CORSO DI LAUREA TRIENNALE
IN
INGEGNERIA INFORMATICA

TESI DI LAUREA

Support Vector Machines vs ensemble methods: studio comparativo e applicazioni

LAUREANDO

Fabio Bozzoli

RELATORE

Chiar.mo Prof. Carlo Drago

ANNO ACCADEMICO 2024-2025

Indice

Capitolo 1

Introduzione

Il *Machine Learning* [2] è una disciplina dell'intelligenza artificiale che si occupa dello sviluppo e dell'implementazione di algoritmi e modelli statistici capaci di eseguire compiti senza istruzioni esplicite, basandosi sull'inferenza. Attraverso l'analisi di grandi quantità di dati, questi algoritmi sono in grado di individuare pattern nascosti e relazioni complesse, migliorando progressivamente le proprie prestazioni. Grazie a questo processo di apprendimento automatico, i sistemi informatici possono effettuare previsioni sempre più accurate, addestrandosi su insiemi di dati iniziali (training set) e affinando le loro capacità con l'esperienza.

Attualmente il machine learning viene utilizzato in qualsiasi ambito: quando interagiamo con le banche, quando effettuiamo acquisti online, quando utlizziamo i social media, ecc... In generale esso viene utilizzato per rendere la nostra esperienza efficiente, facile e sicura.

Gli algoritmi di machine learning rientrano in genere in una delle tre seguenti macro categorie: **apprendimento supervisionato**, **apprendimento non supervisionato**, **apprendimento per rinforzo**.

1.1 Apprendimento supervisionato

L'apprendimento supervisionato [1] utilizza un set di addestramento composto da input e output corretti per insegnare ai modelli a produrre il risultato desiderato. L'algoritmo ottimizza la propria precisione riducendo l'errore tramite una funzione di perdita.

Si distingue in due categorie principali:

- *Classificazione*: assegna i dati a categorie specifiche, identificando schemi e caratteristiche rilevanti. Tra gli algoritmi più comuni troviamo SVM, alberi decisionali, k-nearest neighbor e foreste casuali.

- *Regressione*: analizza la relazione tra variabili dipendenti e indipendenti, spesso usata per previsioni come l'andamento delle vendite. Gli algoritmi più diffusi includono regressione lineare, logistica e polinomiale.

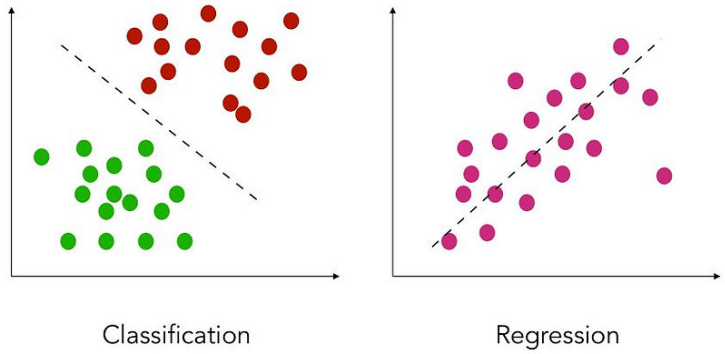

Figura 1.1: A sinistra un esempio di funzione di decisione per classificazione, a destra una funzione di decisione di regressione.

1.2 Apprendimento non supervisionato

A differenza dell'apprendimento supervisionato, l'apprendimento non supervisionato lavora con dati non etichettati. In questo caso, l'obiettivo è quello di trarre conclusioni grazie alla scoperta di pattern nascosti fra i dati senza alcuna guida esplicita. Questo tipo di apprendimento è particolarmente utile quando non si dispone di etichette o quando il costo di etichettatura è proibitivo.

Due delle tecniche principali di apprendimento non supervisionato sono il ***clustering*** e l'***associazione***:

- Clustering: l'obiettivo è raggruppare gli esempi di dati in cluster o gruppi omogenei, basati su caratteristiche simili. Esempi di tali algoritmi sono il k-means, il DBSCAN e le reti neurali auto-organizzanti (SOM).

- Associazione: si cerca di trovare regole interessanti e significative che descrivano grandi porzioni di dati, come nel caso delle analisi delle transazioni nei supermercati per identificare prodotti frequentemente acquistati insieme.

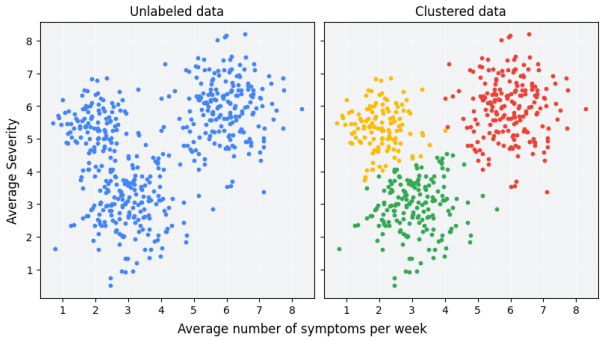

Figura 1.2: A sinistra è mostrato un insieme di dati di input, mentre a destra la loro suddivisione in gruppi ottenuta con un algoritmo di clustering.

L'apprendimento non supervisionato è essenziale per scoprire strutture sottostanti nei dati e per ridurre la dimensionalità del problema in questione, come succede nel caso dell'analisi delle componenti principali (PCA) e del t-SNE, che aiutano a visualizzare dati complessi in spazi a dimensioni ridotte. Nel caso della PCA per esempio, partendo da molte variabili correlate fra loro si riduce la dimensionalità del dataset, mantenendo elevata la variabilità fra le informazioni. Può essere considerato un metodo di sintesi dei dati.

1.3 Apprendimento per rinforzo

L'apprendimento attraverso l'interazione con l'ambiente è uno dei concetti fondamentali dell'intelligenza e della conoscenza [3]. Fin dalla nascita gli esseri umani apprendono osservando le conseguenze delle proprie azioni, senza necessariamente avere un insegnante esplicito e dedicato. Questo principio è alla base del *reinforcement learning*, un approccio computazionale che si concentra sull'apprendimento basato sull'esperienza diretta e sul raggiungimento di obiettivi.

Gli algoritmi di reinforcement learning sono spesso modellati come processi decisionali di Markov, dove un agente interagisce con l'ambiente, esegue azioni e riceve ricompense in base al proprio comportamento.

Anche se il reinforcement learning non usa delle etichette specifiche, non può essere considerato una sottocategoria dell'apprendimento non supervisionato, poiché esso si concentra sull'ottimizzazione di una funzione di ricompensa piuttosto che sulla ricerca di strutture nei dati.

Negli ultimi anni il reinforcement learning ha giocato un ruolo chiave nel progresso dell'intelligenza artificiale, con applicazioni in robotica, finanza, giochi (come AlphaGo) e sistemi autonomi.

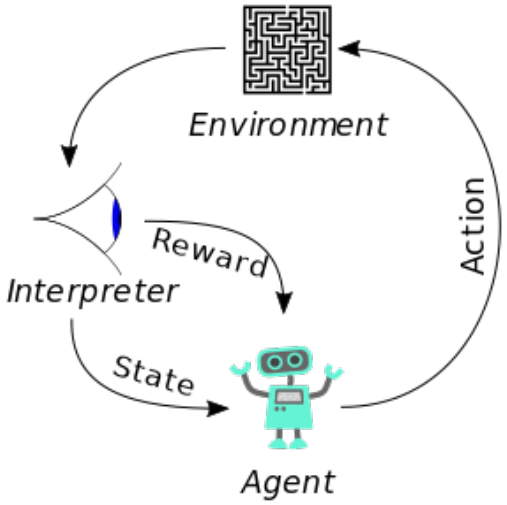

Figura 1.3: Schema rappresentativo di un agente che applica il reinforcement learning: l'agente esegue un'azione nell'ambiente, che in risposta fornisce una ricompensa e aggiorna il proprio stato.

1.4 Metriche di valutazione

Prima di mettere un modello di *Machine Learning* in produzione, è fondamentale valutarne le prestazioni per garantirne l'affidabilità e confrontarlo con altri modelli [4]. Per farlo, i dati disponibili vengono suddivisi in due insiemi principali:

- **Training set**: utilizzato per addestrare il modello;
- **Test set**: utilizzato per valutare il modello, contenente dati mai visti durante l'addestramento.

Il test set permette di stimare la capacità del modello di generalizzare su nuovi dati, attraverso metriche di valutazione appropriate.

1.4.1 Metriche di valutazione nei modelli supervisionati

La valutazione delle prestazioni di un modello dipende dal tipo di problema affrontato. Nel caso di apprendimento supervisionato, il processo risulta più semplice grazie alla presenza delle etichette dei dati, che permettono di confrontare le predizioni con i valori reali.

Tuttavia, nel caso di apprendimento non supervisionato, la valutazione diventa più complessa in quanto non si dispone di etichette di riferimento. In questa trattazione ci concentreremo sulle metriche relative all'apprendimento supervisionato, che è l'ambito di interesse per il presente lavoro.

Le metriche di valutazione variano a seconda della tipologia di problema supervisionato:

- Classificazione: il modello assegna ogni esempio a una categoria specifica.
- Regressione: il modello prevede un valore numerico continuo.

1.4.2 Metriche per la classificazione

Nei problemi di classificazione, è utile introdurre il concetto di matrice di confusione (*confusion matrix*), che riassume le prestazioni del modello mostrando il numero di predizioni corrette e errate per ciascuna classe.

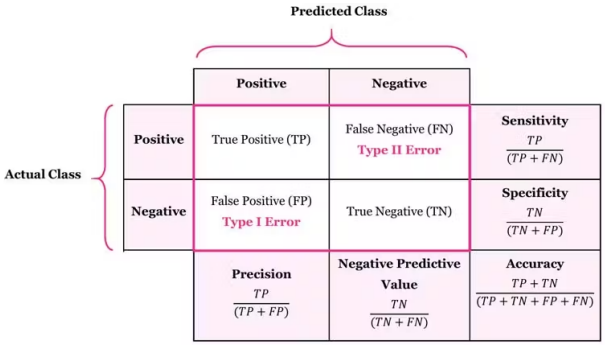

Figura 1.4: Schematizzazione della confusion matrix.

Nel caso di classificazione binaria, la matrice di confusione è composta da quattro valori:

- *True Positives* (TP): esempi classificati come positivi e realmente appartenenti alla classe positiva.
- *True Negatives* (TN): esempi classificati come negativi e realmente appartenenti alla classe negativa.
- *False Positives* (FP): esempi classificati come positivi, ma in realtà appartenenti alla classe negativa.
- *False Negatives* (FN): esempi classificati come negativi, ma in realtà appartenenti alla classe positiva.

A partire da questi valori, possiamo definire le seguenti metriche di valutazione:

- **Accuratezza (Accuracy)**: misura la percentuale di istanze classificate correttamente rispetto al totale.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precisione (Precision)**: indica la proporzione di istanze realmente positive tra quelle classificate come positive.

$$Precision = \frac{TP}{TP + FP}$$

- **Recall (Sensibilità o True Positive Rate)**: misura la capacità del modello di identificare correttamente tutte le istanze positive.

$$Recall = \frac{TP}{TP + FN}$$

- **F1-score**: rappresenta la media armonica tra precisione e recall, utile quando è necessario bilanciare i due valori.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

- **AUC-ROC**: la curva ROC (*Receiver Operating Characteristic*) è una rappresentazione grafica delle prestazioni di un classificatore al variare della soglia di decisione. Essa viene costruita riportando il tasso di veri positivi (True Positive Rate, TPR) sull'asse y e il tasso di falsi positivi (False Positive Rate, FPR) sull'asse x, per diverse soglie di classificazione. L'**AUC** (*Area Under the Curve*) rappresenta l'area sottesa dalla curva ROC e fornisce una misura sintetica della capacità discriminativa del

modello. Un valore di AUC = 1 indica un classificatore perfetto, mentre un valore di AUC = 0.5 corrisponde a un classificatore casuale.

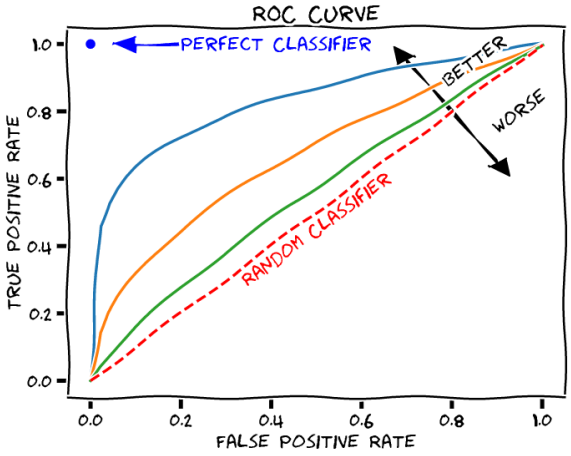

Figura 1.5: La bisettrice del primo e terzo quadrante rappresenta un classificatore casuale ($AUC = 0.5$). Se la curva ROC si trova al di sopra di questa diagonale, significa che il modello possiede capacità predittive superiori al caso casuale. Un'area più ampia sotto la curva indica prestazioni migliori.

1.4.3 Metriche per la regressione

Nei problemi di regressione, il modello deve prevedere un valore numerico e le metriche di valutazione misurano l'errore tra i valori predetti e quelli reali. Tra le principali metriche troviamo:

- **Errore Quadratico Medio (Mean Squared Error, MSE)**: misura la media dei quadrati degli errori tra valori reali e predetti.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- **Errore Assoluto Medio (Mean Absolute Error, MAE)**: calcola la media del valore assoluto degli errori.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- **R-quadrato (R^2 Score)**: misura quanto bene il modello spiega la variabilità dei dati.

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$$

1.5 Underfitting vs overfitting

Il machine learning è fondamentalmente incentrato sulla generalizzazione, ovvero la capacità di un modello di effettuare previsioni accurate su dati non visti [7].

Nel caso dell'apprendimento supervisionato, si utilizza un insieme finito di esempi etichettati per selezionare una funzione all'interno di un insieme di ipotesi (hypothesis set). La funzione scelta viene poi usata per classificare sia gli esempi di addestramento che nuovi dati.

La scelta dell'insieme delle ipotesi (*hypothesis set*) gioca un ruolo fondamentale nella capacità di un modello di apprendere e generalizzare su nuovi dati. In generale, esistono due scenari opposti:

- Se il modello è troppo complesso, può adattarsi perfettamente ai dati di addestramento, riducendo l'errore a zero. Tuttavia, questo può portare a un overfitting, ovvero un'eccessiva dipendenza dai dati di training, che riduce la capacità del modello di generalizzare su nuovi esempi.

- Se il modello è troppo semplice, potrebbe non essere in grado di catturare la struttura sottostante dei dati e commettere errori già durante l'addestramento (underfitting). In questo caso, il modello rischia di essere troppo rigido e incapace di rappresentare adeguatamente la relazione tra input e output.

L'obiettivo è trovare un compromesso tra queste due situazioni. La sfida consiste nel bilanciare complessità e capacità predittiva per ottenere un modello che funzioni bene sia sui dati di addestramento che su quelli mai visti prima.

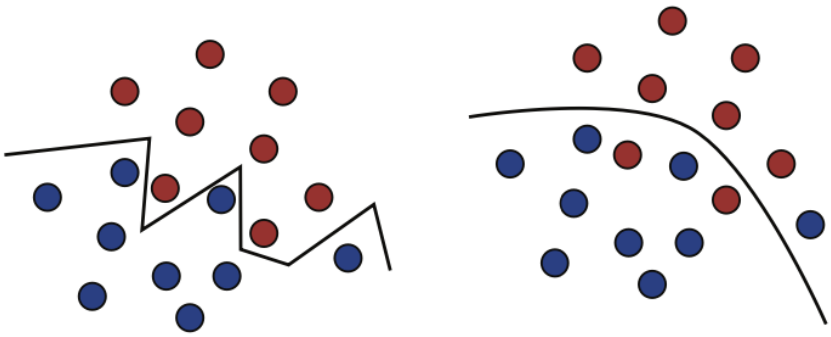

Figura 1.6: Il pannello di sinistra mostra un modello complesso che separa perfettamente i dati di addestramento, ma con scarsa capacità di generalizzazione. Al contrario, il pannello di destra presenta un confine decisionale più semplice, che pur commettendo alcuni errori, potrebbe generalizzare meglio su nuovi dati.

Capitolo 2

Support Vector Machines

Le *Support Vector Machines* [6] sono uno degli algoritmi più solidi dal punto di vista teorico e tra i più efficaci in ambito pratico nel machine learning moderno.

Consideriamo uno spazio di input $\mathbb{X} \subset \mathbb{R}^N$ e uno spazio di output binario $\mathbb{Y} = \{-1, 1\}$. L'obiettivo è quello di trovare una funzione $f : \mathbb{X} \to \mathbb{Y}$ che assegni correttamente le istanze di un dataset di addestramento S, composto da m coppie (x_i, y_i), con dati estratti da una distribuzione sconosciuta D. Ossia:

$$S = \{(\mathbf{x}_i, y_i), \quad \mathbf{x}_i \in \mathbb{X}, \quad y_i \in \mathbb{Y}, \quad i = 1, 2, ..., m\}$$

Si vuole quindi trovare un classificatore binario $h \in \mathcal{H}$ che minimizzi l'errore di generalizzazione:

$$R_D(f) = P_{x \sim D}[h(x) \neq f(x)]$$

Diversi insiemi di ipotesi $\mathcal{H}$ possono essere scelti per questo compito. Secondo il principio del *rasoio di Occam*, modelli con bassa complessità offrono migliori garanzie di apprendimento a parità di altre condizioni. Dunque un'ipotesi naturale con bassa complessità è il *classificatore lineare* definito dalla seguente famiglia di funzioni:

$$\mathcal{H} = \{\mathbf{x} \to \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) : \mathbf{w} \in \mathbb{R}^N, b \in \mathbb{R}\}$$

Quindi un problema di classificazione lineare consiste nel trovare un iperpiano che separi le classi nello spazio N-dimensionale. L'equazione generale di un iperpiano in tale spazio è proprio:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

dove $\mathbf{w}$ è il vettore normale all'iperpiano e b è uno scalare detto anche *bias*.

2.1 Caso linearmente separabile

Si assuma che il dataset di punti a disposizione S sia linearmente separabile [7], ovvero che esista un iperpiano in grado di separare perfettamente i punti appartenenti alle due classi, come illustrato nella Figura 2.1.

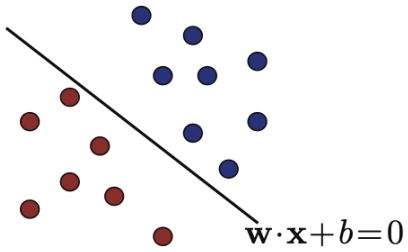

Figura 2.1: Esempio di dati linearmente separabili.

Tuttavia, esistono infiniti iperpiani separatori che soddisfano questa condizione. Ma non tutti hanno la stessa capacità di generalizzazione. L'obiettivo di un buon classificatore è trovare l'iperpiano che massimizza la distanza dai punti più vicini, evitando sia l'overfitting (iperpiano troppo complesso) che l'underfitting (iperpiano troppo semplice).

Per quantificare questa distanza utilizziamo il concetto di **margine geometrico**. Il margine misura quanto un iperpiano è distante dai punti più vicini di ciascuna classe.

Definizione 2.1.1 (Margine geometrico)**.** Il margine geometrico di un classificatore lineare $h(x) = \mathbf{w} \cdot \mathbf{x} + b$ in un punto $\mathbf{x}$ è definito come la distanza euclidea tra il punto e l'iperpiano $\mathbf{w} \cdot \mathbf{x} + b = 0$:

$$\rho_h(\mathbf{x}) = \frac{|\mathbf{w} \cdot \mathbf{x} + b|}{||\mathbf{w}||_2} \tag{2.1}$$

Per un insieme di dati $S = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m)$, il margine geometrico del classificatore è il valore minimo del margine su tutti i punti del campione:

$$\rho_h = \min_{i \in \{1,2,...,m\}} \rho_h(\mathbf{x}_i) \tag{2.2}$$

L'iperpiano che massimizza questa distanza viene detto *iperpiano a margine massimo* (*maximum-margin hyperplane*), illustrato nella Figura 2.2. Per trovarlo, dobbiamo risolvere un problema di ottimizzazione vincolata, che garantisca il massimo margine e la corretta classificazione di tutti i punti.

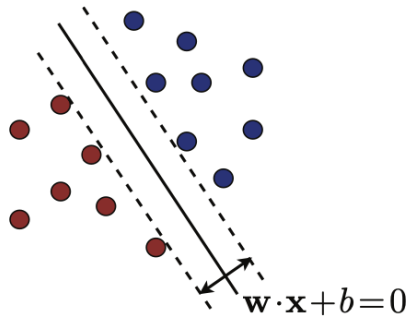

Figura 2.2: Esempio *maximum-margin hyperplane*.

Dal punto di vista teorico, questa soluzione è giustificata dal fatto che massimizzare il margine riduce il rischio di generalizzazione. Inoltre possiamo anche osservare che maggiore è il margine, maggiore sarà anche la robustezza della soluzione. Questa proprietà rende l'SVM una scelta ottimale.

Il problema della scelta del miglior iperpiano di separazione diventa così un problema di ottimizzazione vincolata. Infatti sappiamo che:

- se un punto $\mathbf{x}_i$ appartiene alla classe positiva ($y_i = +1$), allora deve valere $\mathbf{w} \cdot \mathbf{x}_i + b \geq 0$, ossia $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 0$;
- se un punto $\mathbf{x}_i$ appartiene alla classe negativa ($y_i = -1$), allora deve valere $\mathbf{w} \cdot \mathbf{x}_i + b \leq 0$, ossia $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 0$;

quindi l'iperpiano deve rispettare la seguente condizione per tutti i punti del dataset:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 0 \tag{2.3}$$

2.1.1 Formulazione del problema primale

In particolare quello che si vuole trovare è l'iperpiano $\mathbf{w} \cdot \mathbf{x} + b = 0$ in grado di soddisfare il seguente problema di massimo:

$$\rho = \max_{\mathbf{w},b:\ y_i(\mathbf{w}\cdot\mathbf{x}_i+b)\geq 0} \ \min_{i\in\{1,2,\ldots,m\}} \frac{|\mathbf{w} \cdot \mathbf{x}_i + b|}{||\mathbf{w}||} = \max_{\mathbf{w},b} \ \min_{i\in\{1,2,\ldots,m\}} \frac{y_i(\mathbf{w} \cdot \mathbf{x}_i + b)}{||\mathbf{w}||} \tag{2.4}$$

Inoltre abbiamo che l'operazione (2.3) è invariante rispetto alla moltiplicazione della coppia $(\mathbf{w}, b)$ per uno scalare positivo. Pertanto, possiamo ridurre il problema a considerare solo le coppie $(\mathbf{w}, b)$ scalate in modo che:

$$\min_{i\in\{1,2,\ldots,m\}} y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1$$

In questo modo si ottiene che

$$\rho = \max_{\mathbf{w},b:\ \min_{i\in\{1,2,\ldots,m\}} y_i(\mathbf{w}\cdot\mathbf{x}_i+b)=1} \frac{1}{||\mathbf{w}||} = \max_{\mathbf{w},b:\ \forall i\in\{1,2,\ldots,m\}\ y_i(\mathbf{w}\cdot\mathbf{x}_i+b)\geq 1} \frac{1}{||\mathbf{w}||}$$

che scriveremo per comodità:

$$\rho = \max_{y_i(\mathbf{w}\cdot\mathbf{x}_i+b)\geq 1} \frac{1}{||\mathbf{w}||} \tag{2.5}$$

Infatti se si sa che per i punti più vicini all'iperpiano la quantità $y_i(\mathbf{w}\cdot\mathbf{x}_i+b) = 1$, allora si avrà certamente che per tutti i punti presenti nel dataset S si deve necessariamente avere che $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$.

La figura 2.3 illustra la soluzione del problema di ottimizzazione (2.5). Oltre il *maximum-margin hyperplane*, la figura mostra anche gli iperpiani marginali, rappresentati dalle linee tratteggiate.

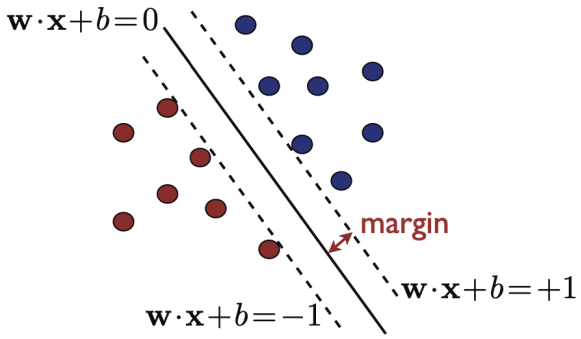

Figura 2.3: Soluzione del problema di ottimizzazione e iperpiani marginali.

Poiché massimizzare la quantità $\frac{1}{||\mathbf{w}||}$ equivale a minimizzare $\frac{1}{2}||\mathbf{w}||^2$, il problema di ottimizzazione delle SVM nel caso linearmente separabile può essere riformulato nella seguente forma:

$$\begin{aligned} \min \quad & \frac{1}{2}\mathbf{w}\cdot\mathbf{w} \\ \text{soggetto a} \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1,\ \forall i \in \{1,2,...,m\} \end{aligned} \tag{2.6}$$

Osserviamo così che la funzione obiettivo $F : \mathbf{w} \to \frac{1}{2}||\mathbf{w}||^2$ è una differenziabile. In particolare si ha $\nabla F(\mathbf{w}) = \mathbf{w}$ e $\nabla^2 F(\mathbf{w}) = \mathbf{I}$, ossia è una funzione strettamente convessa.

2.1.2 Teorema di Wolfe e KKT

Dunque per risolvere tale problema di ottimizzazione è necessario andare a definire il concetto di **lagrangiana** del problema.

Sia $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ una funzione differenziabile con continuità e siano $h : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ e $g : \mathbb{R}^n \longrightarrow \mathbb{R}^p$ definite come:

$$h(x) = \begin{pmatrix} h_1(x) \\ h_2(x) \\ \vdots \\ h_n(x) \end{pmatrix} \qquad g(x) = \begin{pmatrix} g_1(x) \\ g_2(x) \\ \vdots \\ g_p(x) \end{pmatrix}$$

Definizione 2.1.2 (Funzione Lagrangiana)**.** La ***funzione Lagrangiana*** (o Lagrangiana) del problema

$$\begin{aligned} \min \quad & f(x) \\ \text{soggetto a} \quad & \begin{cases} h(x) = 0 \\ g(x) \leq 0 \end{cases} \end{aligned}$$

è definita come

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \lambda^T h(x) + \mu^T g(x)$$

dove $\lambda \in \mathbb{R}^m$, $\mu \in \mathbb{R}^p$ sono i vettori dei ***moltiplicatori di Lagrange***.

La Lagrangiana è quindi una funzione $\mathcal{L} : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}$, di classe C^1, che tiene conto di entrambi gli elementi che definiscono il problema vincolato, ossia la funzione obiettivo e i vincoli. In forma esplicita possiamo scrivere:

$$\mathcal{L}(x, \lambda, \mu) = f(x) + \sum_{i=1}^{m} \lambda_i h_i(x) + \sum_{j=1}^{p} \mu_j g_j(x)$$

Consideriamo ora il seguente teorema che garantisce delle condizioni necessarie per la soluzione di un problema di programmazione non lineare.

Teorema 2.1.1 (Teorema di Karush-Kuhn-Tucker)**.** Se x^* è un punto regolare dei vincoli $h(x)$ e $g(x)$ ed è un punto di minimo locale di $f(x)$ sulla regione ammissibile $\Omega = \{x \in \mathbb{R}^n : h(x) = 0, g(x) \leq 0\}$, allora esistono dei moltiplicatori lagrangiani $\lambda^* \in \mathbb{R}^m$ e $\mu^* \in \mathbb{R}^p$ che verificano le seguenti condizioni di Karush-Kuhn-Tucker (KKT):

$$\begin{aligned} \nabla_x \mathcal{L}(x^*, \lambda^*, \mu^*) = \nabla f(x^*) + \nabla h(x^*)\lambda^* + \nabla g(x^*)\mu^* &= 0 \\ h(x^*) &= 0 \\ g(x^*) &\leq 0 \\ \mu^* &\geq 0 \\ g(x^*)^T \mu^* &= 0 \end{aligned}$$

La seconda e la terza equazione stanno a indicare che il punto x^* rispetta i vincoli e vengono perciò dette ***condizioni di ammissibilità***. L'ultima relazione viene invece detta ***condizione di complementarietà***. Essa può anche essere scritta in forma esplicita come segue:

$$\sum_{j=1}^{p} g_j(x^*)\mu_j^* = 0$$

Il seguente teorema rientra nella cosiddetta teoria della dualità. Infatti esso permette di associare ad un dato problema di ottimizzazione un suo duale che, sotto determinate ipotesi, può presentare stretti legami con il problema di partenza, ma allo stesso tempo, avere una struttura più favorevole.

Teorema 2.1.2 (Teorema di Wolfe)**.** Si consideri il problema primale

$$\begin{aligned} \min \quad & f(x) \\ \text{soggetto a} \quad & g(x) \leq 0 \end{aligned}$$

Se $f, g_j \ j = 1, ..., p$ sono convesse e di classe C^1 e se x^* è soluzione del problema primale e punto regolare dei vincoli, allora esiste un vettore $\lambda^* \in \mathbb{R}^p$ tale che (x^*, λ^*) è soluzione del problema duale nelle variabili x, λ

$$\begin{aligned} \max \quad & \mathcal{L}(x, \lambda) \\ \text{soggetto a} \quad & \begin{cases} \nabla_x \mathcal{L}(x, \lambda) = 0 \\ \lambda \geq 0 \end{cases} \end{aligned}$$

Inoltre il minimo del problema primale ed il massimo del problema duale coincidono, ovvero

$$f(x^*) = \mathcal{L}(x^*, \lambda^*) \tag{2.7}$$

2.1.3 Calcolo della lagrangiana e KKT

Tornando al problema di minimo a cui eravamo precedentemente arrivati (2.6), si nota che i vincoli sono delle funzioni affini e quindi qualificati. Inoltre la funzione obiettivo e i vincoli sono entrambi convessi e differenziabili. Di conseguenza le condizioni KKT sono valide e possono essere utilizzate per analizzare l'algoritmo SVM e derivare il problema duale, che verrà successivamente trattato.

Introduciamo quindi i cosiddetti moltiplicatori di Lagrange: $\alpha = (\alpha_1, ..., \alpha_m)^T$ e la funzione lagrangiana può essere definita come segue:

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{m} \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1] \tag{2.8}$$

Dalle condizioni KKT si ottiene invece il seguente sistema:

$$\begin{cases} \dfrac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \displaystyle\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i = 0 & \Rightarrow \quad \mathbf{w} = \textstyle\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i & \text{(2.9a)} \\ \dfrac{\partial \mathcal{L}}{\partial b} = -\displaystyle\sum_{i=1}^{m} \alpha_i y_i = 0 & \Rightarrow \quad \textstyle\sum_{i=1}^{m} \alpha_i y_i = 0 & \text{(2.9b)} \\ \forall i,\ \alpha_i \big[y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \big] = 0 & \Rightarrow \quad \alpha_i = 0 \vee y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1 & \text{(2.9c)} \end{cases}$$

Dall'equazione del gradiente (2.9a) si deduce che il vettore dei pesi è una combinazione lineare solo dei punti con $\alpha_i \neq 0$. Questi punti sono detti **support vectors** perchè definiscono il *maximum-margin hyperplane*. Ecco alcune proprietà importanti dei support vectors:

- i support vectors si trovano sugli iperpiani marginali $\mathbf{w} \cdot \mathbf{x}_i + b = \pm 1$;
- i punti che non sono support vectors ($\alpha_i = 0$) non influenzano la soluzione dell'SVM;
- l'iperpiano ottimale è unico, ma i support vectors possono variare;
- in uno spazio di dimensione N bastano $N + 1$ punti per definire un iperpiano, quindi se più di $N + 1$ punti si trovano su un iperpiano marginale, esistono più possibili scelte per i $N + 1$ support vectors che definiscono l'iperpiano a margine massimo.

2.1.4 Formulazione del problema duale

Per derivare la forma duale del problema di ottimizzazione si va a inserire all'interno della lagrangiana i risultati trovati grazie alle KKT. Si ottiene così:

$$\mathcal{L} = \underbrace{\frac{1}{2}||\sum_{i=1}^{m} \alpha_i y_i \mathbf{x}_i||^2 - \sum_{i=1}^{m} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)}_{-\frac{1}{2}\sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)} - \underbrace{\sum_{i=1}^{m} \alpha_i y_i b}_{0} + \sum_{i=1}^{m} \alpha_i$$

che semplificando diventa:

$$\mathcal{L} = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \tag{2.10}$$

Si ottiene quindi il seguente problema duale:

$$\begin{aligned} \max \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \\ \text{soggetto a} \quad & \begin{cases} \alpha_i \geq 0 \\ \sum_{i=1}^{m} \alpha_i y_i = 0, \ \forall i \in \{1, 2, ..., m\} \end{cases} \end{aligned}$$

2.1.5 Soluzione del problema duale

Si ottiene che la funzione obiettivo del problema duale risulta essere

$$G(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

Essa risulta essere una funzione infinitamente differenziabile e la sua matrice hessiana è data da: $\nabla^2 G(\alpha) = -A$, con $A = y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$ che è la matrice di Gram associata ai vettori $y_1\mathbf{x}_1, y_2\mathbf{x}_2, ..., y_m\mathbf{x}_m$. Essa è perciò semidefinita positiva, ossia $\nabla^2 G \preceq 0$ e $G(\alpha)$ è una funzione concava, il che implica che il problema di ottimizzazione è un problema do ottimizzazione convessa.

Inoltre poichè la funzione obiettivo è quadratica in α, il problema duale delle SVM è un **Quadratic Programming problem** (**QP problem**). Per risolverlo si possono usare sia dei QP solvers generici sia metodi specifici (che non saranno argomento del presente lavoro di tesi).

Poiché i vincoli del problema sono affini e convessi, la dualità forte è valida. Ossia il teorema di Wolfe permette di affermare con assoluta certezza che la soluzione del problema duale è equivalente alla soluzione del problema primale. In altre parole, il valore ottimo dei moltiplicatori di Lagrange α può essere usato per determinare direttamente il classificatore SVM:

$$h(x) = \text{sign}\left(\sum_{i=1}^{m} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{x} + b) \right) \tag{2.11}$$

Dal momento che i support vectors si trovano sugli iperpiani marginali, per ogni support vectors $\mathbf{x}_i$ deve valere:

$$\mathbf{w} \cdot \mathbf{x}_i + b = y_i$$

Da questa equazione si può ottenere b:

$$b = y_i - \sum_{j=1}^{m} \alpha_j y_j (\mathbf{x}_j \cdot \mathbf{x}_i) \tag{2.12}$$

Osservazione 2.1.1. Le equazioni del problema duale rivelano una proprietà fondamentale delle SVM: la soluzione dipende solo dai prodotti scalari tra i vettori, e non direttamente dai vettori stessi.

Questa osservazione è cruciale e verrà approfondita in seguito con l'introduzione dei metodi kernel, che permettono di estendere le SVM ai casi non linearmente separabili.

2.2 Caso non linearmente separabile

Nella maggior parte dei casi i punti presenti nel dataset S non risultano essere linearmente separabili, il che significa che non è possibile trovare un iperpiano $\mathbf{w} \cdot \mathbf{x} + b = 0$ che separi perfettamente tutte le istanze. Di conseguenza ci saranno alcuni punti $\mathbf{x}_i$ per cui si ha che $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \ngeq 1$.

Per gestire questa situazione è possibile andare ad utilizzare una versione rilassata del vincolo introducendo delle **slack varaiables** ξ_i, con $i = 1, 2, ..., m$. Esse misurano quanto ogni punto $\mathbf{x}_i$ viola il vincolo di separabilità. Si può scrivere così la nuova versione dei vincoli:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i, \text{ con } \xi_i \geq 0 \ \ \forall i \in \{1, 2, ..., m\} \tag{2.13}$$

Data questa definizione di slack variables possiamo effettuare la seguente classificazione:

- se $\xi_i = 0$, il punto è correttamente classificato fuori dalla zona di margine;
- se $0 < \xi_i < 1$, il punto è dentro il margine ma correttamente classificato;
- se $\xi_i > 1$, il punto non è correttamente classificato.

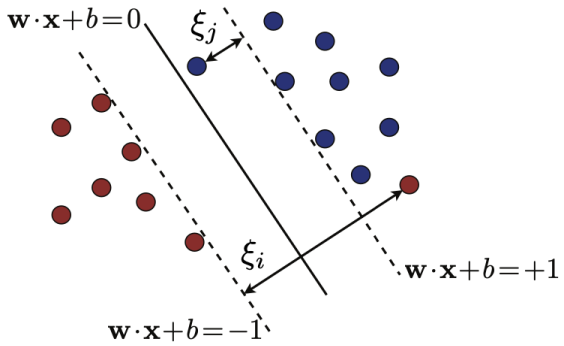

Figura 2.4: Rappresentazione grafica delle slack variables. Dato il grafico si ha sicuramente che $0 < \xi_j < 1$ dato che il punto è correttamente classificato, ma all'interno della zona di margine; mentre $\xi_i > 1$ visto che il punto è misclassificato.

2.2.1 Formulazione del problema primale

Le SVM con slack variables permettono di classificare correttamente i dati con margine ridotto, introducendo il concetto di *soft margin SVM*, in contrapposizione alle *hard margin SVM* (valide solo per dati separabili linearmente).

In questo contesto per scegliere l'iperpiano di separazione migliore si utilizza un compromesso tra:

- minimizzazione della penalizzazione introdotta dagli outlier ($\sum_{i=1}^{m} \xi_i$);
- massimizzazione del margine (che come avevamo visto precedentemente equivale alla minimizzazione di $||\mathbf{w}||^2$).

Il problema di ottimizzazione diventa dunque il seguente:

$$\min_{\mathbf{w},b,\xi} \quad \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{m} \xi_i \tag{2.14}$$

$$\text{soggetto a} \quad \begin{cases} y_i(\mathbf{w} \cdot \mathbf{x_i} + b) \geq 1 - \xi_i \\ \xi_i \geq 0, \ \forall i \in \{1, 2, ..., m\} \end{cases} \tag{2.15}$$

dove $\xi = (\xi_1, \xi_2, ..., \xi_m)^T$ e C è un iperparametro dell'algoritmo. Studiamo cosa succede al variare di C in $\mathbb{R}$:

- Se C è molto grande, allora maggiore peso verrà dato a

  $$\sum_{i=1}^{m} \xi_i$$

  e dunque maggiore sforzo sarà impiegato nel minimizzare l'errore: rischio di *overfitting*.

- Se C è molto piccolo, allora maggiore peso verrà dato a

  $$\frac{1}{2}||\mathbf{w}||^2$$

  e dunque maggiore sforzo sarà dedicato ad aumentare la *flatness*: rischio di consentire troppe osservazioni al di fuori dei margini.

2.2.2 Calcolo della lagrangiana e KKT

Come nel caso separabile, i vincoli del problema sono affini e convessi, il che garantisce che il problema soddisfi le condizioni KKT. Queste condizioni verranno utilizzate per analizzare l'algoritmo, dimostrare alcune proprietà chiave delle SVM e successivamente derivare il problema di ottimizzazione duale.

Per risolvere il problema di ottimizzazione, introduciamo le variabili di Lagrange $\alpha = (\alpha_1, ..., \alpha_m)^T$ associate ai vincoli di separabilità e $\beta = (\beta_1, ..., \beta_m)^T$ associate alle slack variables. A questo punto si trova che la lagrangiana del problema di ottimizzazione risulta essere:

$$\mathcal{L}(\mathbf{w}, b, \xi, \alpha, \beta) = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{m}\xi_i + \\ - \sum_{i=1}^{m}\alpha_i\left[y_i(\mathbf{w}\cdot\mathbf{x}+b) - 1 + \xi_i\right] - \sum_{i=1}^{m}\beta_i\xi_i$$

Andando a calcolare come prima le condizioni KKT si ottiene:

$$\begin{cases} \dfrac{\partial\mathcal{L}}{\partial\mathbf{w}} = \mathbf{w} - \sum_{i=1}^{m}\alpha_i y_i \mathbf{x}_i = 0 & \Rightarrow \quad \mathbf{w} = \sum_{i=1}^{m}\alpha_i y_i \mathbf{x}_i & \text{(2.16a)} \\ \dfrac{\partial\mathcal{L}}{\partial b} = -\sum_{i=1}^{m}\alpha_i y_i = 0 & \Rightarrow \quad \sum_{i=1}^{m}\alpha_i y_i = 0 & \text{(2.16b)} \\ \dfrac{\partial\mathcal{L}}{\partial\xi} = C - \alpha_i - \beta_i = 0 & \Rightarrow \quad \alpha_i + \beta_i = C & \text{(2.16c)} \\ \forall i,\ \alpha_i\big[y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1 + \xi_i\big] = 0 & \Rightarrow \quad \alpha_i = 0 \vee y_i(\mathbf{w}\cdot\mathbf{x}_i + b) = 1 - \xi_i & \text{(2.16d)} \\ \forall i,\ \beta_i\xi_i = 0 & \Rightarrow \quad \beta_i = 0 \vee \xi_i = 0 & \text{(2.16e)} \end{cases}$$

Come nel caso linearmente separabile si ha che il vettore dei pesi $\mathbf{w}$ è una combinazione lineare dei support vectors. Un punto $\mathbf{x}_i$ è un support vector se e solo se $\alpha_i \neq 0$. Di conseguenza possono esistere due diversi tipi di support vectors:

- Se $0 < \alpha_i < C$ allora dalla (2.16c) si ha che $\beta_i \neq 0$ e quindi dalla (2.16e) $\xi_i = 0$. Cioè il support vector si trova esattamente sul margine.
- Se $\alpha_i = 0$ allora dalla (2.16c) si ha che $\beta_i = 0$ e dalla (2.16e) che $\xi_i > 0$, cioè il support vector si trova dentro al margine oppure è misclassificato.

Osserviamo perciò che il vettore $\mathbf{w}$ è unico, ma la scelta dei support vectors non lo è, poiché più punti possono trovarsi sui margini.

2.2.3 Formulazione del problema duale

Come nel caso linearmente separabile si va a sostituire $\mathbf{w}$ dalla (2.16a) all'interno della lagrangiana:

$$\mathcal{L}(\alpha) = \underbrace{\frac{1}{2}||\sum_{i=1}^{m}\alpha_i y_i \mathbf{x}_i||^2 - \sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j(\mathbf{x}_i\cdot\mathbf{x}_j)}_{-\frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j(\mathbf{x}_i\cdot\mathbf{x}_j)} - \underbrace{\sum_{i=1}^{m}\alpha_i y_i b}_{0} + \sum_{i=1}^{m}\alpha_i$$

che semplificando diventa:

$$\mathcal{L}(\alpha) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y_i y_j(\mathbf{x}_i\cdot\mathbf{x}_j) \tag{2.17}$$

Questo porta alla formulazione duale del problema di ottimizzazione SVM con soft margin:

$$\begin{aligned} \max \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \\ \text{soggetto a} \quad & \begin{cases} 0 \leq \alpha_i \leq C \\ \sum_{i=1}^{m} \alpha_i y_i = 0, \ \forall i \in \{1, 2, ..., m\} \end{cases} \end{aligned} \tag{2.18}$$

Osservazione 2.2.1. Differenza rispetto al caso separabile:

- il vincolo $0 \leq \alpha_i \leq C$ limita il contributo di ogni support vector;
- per $C \longrightarrow +\infty$ il problema diventa *hard margin*.

2.2.4 Soluzione del problema duale

Le osservazioni effettuate precedentemente nel caso dell'*hard margin* continuano a valere anche nel caso del *soft margin*. In particolare, la funzione obiettivo è concava e infinitamente differenziabile ed è equivalente a un problema QP convesso. Il problema è equivalente al problema primale.

Dunque la soluzione di (2.18) può essere usata direttamente per determinare il classificatore:

$$h(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b) = \text{sign}\left(\sum_{i=1}^{m} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{x}) + b \right)$$

Al calcolo del bias b partecipano solo i support vector sul margine, cioè punti con $0 < \alpha_i < C$. Per questi punti vale la condizione $\mathbf{w} \cdot \mathbf{x}_i + b = y_i$, da cui:

$$b = y_i - \sum_{j=1}^{m} \alpha_j y_j (\mathbf{x}_j \cdot \mathbf{x}_i) \tag{2.19}$$

L'osservazione 2.1.1 continua a valere di conseguenza anche nel presente caso.

2.2.5 Kernel methods

I metodi kernel (kernel methods) sono tecniche molto diffuse nel campo del machine learning. Permettono di estendere algoritmi come le SVM per definire frontiere di decisione non lineari. In generale, qualunque algoritmo che dipenda solo sui prodotti scalari tra coppie di esempi (come si era visto in 2.1.1) può essere modificato inserendovi un kernel, con effetti simili a quelli che si ottengono sulle SVM.

L'idea chiave è la sostituzione del normale prodotto scalare nello spazio di input $\mathbb{X}$ con un prodotto scalare definito in uno spazio di dimensione (anche molto) maggiore, detto feature space $\mathbb{H}$. Questo viene fatto attraverso una funzione kernel K, che rispetta determinate condizioni di positività (Mercer's condition).

$$K(\mathbf{x}, \mathbf{x}') = \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}') \rangle$$

Dove $\Phi : \mathbb{X} \longrightarrow \mathbb{H}$ è una mappa (anche non esplicitamente nota) in uno spazio di Hilbert e K gioca il ruolo di misura di similarità tra $\mathbf{x}$ e $\mathbf{x}'$. In questo modo, un algoritmo che è lineare nello spazio di input $\mathbb{X}$ può essere trasformato, attraverso l'uso di un kernel, in un algoritmo che è lineare nello spazio delle feature $\mathbb{H}$, ma che rappresenta una soluzione non lineare nello spazio originale $\mathbb{X}$.

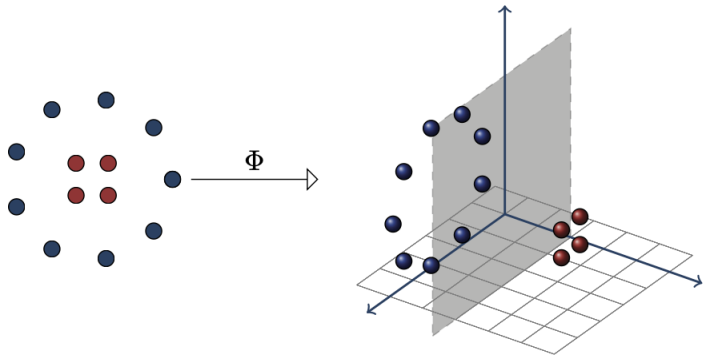

Figura 2.5: Esempio di una mappatura non lineare da uno spazio bidimensionale a uno spazio tridimensionale, in cui il problema diventa linearmente separabile.

Vantaggi dei kernel methods:

- Non linearità: Permettono di separare dati non separabili linearmente nello spazio originale, trasformandoli in uno spazio ad alta (o infinita) dimensione dove la separazione diventa (potenzialmente) lineare.
- Efficienza: invece di calcolare la mappatura $\Phi(\mathbf{x})$ esplicitamente (cosa spesso proibitiva poiché lo spazio $\mathbb{H}$ può essere enorme), si calcola direttamente $K(\mathbf{x}, \mathbf{x}')$ in tempo tecnicamente molto inferiore. Questo è possibile grazie a 2.1.1.
- Flessibilità: non occorre esplicitare la forma di Φ. E' sufficiente che K soddisfi delle condizioni di positività che vedremo in seguito per garantire buone proprietà di apprendimento.

Definizione 2.2.1. Un kernel $K : \mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{R}$ è detto *simmetrico definito positivo* se per ogni $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\} \subseteq \mathcal{X}$, la matrice $\mathbf{K} = [K(\mathbf{x}_i, \mathbf{x}_j)]_{ij} \in \mathbb{R}^{m \times m}$ è *simmetrica semidefinita positiva.*

In particolare diciamo che la matrice $\mathbf{K}$ è simmetrica e semidefinita positiva se è simmetrica e vale almeno una delle due seguenti affermazioni:

- gli autovalori di $\mathbf{K}$ sono non-negativi;

- per ogni vettore $\mathbf{c} = (c_1, c_2, ..., c_m)^T \in \mathbb{R}^{m\times 1}$,

$$\mathbf{c}^T\mathbf{K}\mathbf{c} = \sum_{i,j=1}^{m} c_i c_j K(\mathbf{x}_i, \mathbf{x}_j) \geq 0.$$

K è detta anche *matrice kernel* (*kernel matrix*) o la *matrice di Gram* associata a K e al dataset S. Quindi diremo che la matrice kernel associata a un kernel definito positivo è una matrice semidefinita positiva.

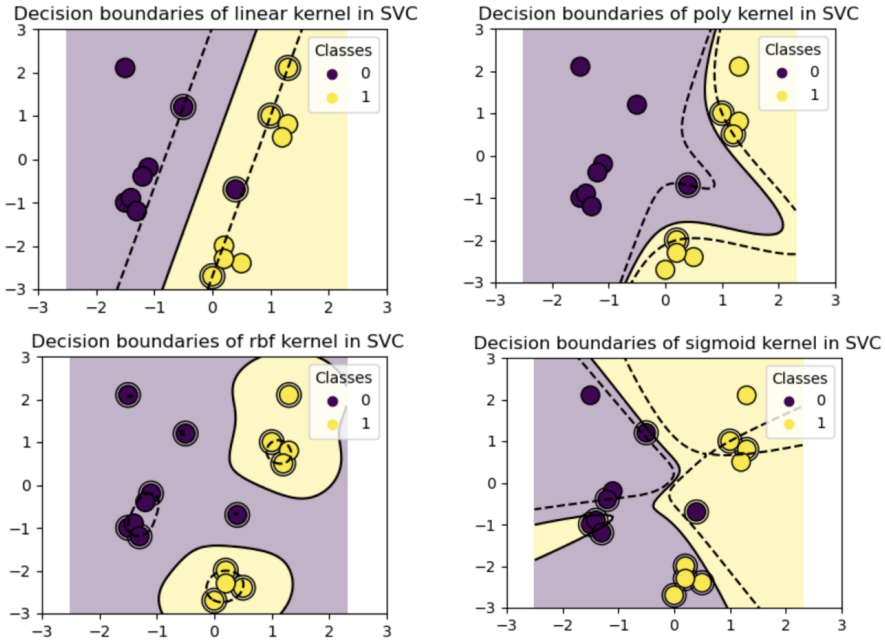

Figura 2.6: Esempio di diverse mappature: lineare, polinomiale, gaussiana e sigmoidea.

Si possono ora riportare alcuni esempi di kernel molto usati nell'ambito del machine learning e in particolare con le SVM:

- Kernel polinomiale:

$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^N, \quad K(\mathbf{x}, \mathbf{x}') = (\mathbf{x} \cdot \mathbf{x}' + c)^d$$

  dove c è una costante positiva e $d \in \mathbb{N}$ è il grado del kernel. Questa tipologia di kernel mappa i dati dell'input space in uno spazio a più alta dimensionalità. In particolare si ha che la dimensionalità del feature space è dato dal numero di monomi di grado fino a d in N variabili, cioè:

$$\dim(\mathbb{H}) = \binom{N+d}{d} = \frac{(N+d)!}{d! \cdot N!}$$

- Kernel gaussiano:

$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^N, \quad K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\sigma^2}\right)$$

dove σ è una costante positiva. Questi sono i tipi di kernel usati più frequentemente dal momento che sono in grado di mappare i dati dell'input space in un feature space di dimensionalità infinita, come si può facilmente vedere andando a scrivere l'espressione esplicita del kernel usando lo sviluppo in serie dell'esponenziale:

$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^N, \quad K(\mathbf{x}, \mathbf{x}') = \sum_{n=0}^{+\infty} \frac{(\mathbf{x} \cdot \mathbf{x}')^n}{\sigma^{2n} n!}$$

- Kernel sigmoideo:

$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^N, \quad K(\mathbf{x}, \mathbf{x}') = \tanh(a(\mathbf{x} \cdot \mathbf{x}') + b)$$

dove $a, b \geq 0$. L'utilizzo dei kernel sigmoidi con le SVM porta a un algoritmo strettamente correlato agli algoritmi di apprendimento basati su reti neurali semplici, che sono anch'essi spesso definiti tramite una funzione sigmoide.

Nella figura 2.6 si può trovare un esempio di applicazione su un caso reale dei diversi kernel che sono appena stati illustrati.

2.3 Metodi di discesa del gradiente

Si è visto che per poter addestrare le SVM, sia nel caso linearmente sia nel caso non linearmente separabile, è necessario risolvere un problema di ottimizzazione. Solitamente si utilizzano *metodi di programmazione quadratica*, come il *Sequential Minimal Optimization*. Tuttavia, nel caso di SVM lineari su dataset di grandi dimensioni, la programmazione quadratica diventa computazionalmente costosa. Per questo motivo, si preferisce utilizzare tecniche basate sulla discesa del gradiente stocastica (SGD).

I metodi di discesa del gradiente sono dei metodi iterativi che, a partire da un punto $x^{(0)} \in \mathbb{R}^n$ generano una successione di vettori $\{x^{(k)}\}_{k \in \mathbb{N}} \subset \mathbb{R}^n$. Essi hanno la proprietà:

$$f(x^{(k+1)}) < f(x^{(k)}), \quad k = 0, 1, 2, ...$$

garantendo dunque che la funzione obiettivo decresca iterativamente ad ogni passo.

In particolare nel metodo di *steepest descent* i vettori $x^{(k)}$ vengono scelti con la seguente regola di aggiornamento:

$$x^{(k+1)} = x^{(k)} - \eta_k \nabla f(x^{(k)})$$

dove:

- $\eta_k \in \mathbb{R}^+$ è detto *steplength* o *learning rate*;
- $\nabla f(x^{(k)})$ è il gradiente della funzione obiettivo da ottimizzare calcolato nel punto $x^{(k)}$.

Nel seguito verrà utilizzato il metodo di discesa del gradiente stocastico per la risoluzione del problema di ottimizzazione primale delle SVM:

$$\min_{w,b} \lambda||w||^2 + \sum_{i=1}^{n} \max(0, 1 - y_i(w \cdot x_i - b)) \tag{2.20}$$

Di conseguenza le regole di aggiornamento per i parametri w e b sono le seguenti:

$$w \leftarrow w - \eta(2\lambda w - x_i y_i), \quad b \leftarrow b - \eta y_i$$

La discesa del gradiente è particolarmente utile per SVM lineari su dataset di grandi dimensioni, mentre per SVM con kernel l'ottimizzazione avviene in uno spazio trasformato, rendendo meno praticabile l'approccio SGD. Per questo motivo, i modelli kernelizzati utilizzano soluzioni basate su metodi duali, come l'algoritmo SMO.

Nel caso specifico della SVM implementata in questa tesi, è stato utilizzato SGD per eseguire la minimizzazione iterativa della funzione obiettivo, rendendo il modello più efficiente in termini computazionali.

Capitolo 3

Ensemble Methods: Adaboost

Uno dei più popolari game show trasmessi in oltre 70 paesi dal 1999 ha attirato milioni di telespettatori grazie all'enorme montepremi e a un'innovativa meccanica di aiuti chiamata *"lifelines"*. I concorrenti potevano:

1. eliminare due risposte errate;
2. telefonare a un amico;
3. chiedere l'aiuto del pubblico.

Questa struttura ha suscitato l'interesse di ricercatori di statistica e intelligenza artificiale, i quali si sono chiesti quale fosse la strategia migliore tra le tre. Essi riuscirono a dimostrare che la scelta migliore era la terza, facendo così valere un principio fondamentale anche nel mondo dell'intelligenza artificiale: combinare più fonti di decisione indipendenti può migliorare significativamente l'accuratezza complessiva. Questo principio è alla base dell'ensemble learning, una tecnica in cui più classificatori vengono combinati per migliorare la robustezza e l'affidabilità delle predizioni.

Nella vita quotidiana, consultiamo più esperti per prendere decisioni migliori, come chiedere opinioni a più medici, leggere recensioni prima di un acquisto o richiedere referenze prima di un'assunzione. Questo stesso principio viene applicato nei sistemi di apprendimento automatico, dove più classificatori vengono combinati per migliorare le prestazioni rispetto a un singolo modello, in particolare in termini di robustezza e di affidabilità.

L'**ensemble learning** è quindi una tecnica che combina più modelli per migliorare la precisione delle decisioni rispetto a un singolo classificatore. Le ragioni principali per utilizzarlo sono 5.

Motivazioni statistiche

E' già stato detto che ottenere buone prestazioni sui dati di addestramento non garantisce necessariamente una buona capacità di generalizzazione su dati mai visti in precedenza. Infatti è possibile che due classificatori con prestazioni simili sull'insieme di training abbiano capacità di generalizzazione molto diverse. Inoltre, anche classificatori con capacità di generalizzazione simili

possono comportarsi in modo molto diverso quando vengono testati su dati reali, specialmente se il dataset utilizzato per misurare le loro prestazioni non è sufficientemente rappresentativo della realtà. L'uso di ensemble methods riduce il rischio di affidarsi a un singolo classificatore con prestazioni subottimali, poiché la combinazione delle loro previsioni mitiga gli errori individuali e stabilizza le decisioni finali. Anche se l'ensemble non supera necessariamente il classificatore migliore all'interno del gruppo, esso riduce in modo significativo la possibilità di selezionare un modello con un comportamento particolarmente scarso.

Questo è lo stesso principio che seguiamo nella vita quotidiana quando chiediamo più pareri: ad esempio, se diversi medici concordano su una diagnosi, ci fidiamo di più rispetto al parere di un singolo dottore, il quale potrebbe avere un'esperienza atipica rispetto alla norma.

Gestione di grandi volumi di dati

In alcune applicazioni, la quantità di dati da analizzare può essere troppo grande per essere gestita in modo efficiente da un singolo classificatore, sia per limiti computazionali che per il tempo necessario. Una soluzione più efficiente consiste nel suddividere i dati in sottoinsiemi più piccoli, addestrare diversi classificatori su partizioni diverse del dataset, e infine combinare i loro output attraverso una regola di aggregazione intelligente.

Questo approccio ottimizza l'elaborazione, distribuendo il carico computazionale tra più modelli e permettendo di gestire dataset di dimensioni molto grandi senza compromettere la qualità delle previsioni.

Gestione di dati limitati

I sistemi di ensemble learning possono essere utilizzati anche per affrontare il problema opposto, ovvero la scarsità di dati. In questi casi si possono applicare tecniche di campionamento (resampling) per generare sottoinsiemi casuali sovrapposti dei dati disponibili. Ogni sottoinsieme può essere utilizzato per addestrare un classificatore diverso, dando origine a un sistema di ensemble.

Questi metodi si sono dimostrati molto efficaci, poiché permettono di massimizzare l'uso delle informazioni disponibili e di ridurre il rischio di overfitting, migliorando la capacità di generalizzazione del modello.

Approccio *divide-et-impera*

Indipendentemente dalla quantità di dati disponibili, alcuni problemi di classificazione possono essere troppo complessi per un singolo classificatore. In particolare, il confine decisionale che separa le classi potrebbe essere troppo complicato o non appartenere allo spazio delle funzioni che il modello scelto è in grado di apprendere.

Un esempio tipico è un problema di classificazione bidimensionale con due classi e un confine decisionale non lineare. Se si utilizza un classificatore lineare, questo non sarà in grado di apprendere un confine di separazione complesso.

Tuttavia, combinando più classificatori lineari in un ensemble, è possibile approssimare e apprendere anche un confine decisionale non lineare.

Seguendo la strategia divide-et-impera, l'algoritmo scompone un problema complesso in sotto-problemi più semplici, ciascuno risolto da un classificatore specializzato. Questa suddivisione consente di apprendere strutture più intricate e di ottenere una modellazione più accurata del problema generale.

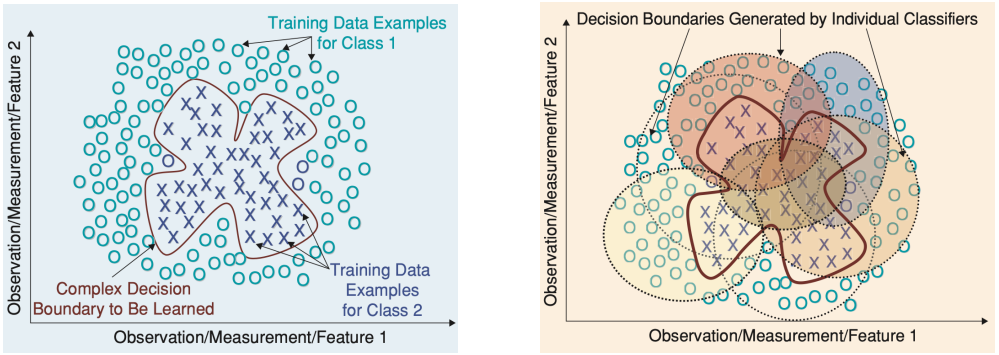

Figura 3.1: A sinistra un dataset con decision boundary impossibile per imparare da un solo classificatore circolare; a destra il risultato della combinazione di più classificatori circolari.

Fusione dei dati

L'uso degli ensemble di classificatori è particolarmente utile nei casi in cui i dati provengano da fonti diverse e contengano caratteristiche eterogenee. In queste situazioni, un singolo classificatore non è in grado di apprendere e interpretare tutte le informazioni contenute nei diversi tipi di dati.

Un esempio pratico è la diagnosi di un disturbo neurologico, in cui un neurologo può richiedere test diversi, come una risonanza magnetica (MRI), un elettroencefalogramma (EEG) e analisi del sangue. Ogni test fornisce dati con caratteristiche differenti, sia nel numero che nella tipologia delle variabili, rendendo impossibile l'uso di un unico classificatore per elaborare tutte le informazioni.

In questi casi, la soluzione migliore è allenare classificatori separati su ciascun tipo di dati e poi combinare i loro output.

3.1 Tipologie di Ensemble: selezione vs fusione

I diversi approcci agli ensemble di classificatori si distinguono per il modo in cui vengono generati i singoli classificatori e per la strategia con cui i loro output vengono combinati. Le due principali metodologie sono la selezione dei classificatori (classifier selection) e la fusione dei classificatori (classifier fusion).

Nella *selezione dei classificatori*, ogni classificatore viene addestrato per specializzarsi su una particolare area dello spazio delle caratteristiche. Quando viene effettuata una nuova predizione, il sistema seleziona il classificatore che è

stato addestrato con dati più vicini al punto da classificare, basandosi su una metrica di distanza. A volte, più classificatori possono essere selezionati per prendere una decisione congiunta.

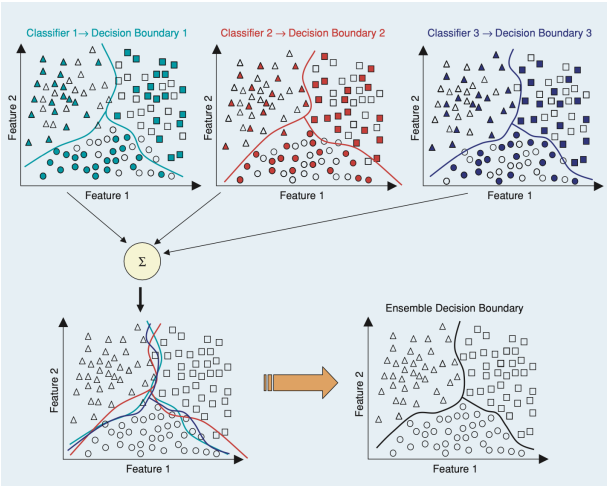

Figura 3.2: Combinazione di classificatori addestrati su diversi sottoinsiemi dei dati di addestramento.

Nella *fusione dei classificatori*, invece, tutti i modelli vengono addestrati sull'intero spazio delle caratteristiche e i loro risultati vengono combinati per produrre una decisione finale. Questo approccio consente di trasformare un insieme di classificatori deboli in un singolo classificatore forte, migliorando la precisione complessiva del sistema. Tecniche come il bagging e il boosting appartengono a questa categoria.

La fusione può avvenire in diversi modi:

- sulle etichette di classificazione (per esempio voto di maggioranza tra i classificatori);
- sui valori di output continui generati dai classificatori, che possono essere normalizzati nell'intervallo [0,1] e interpretati come probabilità a posteriori. In questo caso vengono applicate regole matematiche di combinazione, come la somma, il prodotto, la media, il massimo/minimo ecc...

3.2 Importanza della diversità

Se esistesse un classificatore con prestazioni di generalizzazione perfette, non ci sarebbe bisogno di sistemi ensemble. Tuttavia, nella realtà, la presenza di rumore nei dati, outlier e distribuzioni complesse rende impossibile ottenere un modello perfetto. Per questo motivo, l'approccio migliore è creare più classificatori e combinare i loro output in modo da migliorare le prestazioni rispetto a un singolo modello.

Affinché un ensemble funzioni efficacemente, è fondamentale che i singoli classificatori commettano errori diversi tra loro. L'idea alla base di questa

strategia è simile a quella del filtro passa-basso nel trattamento del rumore: se ogni modello fa errori differenti, una combinazione strategica dei loro output può ridurre l'errore complessivo del sistema.

Per ottenere questa diversità, si possono seguire diverse strategie.

- **Utilizzo di dataset diversi per l'addestramento dei classificatori**

  Si può generare diversità campionando sottoinsiemi casuali del dataset di training. Nel *Bagging* (**b**ootstrap **agg**regat**ing**) per esempio si creano insiemi di dati sovrapposti, addestrando i classificatori con differenti porzioni di dati. Se il campionamento viene effettuato senza ripetizione si parla di *k-fold cross validation*: il dataset viene suddiviso in k blocchi e ogni classificatore viene addestrato su $k-1$ blocchi diversi.

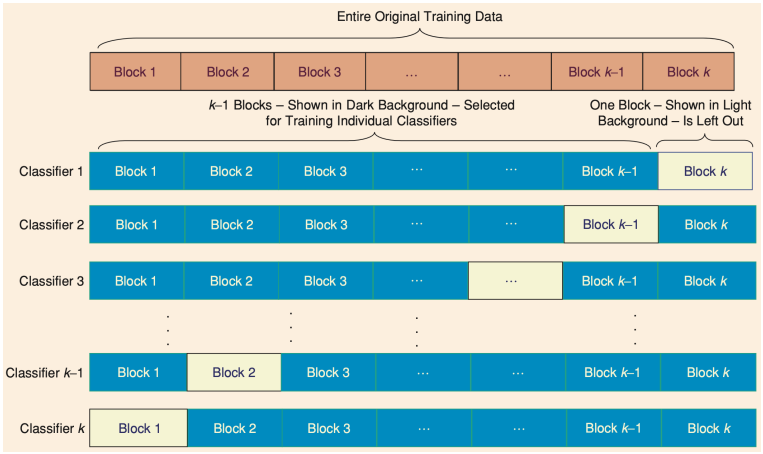

Figura 3.3: Divisione dei dati in k-fold per generare dataset di addestramento diversi ma sovrapposti.

- **Variazione dei parametri di addestramento.**

  Anche addestrando più modelli sugli stessi dati si può ottenere diversità cambiando i parametri interni.

- **Uso di diversi modelli di classificazione**

  Invece di addestrare più classificatori dello stesso tipo si possono combinare modelli completamente differenti, come reti neurali, alberi decisionali, SVM, k nearest neighbour, ecc...

  Tuttavia, l'uso di modelli eterogenei è più comune in applicazioni specifiche, dove la combinazione di tecnologie diverse porta vantaggi concreti.

- **Selezione di sottoinsiemi di features**

  In alcuni scenari la diversità può essere ottenuta variando le caratteristiche utilizzate dai classificatori. Una tecnica nota è il random subspace method, in cui si addestrano diversi classificatori utilizzando sottoinsiemi casuali delle feature disponibili. Questo metodo è particolarmente efficace in applicazioni in cui il numero di caratteristiche è elevato.

La diversità di cui si sta parlando si traduce in termini formali nell'indipendenza probabilistica.

Per convincersi della potenza di questa tipologia di metodi si consideri il seguente esempio.

Supponiamo di avere un ensemble di 5 classificatori indipendenti, ciascuno con un'accuratezza del 75%. Se prendiamo una decisione con voto di maggioranza, il sistema sarà errato solo se almeno 3 classificatori su 5 sbagliano contemporaneamente. La probabilità di questo evento può essere modellata come una distribuzione binomiale:

$$\mathbb{P}(x \leq 2) = \sum_{i=0}^{2} \binom{5}{i} 0.75^i 0.25^{5-i} = 0.105$$

Siamo così riusciti a diminuire di gran lunga l'error rate del nostro classificatore usandone solo 5.

3.3 Boosting

Nel 1990 [8], Schapire dimostrò che un **weak learner**, ossia un algoritmo in grado di generare classificatori che si comportano solo leggermente meglio di una scelta casuale, può essere trasformato in un **strong learner**, ovvero un classificatore in grado di classificare correttamente quasi tutte le istanze, con un margine di errore arbitrariamente piccolo. Le definizioni formali di weak learner e strong learner si possono trovare nel paper originale di Schapire [9].

Inoltre nello stesso articolo l'autore presenta anche un elegante algoritmo per migliorare le prestazioni di un weak learner fino al livello di uno strong learner. Questo metodo è il boosting, ed è oggi considerato uno degli sviluppi più importanti nella storia recente del machine learning.

3.3.1 Algoritmo

L'algoritmo [8] prevede la creazione di un ensemble di classificatori mediante il campionamento ripetuto dei dati di addestramento, combinando poi le previsioni attraverso il voto di maggioranza. La strategia di campionamento è progettata per fornire il set di dati più informativo per ciascun classificatore successivo.

1. Il primo classificatore C_1 viene addestrato su un sottoinsieme casuale dei dati di training disponibili.

2. Il secondo classificatore C_2 viene addestrato su un sottoinsieme più informativo, composto per metà da istanze classificate correttamente da C_1 e per metà da istanze erroneamente classificate da C_1.

3. Il terzo classificatore C_3 viene addestrato su istanze in cui C_1 e C_2 disaccordano.

Alla fine i tre classificatori vengono combinati con una votazione a tre vie, creando un ensemble più accurato.

Schapire dimostrò che l'errore di addestramento dell'ensemble di tre classificatori è limitato superiormente da

$$g(\varepsilon) < 3\varepsilon^2 - 2\varepsilon^3$$

dove ε è l'errore di ciascun classificatore, a condizione che ogni classificatore abbia un tasso di errore inferiore a 0.5, che rappresenta la soglia minima per un modello binario, equivalente al tirare a caso. In particolare Schapire riuscì a dimostrare che l'errore dell'ensemble dei tre classificatori è sempre minore dell'errore del miglior classificatore (sempre a condizione che ciascun classificatore sia un weak learner).

Infine, un classificatore forte può essere ottenuto applicando ricorsivamente il boosting su più iterazioni.

3.4 Adaboost

Nel 1997, Freund e Schapire introdussero **AdaBoost**, un algoritmo di boosting che ha ricevuto un'attenzione straordinaria nel campo dell'intelligenza artificiale e dell'apprendimento automatico. AdaBoost è una versione più generale del boosting originale e ha diverse varianti, tra cui AdaBoost.M1, utilizzato per problemi di classificazione multiclasse, e AdaBoost.R, adatto per problemi di regressione. Nel seguito ci si concentrerà solo sul problema di classificazione.

AdaBoost genera un insieme di ipotesi e le combina attraverso un voto a maggioranza ponderato delle classi predette dalle singole ipotesi. Le ipotesi sono generate addestrando un classificatore debole, utilizzando istanze estratte da una distribuzione iterativamente aggiornata dei dati di addestramento. Questo aggiornamento della distribuzione garantisce che le istanze classificate erroneamente dal classificatore precedente abbiano una maggiore probabilità di essere incluse nei dati di addestramento del classificatore successivo. Pertanto, i dati di addestramento dei classificatori consecutivi si concentrano su istanze sempre più difficili da classificare.

L'algoritmo mantiene una distribuzione di pesi $\mathcal{D}_t(i)$ assegnata a ogni istanza di training $\mathbf{x}_i$. All'inizio la distribuzione è uniforme, quindi tutte le istanze hanno la stessa probabilità di essere selezionate per il primo classificatore. Il tasso di errore ε_t del classificatore h_t viene calcolato come la somma dei pesi delle istanze classificate erroneamente. Questo errore viene poi normalizzato in un valore che serve poi come fattore di aggiornamento per la distribuzione dei pesi.

La regola di aggiornamento segue due principi chiave:

1. Le istanze classificate correttamente vedono il loro peso ridotto.
2. Le istanze classificate erroneamente vedono il loro peso aumentato, aumentando la probabilità di essere scelte nella generazione del prossimo classificatore.

Perciò si può affermare che AdaBoost si concentra iterativamente sulle istanze più difficili.

Una volta che AdaBoost ha generato i T classificatori, a differenza del boosting tradizionale, utilizza un metodo di voto a maggioranza ponderato. In questo modo, i classificatori che hanno avuto migliori prestazioni durante l'addestramento ricevono un peso maggiore nelle decisioni finali.

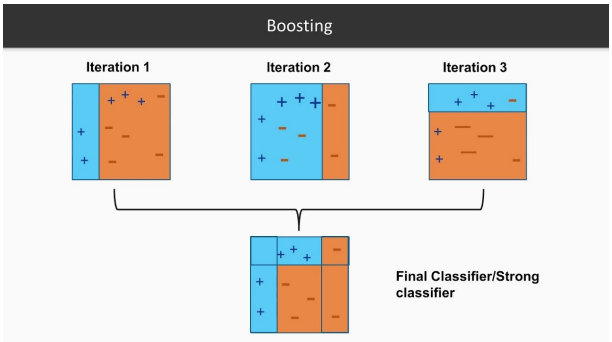

Figura 3.4: Esempio di un algoritmo AdaBoost su tre iterazioni

3.4.1 Algoritmo

Si consideri il caso di classificazione binaria [10], ossia $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ con $y_i \in \{-1, 1\}$. Si inizializzano tutti i pesi in modo che la distribuzione sia uniforme e successivamente inizia il processo iterativo per T cicli.

Ad ogni ciclo viene addestrato un classificatore h_t utilizzando la distribuzione $\mathcal{D}_t$ e si calcola l'errore

$$\epsilon_t = \mathbb{P}_{i \sim \mathcal{D}_t}[h_t(\mathbf{x}_i) \neq y_i] = \sum_{i:\, h_t(\mathbf{x}_i) \neq y_i} \mathcal{D}_t(i)$$

In questo modo il peso del classificatore sarà

$$\alpha_t = \frac{1}{2} \log\left(\frac{1 - \epsilon_t}{\epsilon_t}\right)$$

I nuovi pesi delle istanze, ancora non normalizzati, saranno perciò:

$$w_{t+1}^{(i)} = \begin{cases} w_t^{(i)} e^{-\alpha_t} & \text{se } h_t(\mathbf{x}_i) = y_i \\ w_t^{(i)} e^{\alpha_t} & \text{se } h_t(\mathbf{x}_i) \neq y_i \end{cases}$$

Per garantire che i pesi aggiornati $\mathcal{D}_{t+1}(i) = \mathcal{D}_t(i) e^{-\alpha_t y_i h_t(\mathbf{x}_i)}$ formino una distribuzione valida, la costante di normalizzazione Z_t deve soddisfare:

$$\sum_{i=1}^{m} \frac{\mathcal{D}_t(i) e^{-\alpha_t y_i h_t(\mathbf{x}_i)}}{Z_t} = 1 \implies Z_t = \sum_{i=1}^{m} \mathcal{D}_t(i) e^{-\alpha_t y_i h_t(\mathbf{x}_i)}$$

Quindi:

$$Z_t = \sum_{i:\, h_t(\mathbf{x}_i) = y_i} \mathcal{D}_t(i) e^{-\alpha_t} + \sum_{i:\, h_t(\mathbf{x}_i) \neq y_i} \mathcal{D}_t(i) e^{\alpha_t}$$

Poiché $\sum_{i:\, h_t(\mathbf{x}_i)=y_i} \mathcal{D}_t(i) = 1 - \epsilon_t$ e $\sum_{i:\, h_t(\mathbf{x}_i)\neq y_i} \mathcal{D}_t(i) = \epsilon_t$ si ottiene:

$$Z_t = (1-\epsilon_t)e^{-\alpha_t} + \epsilon_t e^{\alpha_t}$$

Sostituendo al posto di α_t il valore precedentemente calcolato si ottiene:

$$\begin{aligned} Z_t &= (1-\epsilon_t)e^{-\frac{1}{2}\log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)} + \epsilon_t e^{\frac{1}{2}\log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)} \\ &= (1-\epsilon_t)\sqrt{\frac{\epsilon_t}{1-\epsilon_t}} + \epsilon_t\sqrt{\frac{\epsilon_t}{1-\epsilon_t}} = 2\sqrt{\epsilon_t(1-\epsilon_t)} \end{aligned}$$

Una volta aggiornata la distribuzione termina il processo iterativo e si avrà che l'output del classificatore finale deve essere:

$$f(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}_t)$$

Algorithm 1 AdaBoost per la classificazione binaria

Require: $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$ con $y_i \in \{-1, 1\}$
Require: H (insieme di classificatori deboli), T (numero di iterazioni)

1: **Inizializzazione**: $\mathcal{D}_1(i) \leftarrow \frac{1}{m}, \quad \forall i \in \{1, \ldots, m\}$
2: **for** $t \leftarrow 1$ to T **do**
3: Addestra il classificatore debole $h_t \in H$ utilizzando la distribuzione $\mathcal{D}_t$
4: Calcola l'errore:

$$\epsilon_t = \mathbb{P}_{i\sim\mathcal{D}_t}[h_t(\mathbf{x}_i) \neq y_i]$$

5: Calcola il peso del classificatore:

$$\alpha_t = \frac{1}{2}\log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

6: Calcola il fattore di normalizzazione:

$$Z_t = 2\sqrt{\epsilon_t(1-\epsilon_t)}$$

7: **for** $i \leftarrow 1$ to m **do**
8: Aggiorna la distribuzione:

$$\mathcal{D}_{t+1}(i) \leftarrow \frac{\mathcal{D}_t(i)e^{-\alpha_t y_i h_t(x_i)}}{Z_t}$$

9: **end for**
10: **end for**
11: **Output del classificatore finale**:

$$f \leftarrow \sum_{t=1}^{T} \alpha_t h_t(x)$$

return f

Capitolo 4

Analisi predittiva delle malattie cardiovascolari

Le malattie cardiovascolari rappresentano la prima causa di morte a livello globale, causando ogni anno circa 17,9 milioni di decessi, pari al 31% del totale delle morti nel mondo. Quattro decessi su cinque dovuti a malattie cardiovascolari sono causati da infarti e ictus, e un terzo di questi avviene prematuramente in persone di età inferiore ai 70 anni. L'insufficienza cardiaca è una delle principali conseguenze delle malattie cardiovascolari, e il dataset che si utilizzerà nel seguito contiene 11 caratteristiche che possono essere utilizzate per prevedere una possibile malattia cardiaca.

Le persone affette da malattie cardiovascolari o che presentano un alto rischio cardiovascolare (a causa della presenza di uno o più fattori di rischio, come ipertensione, diabete o una patologia già diagnosticata) necessitano di una diagnosi precoce e di una gestione adeguata. In questo contesto, i modelli di machine learning possono offrire un valido supporto, aiutando a identificare i pazienti a rischio e migliorando l'efficacia degli interventi medici.

Il dataset è composto da 917 istanze, ognuna delle quali è descritta dalle seguenti 12 caratteristiche.

- Age: età del paziente espressa in anni.
- Sex: genere del paziente, codificato in M (male, maschio) e F (female, femmina).
- ChestPainType: tipologia di dolore toracico, suddiviso in 4 categorie:
  - TA (Typical Angina): angina tipica;
  - ATA (Atypical Angina): angina atipica;
  - NAP (Non-Anginal Pain): dolore non anginoso;
  - ASY (Asymptomatic): asintomatico.
- RestingBP: pressione arteriosa a riposo, espressa in mmHg.
- Cholesterol: livello di colesterolo nel sangue.

- FastingBS: livello di glicemia a digiuno, variabile binaria che assume valore 1 se la glicemia è superiore a 120mg/dl e 0 altrimenti.
- RestingECG: risultati dell'elettrocardiogramma a riposo, classificati in:
  - Normal: esame nella norma;
  - ST: anomalie dell'onda ST-T;
  - LVH: ipertrofia ventricolare sinistra secondo i criteri di Estes.
- MaxHR: frequenza cardiaca massima raggiunta durante un test da sforzo, con valori compresi tra 60 e 202.
- ExerciseAngina: presenza di angina indotta da esercizio fisico, variabile categorica che può assumere valori Y se presente e N se assente.
- Oldpeak: valore numerico che misura la depressione del segmento ST rispetto alla linea di base.
- ST_Slope: pendenza del tratto ST durante il test da sforzo suddivisa in:
  - Up: in salita;
  - Flat: piatta;
  - Down: in discesa.
- HeartDisease: classe predire che indica la presenza della malattia cardiaca e assume valore 1 in caso di presenza e 0 altrimenti.

L'obiettivo dell'analisi è confrontare le prestazioni di SVM e AdaBoost utilizzando le metriche discusse in precedenza.

Verranno innanzitutto presentate le implementazioni delle classi dei due modelli, concentrandosi su una versione di SVM senza kernel non lineari. Poiché l'implementazione da zero di questi algoritmi può risultare computazionalmente onerosa e meno ottimizzata rispetto alle librerie standard, si utilizzerà anche la versione ottimizzata di SVM fornita dalla libreria **Scikit-Learn**, che garantisce efficienza e conformità agli standard più moderni di ricerca.

Una volta definiti i modelli, è fondamentale garantire che i dati siano adeguatamente preparati per l'addestramento. Per questo motivo sarà necessario effettuare il **preprocessing**, una fase che include:

- Gestione dei dati mancanti: identificazione e trattamento dei dati mancanti, se presenti.
- Gestione delle feature categoriche: conversione delle variabili categoriche in formato numerico tramite tecniche come *one-hot encoding* o *ordinal encoding*.
- Normalizzazione delle feature: trasformazione delle variabili numeriche per garantire che abbiano scale comparabili, utilizzando tecniche di standardizzazione o min-max scaling.

Infine, verranno effettuate le valutazioni delle performance, confrontando SVM e AdaBoost sulla base delle metriche di classificazione presentate nei capitoli precedenti.

4.1 Implementazione SVM

Verrà creata la classe SVM e saranno implementati i metodi `__init__`, `fit`, `decision_function`, `predict_proba` e `predict` [14].

Metodo `__init__`

Il metodo `__init__` è il metodo costruttore della classe. Prende in input i seguenti parametri:

- `learning_rate`: rappresenta lo steplength per l'ottimizzazione tramite discesa del gradiente e se non inizializzato assume il valore di 0.001.
- `lambda_param`: coincide con $\frac{1}{C}$, dove C era il parametro di regolarizzazione nella formulazione standard delle SVM. Esso regolava il trade-off tra margine massimo e minimizzazione degli errori di classificazione. Se non inizializzato assume il valore di 0.01.
- `n_iters`: numero di iterazioni che vengono eseguite per la soluzione del problema di ottimizzazione.

All'interno del metodo costruttore vengono quindi inizializzati i parametri della classe SVM, che sono `self.lr`, `self.lp` e `self.n_iters` e vengono posti uguali ai valori passati alla funzione. I parametri della funzione `self.w` e `self.b` vengono inizializzati a `None` e il parametro `self.model_name` viene inizializzato a `SVM`.

```
1 def __init__(self, learning_rate = 0.001,
2               lambda_param = 0.01, n_iters = 1000):
3     self.lr = learning_rate
4     self.lp = lambda_param
5     self.n_iters = n_iters
6     self.w = None
7     self.b = None
8     self.model_name = 'SVM'
```

Metodo `fit`

Il metodo `fit` si occupa del processo di training del modello. La funzione prende in input i parametri:

- `X`: matrice contenente le istanze del dataset;
- `y`: vettore contenente le etichette delle istanze.

All'interno del metodo vengono innanzitutto memorizzate le dimensioni della matrice di input nelle variabili `n_samples` e `n_features`, dove la prima variabile tiene traccia del numero di righe, mentre la seconda del numero di colonne.

Successivamente si inizializzano i parametri del modello `self.w` e `self.b` a 0 e si entra solo dopo nel ciclo iterativo di addestramento.

Per ogni campione `x_i` del dataset `X`, il modello verifica se il campione è correttamente classificato e si trova al di fuori del margine attraverso la condizione memorizzata nella variabile booleana `condition`:

- se è vera il modello aggiorna solo i pesi per minimizzare il termine di regolarizzazione $\lambda||w||^2$ attraverso la formula

  $$\mathbf{w} \leftarrow \mathbf{w} - lr \cdot 2\lambda\mathbf{w}$$

  Il termine $2\lambda w$ deriva dalla regolarizzazione $\lambda||w||^2$, che penalizza pesi grandi per evitare overfitting.

- se è falsa il modello aggiorna sia i pesi `self.w` che il bias `self.b` per minimizzare sia il termine di regolarizzazione che l'errore di classificazione. Le formule sono:

  $$\mathbf{w} \leftarrow \mathbf{w} - lr \cdot (2\lambda\mathbf{w} - \mathbf{x}_i y_i)$$

  $$b \leftarrow b - lr \cdot y_i$$

  Il termine $-\mathbf{x}_i y_i$ deriva dalla slack variables $\xi_i = \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i - b))$ che penalizza i campioni classificati erroneamente o che violano il margine.

Una volta completato l'addestramento il modello è in grado di restituire un valore continuo tramite la funzione di decisione `decision_function()` che rappresenta la distanza del campione dal piano separatore: $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b$.

Affinché il modello possa fornire probabilità affidabili, viene applicata la tecnica nota come *platt scaling*, che consiste nell'addestrare un modello di regressione logistica sui punteggi della funzione di decisione. Il modello di regressione logistica apprende una funzione sigmoidea per convertire i valori continui della funzione di decisione in probabilità:

$$\mathbb{P}(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-f(\mathbf{x})}}.$$

Nel codice, il processo avviene nel seguente modo:

1. Si calcolano i punteggi della funzione di decisione tramite il metodo `decision_function()`.

2. Si addestra un modello di regressione logistica su questi punteggi per ottenere probabilità calibrate.

```
def fit(self, X, y):
    n_samples, n_features = X.shape

    self.w = np.zeros(n_features)
    self.b = 0
    decision_scores = []

    for _ in range(self.n_iters):
        for idx, x_i in enumerate(X):
            condition = y[idx] * (np.dot(x_i, self.w)
                - self.b) >= 1

            if condition:
                self.w -= self.lr * (2 * self.lp *
                   self.w)
            else:
                self.w -= self.lr * (2 * self.lp *
                   self.w - x_i * y[idx])
                self.b -= self.lr * y[idx]

    decision_scores = self.decision_function(X)
    self.platt_model = LogisticRegression()
    self.platt_model.fit(decision_scores.reshape(-1,
       1), y)
```

Metodo `decision_function()`

Il metodo `decision_function()` calcola i punteggi della funzione di decisione per ciascun campione di input. Questo valore rappresenta la distanza dal piano separatore e indica quanto un punto appartenga a una classe piuttosto che all'altra. La funzione di decisione, come detto precedentemente è $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b$.

Maggiore è il valore assoluto di $f(\mathbf{x})$, più sicuro è il modello della classificazione. Se il valore è positivo, il campione è classificato come +1, altrimenti -1.

```
    def decision_function(self, X):
        return np.dot(X, self.w) - self.b
```

Metodo `predict_proba()`

Il metodo `predict_proba()` trasforma i punteggi della funzione di decisione in probabilità calibrate utilizzando il metodo platt scaling, ossia una regressione logistica sui punteggi $f(x)$

La probabilità di appartenere alla classe positiva (+1) è data dalla sigmoide come visto poco prima:

$$\mathbb{P}(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-f(\mathbf{x})}}.$$

Questo permette di ottenere un valore compreso tra 0 e 1, interpretato come la confidenza del modello nella classificazione.

Nel codice, per evitare di implementare manualmente il Platt Scaling, viene utilizzata la classe `LogisticRegression` di `sklearn`

```
def predict_proba(self, X):
        if self.platt_model is None:
            raise ValueError("Platt scaling non e'
               stato addestrato. Chiama fit() prima")

        decision_scores = self.decision_function(X)
        probabilities = self.platt_model.
           predict_proba(decision_scores.reshape(-1,
           1))
        return probabilities
```

Metodo `predict`

Il metodo `predict` è responsabile per effettuare previsioni su un insieme di dati `X` che gli viene passato in input utilizzando il metodo `decision_function` precedentemente descritto.

In output viene utilizzata la funzione `np.sign()` che restituisce il segno (± 1) del valore della funzione di decisione per ogni istanza dell'insieme `X`. Questo segno indica la classe prevista per ogni campione:

- Se il valore è positivo, il campione viene classificato come appartenente alla classe +1.
- Se è negativo, il campione viene classificato come appartenente alla classe -1.

```
def predict(self, X):
    return np.sign(self.decision_function(X))
```

4.2 Implementazione AdaBoost

E' stata implementata la classe AdaBoost e i metodi creati sono `__init__`, `fit`, `decision_function`, `predict_proba` e `predict` [15].

Metodo `__init__`

E' il metodo costruttore e prende in input come unico parametro `n_estimators`, ossia il numero di stimatori che si vogliono utilizzare nell'algoritmo. Tale valore viene memorizzato nel parametro `self.n_estimators` e vengono inizializzati a vuoto:

- `self.alphas`: una lista che memorizza i pesi di ciascun modello debole;
- `self.models`: una lista che memorizza i modelli deboli addestrati.

Viene invece inizializzato a `AdaBoost` il parametro `self.model_name`.

```
1 def __init__(self, n_estimators):
2     self.n_estimators = n_estimators
3     self.alphas = []
4     self.models = []
5     self.model_name = 'AdaBoost'
```

Metodo `fit`

Il metodo `fit` addestra l'ensemble sui dati che vengono forniti in input, cioè `X` (matriche contenente l'insieme di istanze) e `y` (vettore contenente l'etichetta di ogni istanza).

Vengono memorizzate le dimensioni della matrice `X` nelle variabili `n_samples` e `n_features` e poi vengono inizializzati i pesi di ogni campione in modo uniforme.

Si passa poi al ciclo di addestramento.

1. Viene creato un modello debole (un *albero decisionale* con profondità massima pari a 1, ossia un modello molto semplice che tra tutte le features a disposizione ne sceglie solo una e trova una soglia ottimale per dividere i dati in due gruppi).

2. Tale modello viene addestrato sui dati `X` e `y`, utilizzando i pesi `w` per dare più importanza ai campioni con peso maggiore.

3. Vengono memorizzate nella variabile `predictions` le previsioni dei dati di training, fatte attraverso il comando `model.predict`, fornito dalla classe `DecisionTreeClassifier` della libreria `sklearn.tree`.

4. Viene calcolato l'errore `err` come la somma pesata dei campioni classificati erroneamente, divisa per la somma totale dei pesi. Esso misura quanto è "debole" il modello corrente.

5. Il peso `alpha` del modello viene calcolato come si era visto precedentemente con la formule
$$\frac{1}{2}\log\left(\frac{1-\text{err}}{\text{err}}\right)$$
In particolare al denominatore viene aggiunta una quantità trascurabile e pari a 10^{-10} per evitare problemi di divisione per 0.

6. I pesi dei campioni vengono aggiornati attraverso la regola vista in precedenza:
$$\begin{cases} w_t^{(i)} e^{-\alpha_t} & \text{se } h_t(\mathbf{x}_i) = y_i \\ w_t^{(i)} e^{\alpha_t} & \text{se } h_t(\mathbf{x}_i) \neq y_i \end{cases}$$
e vengono poi normalizzati a 1. Dopo ogni iterazione, i campioni classificati erroneamente ricevono un peso maggiore, mentre quelli classificati correttamente vedono il loro peso ridursi. Questo permette ai successivi alberi di concentrarsi sugli esempi più difficili.

7. Il modello e il peso appena trovati vengono aggiunti alla due liste rispettive inizializzate prima.

```
1  def fit(self, X, y):
2      n_samples, n_features = X.shape
3      w = np.ones(n_samples) / n_samples
4
5      for _ in range(self.n_estimators):
6          model = DecisionTreeClassifier(max_depth=1)
7          model.fit(X,y,sample_weight = w)
8          predictions = model.predict(X)
9          err = np.sum(w*(predictions!=y)) / np.sum(w)
10         alpha = 0.5 * np.log((1-err) / (err+1e-10))
11         self.alphas.append(alpha)
12         self.models.append(model)
13         w = w * np.exp(-alpha * y * predictions)
14         w = w / np.sum(w)
```

Metodo `decision_function()`

Il metodo `decision_function()` permette di calolare il punteggio della funzione di decisione per ciascun campione nel dataset `X` che le viene passato come parametro. Il valore continuo che viene restituito in output rappresenta il grado di appartenenza di ogni istanza alla classe positiva o negativa.

Viene inizializzato il vettore `F` con valori nulli, che avrà la stessa lunghezza del numero di campioni in `X`. Successivamente, per ogni modello addestrato (`learner`) e per ogni peso associato (`alpha`), il metodo va ad aggiungere a `F` il prodotto di ciascun `alpha` per la previsione restituita dal modello.

Viene restituito infine il valore di `F`.

```
def decision_function(self, X):
        F = np.zeros(X.shape[0])
        for alpha, learner in zip(self.alphas, self.
           models):
            F += alpha * learner.predict(X)
        return F
```

Metodo `predict_proba()`

Il metodo `predict_proba` permette di calcolare le probabilità di appartenenza di ciascun campione alle due classi.

Il metodo inizia calcolandosi il punteggio `F` per ogni campione del dataset. Esso rappresenta la distanza di ogni campione dal'iperpiano di separazione dell'algoritmo.

Successivamente si calcola la probabilità che ogni campione appartenga alla classe 1 con la funzione:

$$\mathbb{P}(y = 1|X) = \frac{1}{1 + e^{-2F}} \quad \text{e} \quad \mathbb{P}(y = -1|X) = 1 - \mathbb{P}(y = 1|X)$$

Il fattore -2 nella funzione esponenziale deriva da una forma scalata della sigmoide, utilizzata in alcuni modelli boosting per migliorare la separabilità delle classi.

Il metodo restituisce una matrice formata da due colonna e tante righe quanti sono i campioni del test set. La prima colonna contiene le probabilità che ciascuna istanza appartenga alla prima classe, mentre la seconda la probabilità che appartenga all'altra.

```
def predict_proba(self, X):
    F = self.decision_function(X)
    proba_class_1 = 1 / (1 + np.exp(-2 * F))
    proba_class_0 = 1 - proba_class_1
    return np.vstack([proba_class_0, proba_class_1]).
       T
```

Metodo `predict`

Il metodo `predict` in questo caso è del tutto uguale a quello della classe SVM: si richiama la funzione `decision_function` passandogli come parametro le istanze `X` passate alla funzione `predict`. Infine viene utilizzata la funzione `np.sign` vista precedentemente.

```
def predict(self, X):
    strong_preds = self.decision_function(X)
    return np.sign(strong_preds).astype(int)
```

4.3 Preprocessing dei dati

Prima di addestrare i modelli appena descritti è necessaria una corretta preparazione dei dati. Infatti come abbiamo visto poco fa il dataset presenta sia variabili numeriche sia variabili categoriche, che devono essere opportunamente trattate per garantire il corretto funzionamento dei modelli.

Innanzitutto è necessario leggere il file csv per importare i dati nello spazio di lavoro. Per farlo si utilizza la funzione `read_csv()` della libreria `Pandas`. Successivamente i dati vengono divisi in due variabili: `X`, contenente tutte le feature senza l'etichetta, e `y` che memorizza le etichette corrispondenti, opportunamente modificate in ± 1.

```
df = pd.read_csv("heart.csv")
y = df['HeartDisease']
y = np.where(y <= 0, -1, 1)
X = df.drop(columns = 'HeartDisease')
```

In questo modo si riesce a visualizzare il numero di istanze mancanti per ogni feature.

Successivamente la fase di preprocessing si suddivide nei seguenti passaggi.

4.3.1 Verifica della presenza di valori mancanti

Per verificare la presenza di dati mancanti, si utilizza il metodo `isna()`, che restituisce un valore booleano per ogni cella. Successivamente, applicando `sum()`, si ottiene il numero di valori mancanti per ogni feature.

```
print(df.isna().sum())
```

Dall'output capiamo che non ci sono valori mancanti:

```
Age              0
Sex              0
ChestPainType    0
RestingBP        0
Cholesterol      0
FastingBS        0
RestingECG       0
MaxHR            0
ExerciseAngina   0
Oldpeak          0
ST_Slope         0
HeartDisease     0
```

4.3.2 Conversione delle variabili categoriche

Il dataset utilizzato contiene sia variabili numeriche che categoriche. Poiché i modelli di machine learning, come SVM e AdaBoost, lavorano esclusivamente con dati numerici, è necessario trasformare le variabili categoriche in un formato adatto.

Le variabili categoriche presenti nel dataset sono le seguenti:

- Sex: {M, F};
- ChestPainType: {TA, ATA, NAP, ASY};
- RestingECG: {Normal, ST, LVH};
- ExerciseAngina: {Y, N};
- ST_Slope: {Up, Flat, Down}.

Per la loro trasformazione si vanno ad adottare diverse strategie a seconda della loro natura.

Label encoding

Il label encoding [11] è una tecnica che assegna un numero intero univoco a ciascuna categoria presente nei dati, rendendolo particolarmente adatto per modelli di machine learning che operano con input numerici.

Viene solitamente usato quando si devono trattare variabili categoriche binarie e il modello deve interpretarle come etichette numeriche non ordinali. Nel caso in questione è quindi adatto per le features `Sex` e `ExerciseAngina`.

Per implementarlo in Python si può utilizzare il metodo `LabelEncoder()` della classe `preprocessing` di `sklearn`.

```
1 X['Sex'] = LabelEncoder().fit_transform(X['Sex'])
2 X['ExerciseAngina'] = LabelEncoder().fit_transform(X[
    'ExerciseAngina'])
```

One-hot encoding

Il one-hot encoding [12] è un metodo per convertire le variabili categoriche in formato binario. Questo processo crea nuove colonne per ogni categoria della variabile, dove 1 indica la presenza, mentre 0 la sua assenza.

Nel caso in questione sarà usato per le features `ChestPainType` e `RestingECG`.

Viene utilizzato per due diversi motivi:

- eliminazione dell'ordine esistente: molte variabili categoriche non hanno un ordine intrinseco e se si assegnassero valori numerici, il modello potrebbe interpretarli erroneamente come un rapporto gerarchico;

- miglioramento delle prestazione del modello: questo metodo permette di catturare relazioni più complesse nei dati, che potrebbero invece essere trascurate se i dati venissero trattati come un'entità numerica.

Per implementarlo in Python è possibile utilizzare la funzione `get_dummies` di `Pandas`:

```
X = pd.get_dummies(X, columns = one_hot_features,
    drop_first = True, dtype = float)
```

Ordinal encoder

Questo metodo [13] è particolarmente utile nei casi in cui le categorie hanno un ordine naturale, come i voti scolastici (A, B, C) o i livelli di intensità (Basso, Medio, Alto).

Nel caso in questione risulta molto utile per la feature `ST_Slope`.

Per implementarlo in Python è possibile usare la funzione `OrdinalEncoder` della classe `preprocessing` di `sklearn`. A tale funzione vanno date in input le categorie ordinate in modo crescente.

```
ordinal_encoder = OrdinalEncoder(categories=[['Up', '
    Flat', 'Down']])
X[ordinal_features] = ordinal_encoder.fit_transform(X
    [ordinal_features])
```

4.3.3 Standardizzazione delle variabili numeriche

L'unità di misura utilizzata può influenzare l'analisi dei dati [16]. Ad esempio, cambiare l'unità di misura dell'altezza da metri a pollici, o del peso da chilogrammi a libbre, può portare a risultati molto diversi. Per evitare la dipendenza dalla scelta delle unità di misura, i dati dovrebbero essere normalizzati o standardizzati. Questo implica trasformare i dati per rientrare in un intervallo più piccolo o comune, come $[-1, 1]$ o $[0, 1]$.

La normalizzazione dei dati tenta di dare a tutti gli attributi un peso uguale. Essa risulta particolarmente utile per gli algoritmi di classificazione che coinvolgono reti neurali o misurazioni di distanza, come l'SVM. Infatti in questo modo si riesce a prevenire che attributi con intervlli inizialmente ampi (come può essere ad esempio il reddito annuo di una persona) prevalgano su attributi con intervalli inizialmente più piccoli (ad esempio attributi binari).

Nel caso in questione si utilizzerà il metodo `StandardScaler` fornito dalla classe `preprocessing` di `sklearn`. Esso va a calcolare il nuovo valore nel seguente modo:

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

dove μ è la media dei valori e σ è la loro deviazione standard.

Nel caso in questione è utile andare ad effettuare tale standardizzazione per tutte le feature numeriche, quindi:

- `Age` (età del paziente)
- `RestingBP` (pressione sanguigna a riposo)
- `Cholesterol` (colesterolo nel sangue)
- `MaxHR` (frequenza cardiaca massima)
- `Oldpeak` (depressione ST).

Per effettuare tali operazioni in un solo comando si utilizza un `ColumnTransformer`, un'utility della libreria Scikit-Learn che consente di applicare trasformazioni differenti a diverse colonne di un dataset in modo efficiente e strutturato.

```
1 coltran_svm = ColumnTransformer([('ss',
     StandardScaler(), numerical_features)])
2 X_svm = coltran_svm.fit_transform(X)
```

Il nuovo dataset in cui vengono salvati i dati standardizzati è stato chiamato `X_svm` dal momento che è stato notato che le prestazioni di AdaBoost su questo nuovo dataset risultano peggiori rispetto all'altro. Questo comportamento può essere dovuto al fatto che AdaBoost è un algoritmo basato su alberi decisionali, che non è sensibile alla scala delle feature, a differenza di SVM che utilizza misure di distanza.

4.3.4 Analisi della distribuzione delle classi

Prima di andare ad effettuare il training dei modelli è necessario verificare che entrambe le classi siano abbastanza rappresentate nell'intero dataset [16]. Questa verifica preliminare è fondamentale poiché la presenza di uno sbilanciamento potrebbe influenzare negativamente le prestazioni del modello, portandolo a favorire la classe maggioritaria. Per fare ciò si è analizzata la distribuzione delle classi attraverso un istogramma.

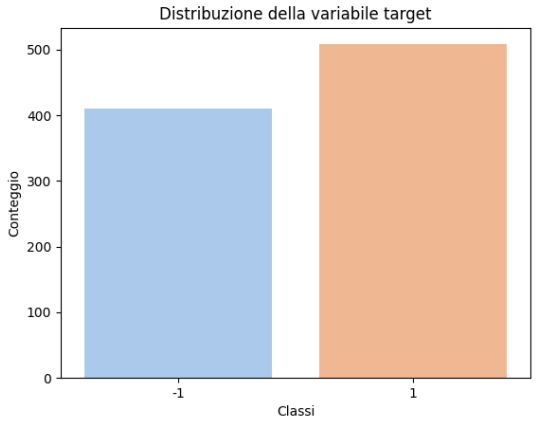

Figura 4.1: Distribuzione della variabile `HeartDisease`

Dai risultati si osserva che la classe 1 è rappresentata nel 55% delle istanze, mentre la classe 0 copre il restante 45%.

Poiché la distribuzione delle classi è relativamente bilanciata, non è necessario applicare tecniche di riequilibrio dei dati. Qualora il dataset fosse risultato significativamente sbilanciato, si sarebbero potute applicare tecniche di riequilibrio come oversampling della classe minoritaria, undersampling della classe dominante o metodi basati su generazione di nuovi campioni sintetici come SMOTE.

4.3.5 Suddivisione del dataset in train e test set

Per valutare le prestazioni dei modelli il dataset viene suddiviso in due insiemi:

- train set (75%): utilizzato per l'addestramento del modello;
- test set (25%): utilizzato per valutare le performance su dati mai visti prima.

Si è inoltre scelto di mantenere la stessa distribuzione delle classi nei due insiemi utilizzando la stratificazione. Questo garantisce che la proporzione tra le classi rimanga inalterata.

In Python è stato utilizzato il metodo `train_test_split` della classe `model_selection` della libreria `sklearn`.

```
X_train_ada, X_test_ada, y_train_ada, y_test_ada =
train_test_split(X, y, test_size = 0.25,
                 stratify = y, random_state=42)
X_train_svm, X_test_svm, y_train_svm, y_test_svm =
train_test_split(X_svm, y, test_size = 0.25,
                 stratify = y, random_state=42)
```

Sono state create due suddivisioni diverse del dataset perché le operazioni di standardizzazione delle variabili numeriche erano servite solo per l'addestramento delle SVM.

I parametri usati (oltre agli insiemi delle istanze su cui era stato effettuato il preprocessing) sono:

- `test_size = 0.25`: la dimensione dell'insieme di test è il 25% del dataset originale;
- `stratify = y`: permette di mantenere inalterata la distribuzione delle classi nei due insiemi;
- `random_state = 42`: garantisce la riproducibilità dei risultati.

Possiamo anche verificare che la distribuzione sia rimasta inalterata:

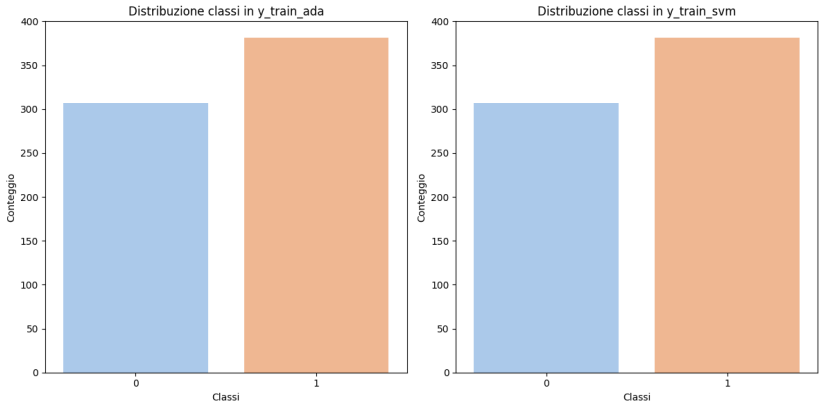

Figura 4.2: Distribuzione delle classi da predire nel train e nel test set

4.4 Addestramento dei modelli

Dopo aver effettuato il preprocessing dei dati e la suddivisione in training e test set, si procede con l'addestramento dei modelli di machine learning. In questa fase vengono considerati tre approcci distinti:

1. AdaBoost con 50 weak learner;
2. SVM personalizzato con ottimizzazione tramite discesa del gradiente;
3. SVM con kernel gaussiano dalla libreria `sklearn`.

```
1 adaboost = AdaBoost(50)
2 svm = SVM(0.0001, 0.1, 1000)
3 adaboost.fit(X_train_ada, y_train_ada)
4 svm.fit(X_train_svm, y_train_svm)
```

Quindi il modello AdaBoost viene addestrato con 50 weak learner, mentre l'SVM personalizzato è implementato utilizzando un learning rate pari a 0.0001, un lambda parameter pari a 0.1 e un numero di iterazioni di 1000.

Successivamente è stato implementato un modello SVM basato su kernel RBF (*radial basis function*). Per ottimizzare le prestazioni si applica la ricerca degli iperparametri tramite `GridSearchCV` della classe `model_selection` di `sklearn` per testare vari valori di C.

```
svc = SVC(kernel = 'rbf', gamma = 'scale', C = 1)
grid = GridSearchCV(svc, {'C':[1.05,1.1,1.2]})
grid.fit(X_train_svm, y_train_svm)
```

`GridSearchCV` cerca il miglior valore che deve assumere C tra quelli elencati per massimizzare le prestazioni del modello sui dati di training. Il miglior modello sarà quello utilizzato per predire i valori sui dati di test.

4.5 Confronto tra i modelli

L'obiettivo è quello di confrontare le prestazioni dei modelli che abbiamo precedentemente introdotto utilizzando il database presentato nelle sezioni antecedenti.

Per valutare i modelli ci si è serviti delle classiche metriche utilizzate per la classificazione, quindi:

- *accuratezza, precisione, recall* e *F1-score* per comprendere l'efficacia dei modelli nella classificazione;
- *matrici di confusione* per analizzare la distribuzione degli errori di classificazione;
- *curve ROC* e *valori AUC* per valutare la capacità discriminante dei modelli.

4.5.1 Confronto basato sulle metriche

Per il calcolo delle metriche sopra elencate è stata implementata la funzione `evaluate_model` che prende in input le previsioni effettuate dai modelli e le etichette reali, oltre che il nome del modello per effettuare la stampa a video.

Tale funzione si serve dei metodi:

- `accuracy_score`: per il calcolo dell'accuratezza;
- `precision_score`: per il calcolo della precisione;
- `recall_score`: per il calcolo della recall;
- `f1_score`: per il calcolo dell'F1.

Essi sono forniti dalla classe `metrics` della libreria `sklearn`. I risultati che si ottengono sono i seguenti:

Modello	Accuracy	Precision	Recall	F1
AdaBoost	0.8826	0.8968	0.8898	0.8933
SVM	0.7913	0.8015	0.8268	0.8140
SVC	0.8217	0.8707	0.7953	0.8313

Si può quindi facilmente osservare che il modello che offre le prestazioni più bilanciate e solide è l'AdaBoost. Ha i valori più alti in tutte le metriche considerate, indicando così che risulta essere l'algoritmo più affidabile e generale tra i tre, in grado cioè di gestire bene sia le istanze positive, sia quelle negative. In particolare, la sua capacità di adattarsi ai dati attraverso il meccanismo di apprendimento iterativo permette di ridurre al minimo sia i falsi positivi che i falsi negativi.

SVM ha valori inferiori in tutte le metriche rispetto a AdaBoost, ma rimangono comunque rispettabili. La recall è più alta rispetto a SVC, indicando che è in grado di identificare meglio le istanze positive.

SVC ha un valore di precision più alto rispetto a SVM. Ciò significa che è molto preciso nell'identificare le istanze positive, riducendo il numero di falsi positivi. Tuttavia, ha un valore di recall più basso, indicando che fatica a catturare tutte le istanze positive, generando un numero più elevato di falsi negativi.

4.5.2 Analisi delle matrici di confusione

Grazie alle matrici di confusione è possibile andare a verificare se ciò che è appena stato inferito dalle metriche è veritiero oppure no. Procediamo dunque con la loro lettura, grazie ai metodi `confusion_matrix` e `ConfusionMatrixDisplay`.

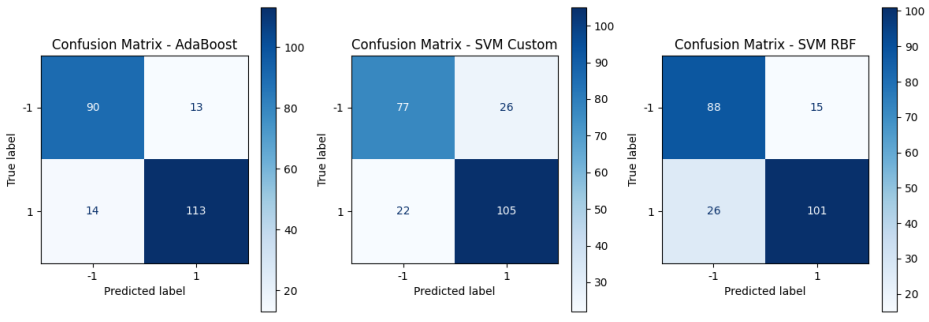

Figura 4.3: Matrici di confusione dei tre modelli

AdaBoost ha il numero più basso di falsi positivi (FP) e falsi negativi (FN), il che spiega i suoi alti valori di precision e recall. Questo modello è quindi il più affidabile e bilanciato.

L'SVM con kernel lineare ha un numero più alto di FP e FN rispetto ad AdaBoost, il che si riflette in valori più bassi di precision e recall. Questo modello è meno preciso ma ha una migliore capacità di identificare le istanze positive rispetto a SVC.

L'SVM con kernel gaussiano, invece, ha un numero di FP molto basso, simile ad AdaBoost, ma un numero più alto di FN. Ciò spiega la sua alta precision (pochi FP) ma recall più bassa (più FN). Questo modello è ideale quando è cruciale minimizzare i falsi positivi, anche a costo di perdere alcune istanze positive.

4.5.3 Analisi delle curve ROC e valori AUC

Come già visto le curve ROC permettono di visualizzare le capacità dei modelli di distinguere tra le due classi, mentre il valore AUC misura l'area sotto la curva, fornendo un'indicazione complessiva della qualità del classificatore.

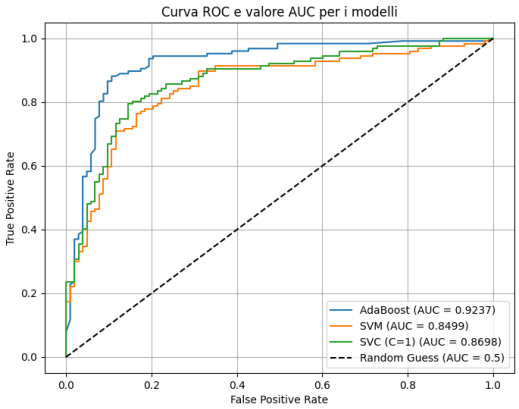

Figura 4.4: Curve ROC e valori AUC dei tre modelli e del random guess

Nel grafico è possibile trovare le curve ROC dei tre modelli considerati. La linea tratteggiata rappresenta la bisettrice del primo quadrante, che corrisponde a un classificatore casuale (AUC = 0.5). Un modello con una curva ROC sopra questa linea ha prestazioni migliori del caso casuale.

Tale grafico è stato realizzato grazie ai metodi `roc_curve` e `auc` forniti dalla classe `metrics` della libreria `sklearn`.

- AdaBoost si conferma essere il modello con la migliore capacità discriminativa tra le classi essendo caratterizzato dalla curva ROC più alta e dal valore AUC più elevato. Esso è perciò in grado di mantenere un alto tasso di TPR (True Positive Rate) e un basso tasso di FPR (False Positive Rate).

- Il modello SVM con kernel lineare ha il valore AUC più basso tra i tre modelli. La sua curva ROC è la meno performante, posizionandosi sotto rispetto alle altre due per quasi tutti i valori di False Positive Rate.

- SVM con kernel gaussiano ha un valore di AUC intermedio tra i tre, nonostante dalla curva ROC si possa osservare che le sue performance risultano essere paragonabili a quelle di SVM con kernel lineare.

In sintesi, AdaBoost è il modello con le migliori prestazioni, seguito da SVM con kernel gaussiano e SVM con kernel lineare, che mostra le performance più basse.

Capitolo 5

Conclusioni

In conclusione il presente lavoro ha mostrato come le Support Vector Machine e i metodi ensemble, in particolare AdaBoost, possano costituire metodi validi e complementari per affrontare problemi di classificazione in ambito medico. Dal confronto sul dataset preso in considerazione si è osservato che AdaBoost è risultato essere il più robusto nel gestire le situazioni con dati parzialmente rumorosi, mentre le SVM si confermano efficienti quando ben ottimizzate in termini di iperparametri.

I risultati sono promettenti, ma il dataset limitato (meno di 1000 istanze considerate) potrebbe non catturare appieno le variabilità presenti in contesti reali. Inoltre il tuning manuale degli iperparametri rappresenta una criticità notevole, dal momento che richiede tempo e competenze specifiche per ottenere migliori prestazioni.

Potrebbe perciò rappresentare uno sviluppo futuro l'utilizzo di tali metodi su dataset più ampi per valutare la capacità di generalizzazione dei modelli in scenari più complessi e variabili. In aggiunta, l'implementazione di metodi di **explainable AI** come *SHAP* (SHapley Additive exPlanations) o *LIME* (Local Interpretable Model-agnostic Explanations) potrebbero fornire spiegazioni delle previsioni ottenute in modo da rendere il processo decisionale dei modelli più trasparente.

Infine potrebbe essere interessante approfondire le implicazioni etiche dell'utilizzo dei modelli di machine learning in campo medico. Infatti è fondamentale garantire che i modelli siano equi, privi di bias e in grado di fornire previsioni affidabili. L'utilizzo di algoritmi di fairness potrebbe contribuire a mitigare eventuali discriminazioni.

Dunque nel complesso lo studio condotto conferma l'importanza dei metodi di apprendimento automatico nel supportare le diagnosi e la prevenzione delle malattie cardiovascolari. Risulta comunque fondamentale continuare a monitorare e migliorare la stabilità e l'affidabilità degli stessi, specialmente quando applicati in settori ad alto impatto come quello sanitario.

Bibliografia

[1] Ra, A., Souza, Rio D., (2019) *A Review on Machine Learning Algorithms.* In *International Journal for Research in applied science & engineering technology* 7(VI), 792-796.

[2] LeCun, Y., Bengio, Y., Hinton, G., (2015) *Deep Learning.* In *Nature*, 521(7553), 436-444.

[3] Sutton, R., S., Barto, A., G., (2015) *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, Massacchussets, London, England

[4] Davis, J., Goadrich, M., (2006) *The Relationship Between Precision-Recall and ROC Curves*, University of Winsconsin-Madison.

[5] Willmott, C., J., Matsuura, K., (2005) *Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance.* In *Climate Research* Vol. 30: 79-82, University of Delaware.

[6] Bottou, L., Lin, C., J., (2007) *Support Vector Machine Solvers.* In *MITP.*

[7] Mohri, M., Rostamizadeh, A., Talwalkar, A., (2012), *Foundations of Machine Learning.*In *The MIT Press.*

[8] Polikar, R., (2006), *Ensemble based systems in decision making.* In *IEEE Circuits and Systems Magazine.*

[9] Schapire, R., E., (1990), *The strength of weak learnability.* In *Machine Learning* 5, 197-220.

[10] Freund, Y., Schapire, R., E., (1997), *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting.* In *Journal of computer and system sciences* 55, 119-139.

[11] Geeks for Geeks: https://www.geeksforgeeks.org/ml-label-encoding-of-datasets-in-python/.

[12] Geeks for Geeks: https://www.geeksforgeeks.org/ml-one-hot-encoding/.

[13] Geeks for Geeks: https://www.geeksforgeeks.org/how-to-perform-ordinal-encoding-using-sklearn/.

[14] Geeks for Geeks: https://www.geeksforgeeks.org/implementing-svm-from-scratch-in-python/.

[15] Geeks for Geeks: https://www.geeksforgeeks.org/implementing-the-adaboost-algorithm-from-scratch/.

[16] Han, J., Kamber, M., Pei, J., (2011), *Data Mining: Concepts and Techniques*. Morgan Kauffmann Pub, 3° edizione, pp. 113-115.