un certo senso, gli algoritmi 1-NN siano potenti.

Più nello specifico, **assumendo che** $\eta$ **sia una funzione di Lipschitz su** $\mathcal{X}$ **con una costante** $c > 0$, si può dire che:

$$\begin{aligned}\mathbb{E}[\ell_{\mathcal{D}}(A(S_m))] &\leq 2\ell_{\mathcal{D}}(f^*) + c\,\mathbb{E}\left[\|X - X_{\pi(S,X)}\|\right] \\ &\leq 2\ell_{\mathcal{D}}(f^*) + 4c\sqrt{d}m^{-1/(d+1)}\end{aligned}$$

Si può quindi dire che asintoticamente, con $m \to \infty$, il rischio di 1-NN oscilla tra $\ell_{\mathcal{D}}(f^*)$ e $2\ell_{\mathcal{D}}(f^*)$.

# 9 Predittori lineari

Un predittore lineare su $\mathcal{X} = \mathbb{R}^d$ è una funzione $h : \mathbb{R}^d \to \mathbb{R}$ tale che $h(x) = f(w^\top x)$ con $w \in \mathbb{R}^d$; $f : \mathbb{R} \to \mathbb{R}$ è detta funzione di attivazione.

Nelle attività di regressione lineare, $f$ è la funzione identità, e quindi $h(x) = w^\top x$.

In quelle di classificazione lineare invece, $h(x) = \text{sgn}(w^\top x - c)$ con $c \in \mathbb{R}$, dove:

$$\text{sgn}(z) = \begin{cases} 1 & z > 0 \\ 0 & \text{altrimenti} \end{cases}$$

## 9.1 Iperpiani

Un iperpiano può essere descritto da un'equazione lineare della seguente forma:

$$w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = c$$

$$\downarrow$$

$$\underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}^\top}_{w} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}}_{x} = c$$

$$\downarrow$$

$$w^\top x = c$$

Si può quindi dire che un iperpiano con coefficienti $(w, c)$ è definito da $\{x \in \mathbb{R}^d : w^\top x = c\}$ dove:

$$w^\top x = \|w\| \cdot \|x\| \cos\theta \tag{21}$$

e $\theta$ è l'angolo tra $w$ e $x$.

I sempispazi $H^+$ e $H^-$ definiti dall'iperpiano $\{x \in \mathbb{R}^d : w^\top x = c\}$ sono:

$$H^+ = \{x \in \mathbb{R}^d : w^\top x > c\}$$

$$H^- = \{x \in \mathbb{R}^d : w^\top x \leq c\}$$

Geometricamente, un classificatore lineare è definito da:

$$h(x) = \begin{cases} +1 & x \in H^+ \\ -1 & x \in H^- \end{cases}$$

## 9.2 Classificatori lineari

Si ricordi che un classificatore lineare è un predittore $h$ tale che $h(x) = \text{sgn}(w^\top x)$. Applicando (21) si ottiene che:

$$\begin{aligned} h(x) &= \text{sgn}(w^\top x) \\ &= \text{sgn}(\|w\| \cdot \|x\| \cos\theta) \\ &= \text{sgn}(\cos\theta) \end{aligned}$$

Questo ci mostra che il valore di $\|w\|$ è irrilevante e si dovrebbe prendere $\|w\| = 1$. Inoltre la *zero-one loss* $\mathbb{I}\{h(x_t) \neq y_t\}$ può essere riscritta come $\mathbb{I}\{y_t w^\top x_t \leq 0\}$.

Sia $\mathcal{H}_d$ la famiglia di classificatori lineari $h(x) = \text{sgn}(w^\top x)$ con $w \in \mathbb{R}^d : \|w\| = 1$. Si consideri l'algoritmo ERM per la *zero-one loss*:

$$h_S = \operatorname*{argmin}_{h \in \mathcal{H}_d} \frac{1}{m} \sum_{t=1}^{m} \mathbb{I}\{h(x_t) \neq y_t\}$$
$$= \operatorname*{argmin}_{w \in \mathbb{R}: \|w\|=1} \frac{1}{m} \sum_{t=1}^{m} \mathbb{I}\{y_t w^\top x_t \leq 0\} \tag{22}$$

**Sfortunatamente, è improbabile esista un implementazione efficiente di ERM per classificatori lineari con la *zero-one loss*.** Infatti, il problema di decisione associato alla ricerca di $h_S$ è NP-completo.

### 9.2.1 Separabilità lineare

Il problema ERM (22) diventa più semplice quando il *training set* è linearmente separabile, ovvero quando esiste un classificatore lineare con *training error* nullo, e quindi si può dividere linearmente tutti i *data point* in etichette differenti.

Si noti che il problema (22) può essere rappresentato da un sistema di disequazioni lineari:

$$y_t w^\top x_t > 0 \qquad t = 1, \ldots, m$$
$$\Downarrow\Downarrow$$
$$\begin{cases} y_1 w^\top x_1 > 0 \\ \vdots \\ y_m w^\top x_m > 0 \end{cases}$$

**Se il *training set* è linearmente separabile allora il sistema ha almeno una soluzione, che può essere trovata in tempo polinomiale utilizzando un algoritmo per la programmazione lineare**.

Per risolvere il sistema di disequazioni si può usare l'algoritmo del simplesso. Tuttavia esiste un algoritmo ancora più semplice: **l'algoritmo *Perceptron***.

### 9.2.2 *Perceptron*

Il *Perceptron* è in grado di risolvere il problema (22) in presenza di un *training set* linearmente separabile. Per trovare l'iperpiano separatore scorre gli esempi del *training set* uno dopo l'altro. Partendo da un classificatore lineare iniziale, lo testa su ogni esempio e, in caso di errore, corregge l'iperpiano associato.

**Data:** *Training set* $(x_1, y_1), \ldots, (x_m, y_m)$
$w = (0, \ldots, 0)$
**while** *true* **do**
  **for** $t = 1, \ldots, m$ **do** `// epoca`
    **if** $y_t w^\top x_t \leq 0$ **then**
      $w \leftarrow w + y_t x_y$ `// aggiornamento`
  **end**
  **if** *nessun aggiornamento nell'ultima epoca* **then break**
**end**
**Output:** $w$

**Algoritmo 2:** *Perceptron* (per i casi linearmente separabili)

Geometricamente, ogni aggiornamento sposta l'iperpiano $w$ verso $x_t$ se $y_t = 1$; se invece $y_t = -1$, $w$ viene allontanato da $x_t$.

Si può dimostrare che il *Perceptron* termina sempre in presenza di un *training set* linearmente separabile.

**Teorema 2** (Convergenza di *Perceptron*)**.** *Sia* $(x_1, y_1), \ldots, (x_m, y_m)$ *un training set linearmente separabile. Allora il Perceptron terminerà sempre dopo un numero di aggiornamenti non più grande di:*

$$\left( \min_{u \,:\, \gamma(u) \geq 1} \|u\|^2 \right) \cdot \left( \max_{t \,=\, 1, \ldots, m} \|x_t\|^2 \right)$$

*dove* $\gamma(u)$ *è il punto più vicino all'iperpiano* $u$*:*

$$\gamma(u) = \min_{t=1,\ldots,m} y_t u^\top x_t$$

Si noti che questo teorema non implica che il *Perceptron* termini in tempo polinomiale.

## 9.3 Regressione lineare

Nella regressione lineare i predittori sono funzioni lineari $h : \mathbb{R}^d \to \mathbb{R}$ parametrizzate da un vettore $w \in \mathbb{R}^d$ di coefficienti reali, ovvero:

$$h(x) = w^\top x$$

I coefficienti del predittore dell'algoritmo ERM per la *square loss* la regressione lineare sono:

$$\begin{aligned} w_S &= \operatorname*{argmin}_{w \in \mathbb{R}^d} \sum_{t=1}^{m} (w^\top x_t - y_t)^2 \\ &= \operatorname*{argmin}_{w \in \mathbb{R}^d} \|Sw - y\|^2 \end{aligned}$$

### 9.3.1 *Ridge Regression*

Un predittore più stabile di ERM è ottenuto introducendo un regolarizzatore che aumenta l'errore di approssimazione e riduce la varianza con un effetto positivo sul rischio. La forma regolarizzata di $w_S$ è:

$$\begin{aligned} w_{S,\alpha} &= \operatorname*{argmin}_{w \in \mathbb{R}^d} \|S_w - y\|^2 + \alpha \|w\|^2 \\ &= (\alpha I + S^\top S)^{-1} S^\top y \end{aligned}$$

con $\alpha > 0$

# 10 *Online Gradient Descent*

Il *Perceptron* accede al *training set* in maniera sequenziale, elaborando ogni esempio in tempo $\Theta(d)$ dove $d$ è il numero di *feature*. Gli algoritmi che imparano sequenzialmente, come il *Perceptron*, sono anche ottimi nel gestire scenari dove nuovi dati di *training* vengono generati in ogni momento; alcuni esempi sono dati generati da sensori, mercati finanziari o interazioni. In questi casi, i protocolli tradizionali di apprendimento basati sull'utilizzo di *training set* di dimensione fissata, risultano inefficienti, siccome si dovrebbe far girare da capo il *learning algorithm* ogni volta che si aggiungono nuovi esempi nel *training set*.

## 10.1 *Online learning*

Una generalizzione di questo tipo di *learning algorithm* sequenziale, che si dirà *online learning*, può essere riassunta come segue.

**Parametri:** *Classe $\mathcal{H}$ di predittori, funzione di loss $\ell$*
L'algoritmo genera un predittore iniziale di default $h_1 \in \mathcal{H}$
**for** $t = 1, 2, \ldots$ **do**
1. Il prossimo esempio $(x_t, y_t)$ viene osservato
2. L'errore $\ell(h_t(x_t), y_t)$ del predittore corrente $h_t$ viene calcolato
3. $h_t$ viene aggiornato generando un nuovo predittore $h_{t+1} \in \mathcal{H}$

**end**

Una caratteristica tipica dell'*online learning* è che l'aggiornamento del modello $h_{t+1} \in \mathcal{H}$ è locale, ovvero dipende solo dal predittore attuale e dall'esempio attuale.

## 10.2 Analisi dell'errore di *Perceptron*

Un esempio di algoritmo di *online learning* è il *Perceptron*. Si analizzerà ora il numero massimo di errori che *Perceptron* fa su un qualsiasi flusso di esempi, anche se linearmente non separabili.

Sia $M$ il numero di errori commessi dal *Perceptron*; si ha che:

$$\forall u \in \mathbb{R}^d \qquad M \leq \sum_{t=1}^{T} h_t(u) + (\|u\|X)^2 + \|u\|X\sqrt{\sum_{t=1}^{T} h_t(u)}$$

dove:

- $h_t(u)$ è la ***hinge loss*** al tempo $t$ del vettore $u$: $h_t(u) = \max\{0, 1 - y_t u^\top x_t\}$
- $X$ è la norma massima dei *data point*: $X = \max_t \|x_t\|$

## 10.3 Rischio sequenziale e *regret*

Un predittore algoritmo *online* $A$ genera una sequenza $h_1, h_2, \ldots, \in \mathcal{H}$ di predittori. Per valutarne le performance si userà il **rischio sequenziale**

$$\frac{1}{T}\sum_{t=1}^{T} \ell(h_t(x_t), y_t)$$

che misura la *loss* media della sequenza di predittori sui primi $T$ esempi.

Un altra misura di valutazione è la ***regret***

$$\frac{1}{T}\sum_{t=1}^{T} \ell_t(h_t) - \min_{h\in\mathcal{H}} \frac{1}{T}\sum_{t=1}^{T} \ell_t(h)$$

che misura la differenza tra il rischio sequenziale dei primi $T$ predittori e il rischio sequenziale del miglior predittore presente in $\mathcal{H}$. La *regret* può essere visto come la controparte sequenziale dell'errore di variazione dello *statistical learning*.

## 10.4 *Projected Gradient Descent*

Si vedrà ora la versione *online* del *gradient descent*, algoritmo predominante per l'ottimizzazione di funzioni convesse e derivabili. Data una funzione $f$ convessa e differenziabilie, di cui si vuole trovare il minimo, il *gradient descent* funziona iterando $w_{t+1} = w_t - \eta_t \nabla f(w_t)$ partendo da un punto iniziale $w_1$ presente nel dominio di $f$, dove $\eta > 0$ è un parametro. Preso il punto $w_t$, se $\nabla f(w_t) \neq 0$, allora $w_t$ non è un minimo e l'algoritmo prenderà un nuovo punto $w_{t+1}$ andando nella direzione opposta alla direzione del gradiente $\nabla f(w_t)$.

Per poter analizzare l'*Online Gradient Descent* (OGD), bisogna comprendere il comportamento del *gradient descent* quando la funzione $f$ da minimizzare cambia ad ogni iterazione. Ci si concentrerà su predittori lineari $h(x) = w^\top x$ con $w \in \mathbb{R}^d$. Si userà la notazione $\ell_t(w) = \ell(w^\top x_t, y_t)$ e si assumerà che le *loss* $\ell_1, \ell_2, \dots$ siano convesse e derivabili in tutti i loro punti.

Viene ora mostrato l'algoritmo dell'*projected OGD*:

**Parametri:** $\eta > 0,\ U > 0$
$w_1 = 0$ `// Inizializzazione`
**for** $t = 1, 2, \dots$ **do**
  $w'_{t+1} = w_t - \frac{\eta}{\sqrt{t}} \nabla \ell_t(w_t)$
  $w_{t+1} = \underset{w:\|w\| \leq U}{\operatorname{argmin}} \|w - w'_{t+1}\|$ `// Proiezione`
**end**

**Algoritmo 3:** *Projected OGD*

Il parametro $U$ indica il raggio di una sfera dentro la quale cercare il punto con la minor *regret* (proiezione).

### 10.4.1 Analisi *regret*

Sia $G$ il valore massimo che la norma del gradiente può assumere:

$$\forall t \leq T \quad \|\nabla \ell_t(w_t)\| \leq G$$

Si può mostrare come, con qualsiasi funzione di *loss* si ha:

$$\underbrace{\frac{1}{T}\sum_{t=1}^{T} \ell_t(w_t)}_{\ell_T(w)} \leq \underbrace{\min_{u:\|u\| \leq U} \frac{1}{T}\sum_{t=1}^{T} \ell_t(u)}_{\ell_T(u^*)} + \underbrace{UG\sqrt{\frac{8}{T}}}_{regret}$$

dove:

- $\ell_T(w)$ è la *loss* media dell'algoritmo fino al passo $T$;
- $\ell_t(u^*)$ è la *loss* media del miglior predittore trovato fino al passo $T$;
- $UG\sqrt{\frac{8}{T}}$ è la *regret* dell'algoritmo.

## 10.5 OGD con *loss* fortemente convesse

Il limite mostrato nella precedente sezione non può essere migliorato in presenza di funzioni di *loss* lineari. Si può invece migliore in presenza di funzioni di *loss* convesse e mai piatte. Per formalizzare questo scenario verrà introdotto il concetto di funzione fortemente convessa.

Una funzione differenziabilie $\ell$ è $\sigma$-fortemente convessa con $\sigma > 0$ se, per ogni $w, u \in RN^d$ vale che:

$$\ell(w) - \ell(u) \leq \nabla \ell(w)^\top (w - u) - \frac{\sigma}{2}\|u - w\|^2$$

o, in altri termini, è una funzione che cresce alla stessa velocità di una funzione quadratica.

L'algoritmo di OGD per funzioni fortemente convesse non necessita del passo di proiezione.

$w_1 = 0$
**for** $t = 1, 2, \ldots$ **do**
  $w'_{t+1} = w_t - \frac{1}{\sigma t}\nabla \ell_t(w_t)$
**end**

**Algoritmo 4:** *OGD* per funzioni fortemente convesse

### 10.5.1 Analisi *regret*

L'analisi della *regret* mostra che:

$$\underbrace{\frac{1}{T}\sum_{t=1}^{T}\ell_t(w_t)}_{\ell_T(w)} \leq \underbrace{\min_{u\in\mathbb{R}^d}\frac{1}{T}\sum_{t=1}^{T}\ell_t(u)}_{\ell_T(u^*)} + \underbrace{\frac{G^2(1+\ln T)}{2\sigma T}}_{regret}$$

dove:

- $\sigma$ indica che la *loss* è $\sigma$-fortemente convessa;
- $G$ è il valore massimo che la norma del gradiente può assumere: $\max_t \|\nabla \ell_t(w_t)\| \leq G$;

# 11 *Kernel*

I predittori lineari possono soffrire di un errore di approssimazione alto in quanto sono sempre descritti da un numero di coefficienti che non può essere più grande del numero di *feature*. Una tecnica molto usata per ridurre questo problema è la ***feature expansion***.

## 11.1 *Feature expansion*

La tecnica della *feature expansion* consiste nell'aggiungere nuovi parametri costruendo nuove *feature* tramite combinazioni non lineari delle *feature* di base. Formalmente si introduce una funzione $\phi : \mathbb{R}^d \to \mathcal{H}$ che mappa i punti $\boldsymbol{x} \in \mathbb{R}^d$ in uno spazio di dimensioni maggiori $\mathcal{H}$.

Per esempio, si consideri la *feature-expansion* quadratica con la funzione $\phi : \mathbb{R}^2 \to \mathbb{R}^6$ definita da:

$$\phi(x_1, x_2) = (1, x_1^2, x_2^2, x_1, x_2, x_1 x_2)$$

Si ricordi che un iperpiano omogeneo in $\mathbb{R}^6$ con i coefficienti $\boldsymbol{w} = (w_1, \ldots, w_6)$, è un insieme di punti $\{\boldsymbol{z} \in \mathbb{R}^6 : \boldsymbol{w}^\top \boldsymbol{z} = 0\}$. Si ha quindi che:

$$\boldsymbol{w}^\top \phi(\boldsymbol{x}) = w_1 + w_2 x_1^2 + w_3 x_2^2 + w_4 x_1 + w_5 x_2 + w_6 x_1 x_2 \tag{23}$$

Questi insiemi dalla forma $\{\boldsymbol{x} \in \mathbb{R}^2 : \boldsymbol{w}^\top \phi(\boldsymbol{x}) = 0\}$ descrivono curve di secondo grado su $\mathbb{R}^2$ in quanto il grado dell'equazione (23) è due.

In generale si considererà una mappa di *feature-expansion* $\phi : \mathbb{R}^d \to \mathcal{H}$, dove $\mathcal{H} = \mathbb{R}^N$, che crea *feature* della forma:

$$\prod_{s=1}^{k} x_{v_s} \quad \text{per ogni} \quad \boldsymbol{v} \in \{1, \ldots, d\}^k \text{ e per ogni } \quad k = 0, 1, \ldots, n$$

Nell'esempio precedente (con $\mathbb{R}^6$) sono stati usati $d = n = 2$.

Fissata la $\phi$ si consideri il classificatore $h : \mathbb{R}^d \to \{-1, 1\}$ definito come:

$$h(\boldsymbol{x}) = \operatorname{sgn}\left(\boldsymbol{w}^\top \phi(\boldsymbol{x})\right) \qquad \text{dove} \quad \boldsymbol{w}^\top \phi(\boldsymbol{x}) = \sum_{i=1}^{N} w_i \phi(\boldsymbol{x})_i$$

Questo classificatore non lineare in $\mathbb{R}^d$ corrisponde ad un classificatore lineare in $\mathbb{R}^N$. Il problema però, è che $N$ cresce esponenzialmente ($N = \Theta(d^n)$), facendo diventare il calcolo di $\phi$ infattibile per valori moderatamente alti di $n$. Fortunatamente, questa barriera computazionale è stata ampiamente superata grazie al *kernel trick*.

## 11.2 *Kernel trick*

Per introdurre il *kernel trick* si vedrà un esempio: si prenda il passo di aggiornamento del *Perceptron* $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + y_t \boldsymbol{x}_t$ dove $w_1 = 0$. Il classificatore lineare appreso dal *Perceptron* sarà:

$$h(\boldsymbol{x}) = \operatorname{sgn}\left(\sum_{s \in S} y_s \boldsymbol{x}_s^\top \boldsymbol{x}\right)$$

dove $S$ è l'insieme degli indici $s$ dell'esempio di *training* $(\boldsymbol{x_s}, y_s)$. Volendo eseguire *Perceptron* in $\mathbb{R}^N$, il classificatore lineare sarà:

$$h_\phi(\boldsymbol{x}) = \operatorname{sgn}\left(\sum_{s \in S} y_s \phi(\boldsymbol{x}_s)^\top \phi(\boldsymbol{x})\right)$$

Ora la complessità computazionale di $h_\phi(\boldsymbol{x})$ dipende da quanto velocemente si riesce a calcolare $\phi(\boldsymbol{x}_s)^\top \phi(\boldsymbol{x})$.

Si ipotizzi di lavorare in $\mathbb{R}^2$ e si prenda la seguente $\phi$:

$$\phi(x_1, x_2) = (1, x_1^2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2)$$

$$\phi(\boldsymbol{x}_s)^\top \phi(\boldsymbol{x}) = \begin{pmatrix} 1 & x_{s_1}^2 & x_{s_2}^2 & \sqrt{2}x_{s_1} & \sqrt{2}x_{s_2} & \sqrt{2}x_{s_1}x_{s_2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x_1^2 \\ x_2^2 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \sqrt{2}x_1x_2 \end{pmatrix}$$

$$= 1 + x_{s_1}^2 x_1^2 + x_{s_1}^2 x_2^2 + 2x_{s_1}^2 x_1 + 2x_{s_1}^2 x_2 + 2x_{s_1}x_{s_2}x_1x_2$$

Nonostante $d$ ed $n$ siano bassi, il carico computazionale di questa operazione non è indifferente. Per semplificarla si può facilmente notare che:

$$\phi(\boldsymbol{x}_s)^\top \phi(\boldsymbol{x}) = (1 + \boldsymbol{x}^\top \boldsymbol{x}')^2 \tag{24}$$

Si è quindi trovata un'alternativa efficiente per lo stesso tipo di calcolo. È proprio questa l'idea del *kernel trick*, ovvero **trovare una funzione efficiente** $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ **tale che**:

$$\forall \boldsymbol{x}, \boldsymbol{x}' \in \mathbb{R}^d \qquad K(\boldsymbol{x}, \boldsymbol{x}') = \phi(\boldsymbol{x}_s)^\top \phi(\boldsymbol{x})$$

### 11.2.1 *Kernel Perceptron*

Dato un *kernel* $K$, il classificatore lineare generato dal *Kernel Perceptron* è:

$$h_K(\boldsymbol{x}) = \operatorname{sgn}\left(\sum_{s \in S} y_s K(\boldsymbol{x}_s, \boldsymbol{x})\right)$$

Viene ora mostrato lo pseudo-codice del *Kernel Perceptron*:

$S \leftarrow \{\}$
**for** $t = 1, 2, \ldots$ **do**
  Il prossimo esempio $(\boldsymbol{x}_t, y_t)$ viene osservato
  $\hat{y}_t = \operatorname{sgn}\left(\sum_{s \in S} y_s K(\boldsymbol{x}_s, \boldsymbol{x}_t)\right)$
  **if** $\hat{y}_t \neq y_t$ **then**
    $S \leftarrow S \cup \{t\}$
  **end**
**end**

**Algoritmo 5:** *Kernel Perceptron*

### 11.2.2 *Kernel polinomiale*

Generalizzando il *kernel* quadratico (24), si ottiene il *kernel* polinomiale di grado $n$:

$$K_n(\boldsymbol{x}, \boldsymbol{x}') = (1 + \boldsymbol{x}^\top \boldsymbol{x}')^n$$

### 11.2.3 *Kernel* gaussiano

Un altro tipo di *kernel* è quello gaussiano:

$$K_\gamma(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{1}{2\gamma}\|\boldsymbol{x} - \boldsymbol{x}'\|^2\right) \qquad \gamma > 0$$

dove il parametro $\gamma$ è un fattore di scala per il prodotto $\boldsymbol{x}^\top \boldsymbol{x}'$.

## 11.3 Verifica del *kernel*

Dato uno spazio $\mathcal{X}$ e una funzione simmetrica $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ come si può capire se $K$ è un *kernel*?

In altre parole, si vuole capire se esiste una mappatura delle *feature* $\phi_K : \mathcal{X} \to \mathcal{H}_k$, dove $\mathcal{H}_k$ è uno spazio lineare con l'operazione di prodotto $\langle \cdot, \cdot \rangle_K$ tale che:

$$\forall \boldsymbol{x}, \boldsymbol{x}' \in \mathcal{X} \qquad \langle \phi_K(\boldsymbol{x}), \phi_K(\boldsymbol{x}') \rangle_K = K(\boldsymbol{x}, \boldsymbol{x}')$$

Un metodo molto semplice di verifica di $K$ è il seguente: $K$ è un *kernel* se e solo se per ogni $m \in \mathbb{N}$ e per ogni $\boldsymbol{x_1}, \ldots, \boldsymbol{x_m} \in \mathcal{X}$, la matrice $\boldsymbol{K}$ grande $m \times m$ tale che $\boldsymbol{K}_{i,j} = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, è semidefinita positiva, ovvero $\boldsymbol{z}^\top \boldsymbol{K} \boldsymbol{z} \geq 0$ per ogni $\boldsymbol{z} \in \mathbb{R}^m$.

$\mathcal{H}_K$ può essere definito come l'insieme di tutte le combinazioni lineari di $K(\boldsymbol{x}, \cdot)$ con coefficienti $\alpha$ e punti $x \in \mathcal{X}$ arbitrari:

$$\mathcal{H}_K = \left\{ \sum_{i=1}^{N} \alpha_i K(\boldsymbol{x}_i, \cdot) : \boldsymbol{x_1}, \ldots, \boldsymbol{x_N} \in \mathcal{X}, \alpha_1, \ldots, \alpha_N \in \mathbb{R}, N \in \mathbb{N} \right\}$$

Un elemento $f \in \mathcal{H}_K$ è una funzione $f : \mathcal{H} \to \mathbb{R}$ tale che:

$$f(x) = \sum_{i=1}^{N} \alpha_i K(\boldsymbol{x_i}, \boldsymbol{x})$$

## 11.4 Convergenza del *Kernel Perceptron*

Partendo dal teorema 2 si può stabilire il numero massimo di aggiornamenti $M$ che il *Kernel Perceptron* compierà:

$$\begin{aligned}
M &\leq \|u\|^2 \left( \max_t \|x_t\|^2 \right) \\
&= \|f\|_K^2 \left( \max_t \{ \|\phi_K(\boldsymbol{x})\|_K^2 \} \right) \\
&= \left\| \sum_{i=1}^{N} \alpha_i K(\boldsymbol{x_i}, \cdot) \right\|_K^2 \left( \max_t \langle K(\boldsymbol{x}, \cdot), K(\boldsymbol{x}, \cdot) \rangle_K \right) \\
&= \left\langle \sum_{i=1}^{N} \alpha_i K(\boldsymbol{x_i}, \cdot), \sum_{j=1}^{N} \alpha_j K(\boldsymbol{x_j}, \cdot) \right\rangle_K^2 \left( \max_t K(\boldsymbol{x}, \boldsymbol{x}) \right) \\
&= \left( \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j K(\boldsymbol{x}_i, \boldsymbol{x}_j) \right) \left( \max_t K(\boldsymbol{x}, \boldsymbol{x}) \right)
\end{aligned}$$

## 11.5 *Kernel Ridge Regression*

La formula chiusa del predittore *Ridge Regression* è:

$$\boldsymbol{w}_{S,\alpha} = (\alpha I + S^\top S)^{-1} S^\top \boldsymbol{y}$$

Applicando il kernel $K = S^\top S$ si ottiene il predittore:

$$\boldsymbol{w}^\top \boldsymbol{x} = \boldsymbol{y}^\top (\alpha I + \boldsymbol{K})^{-1} \boldsymbol{k}(\cdot)$$

dove:

- $k(\cdot)$ è il vettore $(K(\boldsymbol{x}_1, \cdot), \ldots, K(\boldsymbol{x}_m, \cdot))$ di funzioni $K(\boldsymbol{x}_t, \cdot) = \langle \phi_K(\boldsymbol{x}_t, \cdot) \rangle_K$.