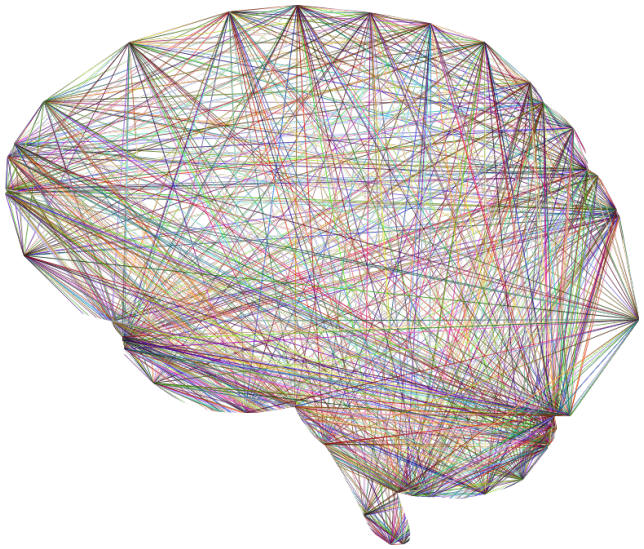
Fondamenti del Machine Learning

Daniele Antonucci, Elisabetta Casciano, Leonardo Pastorelli



Contents

1 Introduzione

Un modello di machine learning è una rappresentazione matematica o computazionale dei dati che l'algoritmo crea durante il processo di apprendimento. Il modello "impara" dai dati di input per fare delle previsioni o classificazioni.
Allenare un modello significa fornire al modello un insieme di **dati di addestramento**, che contengono esempi noti, affinché il modello possa apprendere le relazioni tra le variabili di input e il target di previsione. Durante l'allenamento, l'algoritmo cerca di minimizzare l'errore nelle sue previsioni, adattando i suoi parametri interni. Testare un modello implica utilizzare un insieme di dati separato da quello usato per l'allenamento, chiamato **set di test**. Questo permette di valutare la capacità del modello di generalizzare, ossia di fare previsioni accurate su nuovi dati che non ha mai visto prima. Testando il modello si può capire quanto bene esso stia funzionando, se è stato sovra-allenato (overfitting) su un particolare set di dati o se ha bisogno di ulteriori miglioramenti.
Nella trattazione, dopo una esaustiva spiegazione teorica circa il funzionamento di alcuni classificatori di cui è stato fatto uso, segue una descrizione del lavoro svolto in quattro step. In generale, dato un dataset composto da dati clinici, sono stati allenati e testati dei modelli per predire una classe di rischio (target). Infine sono stati elencati gli strumenti utilizzati per l'analisi dati, visualizzazione dei risultati, valutazione e ottimizzazione dei modelli.

2 Classificatori

2.1 K-nearest neighbors (KNN)

Il **K-nearest neighbors** (KNN) è un algoritmo di apprendimento supervisionato, comunemente utilizzato per problemi di classificazione e regressione. Si basa sulla semplice idea di identificare i "vicini più prossimi" di un dato punto di test e assegnare la classe (o il valore) in base a una maggioranza (nel caso della classificazione) o una media (nel caso della regressione) dei vicini.

Nella classificazione KNN, l'output è un'appartenenza a una classe. Un oggetto è classificato da un voto di pluralità dei suoi vicini, con l'oggetto assegnato alla classe più comune tra i suoi k vicini più vicini (k è un numero intero positivo, tipicamente piccolo). Se k = 1, l'oggetto viene semplicemente assegnato alla classe di quel singolo vicino più prossimo.

Nella regressione KNN, l'output è la media dei valori di k vicini più vicini.

Figure 1: K-nearest neighbors.

2.1.1 Principio di Funzionamento

L'algoritmo KNN prende come input un insieme di dati di addestramento, in cui ogni punto dati è etichettato con una classe (per la classificazione) o un valore numerico (per la regressione). Quando viene presentato un nuovo punto di test, l'algoritmo cerca i **k** vicini più prossimi nel set di addestramento, dove **k** è un parametro predefinito, e decide la classe (o il valore) del punto in base ai vicini più vicini. I vicini sono determinati solitamente tramite una metrica di distanza, come la distanza Euclidea.

Matematicamente, la distanza tra due punti x_i e x_j in uno spazio n-dimensionale può

essere definita come:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2}$$

dove x_{ik} è il valore della k-esima caratteristica di x_i, e x_{jk} è il valore della k-esima caratteristica di x_j.

2.1.2 Scelta del parametro k

Uno degli aspetti più importanti dell'algoritmo KNN è la scelta del parametro k, che determina il numero di vicini da considerare. Se k è troppo piccolo, l'algoritmo può essere sensibile al rumore nei dati, mentre se k è troppo grande, potrebbe non riuscire a catturare le caratteristiche locali dei dati. La scelta di k è quindi cruciale per ottenere un buon modello e può essere ottimizzata tramite tecniche di validazione incrociata.

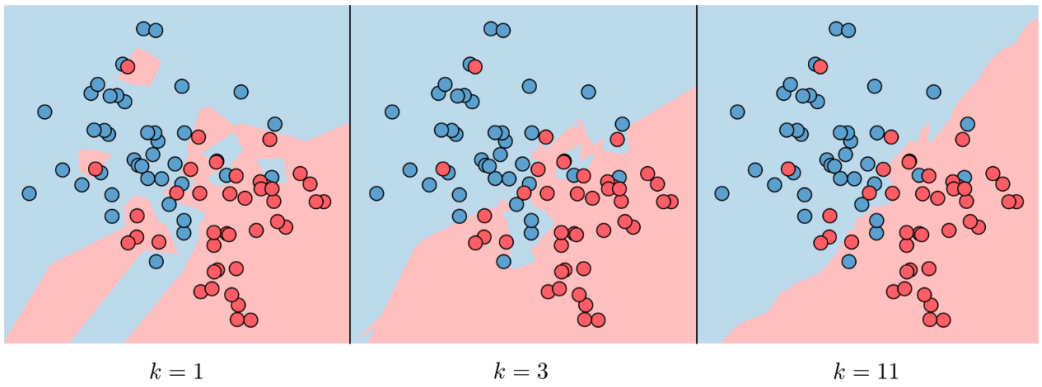


Figure 2: k values.

2.1.3 Metrica di Distanza

La metrica di distanza più comune utilizzata in KNN è la distanza Euclidea, ma l'algoritmo può anche essere adattato a diverse metriche di distanza, a seconda delle caratteristiche dei dati. Alcune delle metriche più comuni includono:

- **Distanza Euclidea:** La metrica più comune, usata nei dati numerici continui.
- **Distanza di Manhattan (o L1):** Spesso utilizzata in ambienti in cui i dati sono distribuiti in modo discreto o sparso.
- **Distanza Coseno:** Utilizzata principalmente per dati ad alta dimensione come nei modelli di text mining.

2.1.4 Algoritmo

L'algoritmo KNN può essere riassunto nei seguenti passaggi:

Algorithm 1 Algoritmo KNN per la classificazione

Input: Dati di addestramento $(X_{\text{train}}, y_{\text{train}})$, punto di test x_{test}, numero di vicini k
Output: Classe predetta per x_{test}
Calcola la distanza tra x_{test} e ogni punto in X_{train}
Ordina i punti di X_{train} in base alla distanza a x_{test}
Seleziona i k vicini più prossimi
Assegna al punto x_{test} la classe più frequente tra i k vicini

2.2 Gaussian Naive Bayes

Il **Gaussian Naive Bayes (GNB)** è una variante del classico *Naive Bayes* che assume che le caratteristiche (variabili indipendenti) siano distribuite secondo una distribuzione normale (gaussiana). Come il Naive Bayes tradizionale, GNB è un classificatore probabilistico che utilizza il teorema di Bayes per calcolare la probabilità di ciascuna classe, dato un insieme di caratteristiche osservate.

2.2.1 Teorema di Bayes

Il teorema di Bayes è alla base di questo modello e afferma che la probabilità di una classe C_k dato un insieme di osservazioni $X = (x_1, x_2, \ldots, x_n)$ è proporzionale alla probabilità della classe data le osservazioni, moltiplicata per la probabilità delle osservazioni date la classe, come segue:

$$P(C_k|X) = \frac{P(X|C_k)P(C_k)}{P(X)}$$

In un contesto di classificazione, vogliamo determinare quale classe C_k massimizza la probabilità condizionata $P(C_k|X)$. Il modello Naive Bayes, applicato al caso gaussiano, semplifica il calcolo delle probabilità $P(X|C_k)$, assumendo che le caratteristiche siano indipendenti dato la classe.

2.2.2 Assunzione di Normalità

Nel caso di *Gaussian Naive Bayes*, si assume che le variabili all'interno di ogni classe siano distribuite normalmente. Pertanto, la probabilità di ciascun x_i dato C_k (cioè $P(x_i|C_k)$) segue una distribuzione gaussiana:

$$P(x_i|C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(-\frac{(x_i - \mu_k)^2}{2\sigma_k^2}\right)$$

dove μ_k è la media della classe C_k e σ_k^2 è la varianza della classe C_k.

Poiché si assume l'indipendenza tra le caratteristiche, la probabilità totale di X dato la classe C_k è il prodotto delle probabilità individuali delle caratteristiche:

$$P(X|C_k) = \prod_{i=1}^{n} P(x_i|C_k)$$

2.2.3 Classificazione

Per classificare un nuovo campione $X = (x_1, x_2, \ldots, x_n)$, il modello calcola la probabilità per ogni classe C_k utilizzando il teorema di Bayes e quindi assegna la classe con la massima probabilità:

$$C_{pred} = \arg\max_{C_k} P(C_k) \prod_{i=1}^{n} P(x_i|C_k)$$

Dove $P(C_k)$ è la probabilità a priori della classe C_k, e $P(x_i|C_k)$ è la probabilità condizionata della caratteristica x_i data la classe C_k, assunta normale in questo caso.

2.2.4 Complessità Computazionale

La complessità computazionale di questo algoritmo è molto bassa, poiché il modello richiede solo la stima delle medie e delle varianze per ciascuna classe durante la fase di addestramento. In seguito, la fase di predizione è efficiente, poiché consiste nel calcolare semplicemente i valori di probabilità per ciascuna classe e nel confrontarli.

2.2.5 Algoritmo

L'algoritmo di classificazione di *Gaussian Naive Bayes* si compone di due fasi principali: la fase di *addestramento* e la fase di *predizione*.

Algorithm 2 Algoritmo Gaussian Naive Bayes per la classificazione

Input: Dati di addestramento $(X_{\text{train}}, y_{\text{train}})$, punto di test x_{test}
Output: Classe predetta per x_{test}
Fase di Addestramento:
Per ogni classe C_k:
Calcola la media μ_k e la varianza σ_k^2 per ogni caratteristica
Stima la probabilità a priori $P(C_k)$
Fase di Predizione:
Per ogni classe C_k:
Calcola la probabilità condizionata $P(x_{\text{test}}|C_k)$ usando:

$$P(x_i|C_k) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \exp\left(-\frac{(x_i - \mu_k)^2}{2\sigma_k^2}\right)$$

Combina con la probabilità a priori per ottenere:

$$P(C_k|x_{\text{test}}) \propto P(C_k) \prod_{i=1}^{n} P(x_i|C_k)$$

Seleziona la classe C_k con probabilità a posteriori massima:

$$C_{\text{pred}} = \arg\max_{C_k} P(C_k|x_{\text{test}})$$

2.3 Gradient Boosting

Il Gradient Boosting è un metodo che costruisce il modello finale combinando molti alberi di decisione semplici (weak learners) che lavorano insieme per ottenere un risultato accurato.

2.3.1 Alberi decisionali

In generale, un albero decisionale è composto da:

- Radice: il nodo iniziale, che rappresenta l'intero dataset.
- Nodi interni: punti decisionali dove i dati vengono divisi in base a una condizione (ad esempio, "L'età è maggiore di 30?").
- Foglie: i nodi terminali che contengono la previsione finale. In un problema di classificazione, la foglia rappresenta la classe predetta.

Gli alberi decisionali operano secondo il principio di divide et impera, cercando di suddividere i dati in sottoinsiemi più omogenei in base a determinate caratteristiche. Ogni nodo nell'albero esegue una domanda sui dati (ad esempio, se una variabile è maggiore o minore di un certo valore) e i dati vengono successivamente indirizzati a nodi diversi in base alla risposta (sì/no). La divisione avviene fino a quando non si raggiungono le foglie dell'albero, che contengono la previsione.

2.3.2 Cosa significa "boosting"?

Il boosting è un metodo che combina modelli deboli (come appunto alberi decisionali semplici) per costruire un modello complesso che fa previsioni migliori. Il boosting può lavorare sui residui (cioè la differenza tra le previsioni e i valori reali), o sui pseudo-residui, che sono versioni modificate dei residui. I pseudo-residui sono calcolati in base alla **funzione di perdita**. Il ruolo della funzione di perdita è inanzitutto il calcolo dell'errore: confronta l'output previsto del modello con i valori reali osservati . Il metodo per calcolare questa differenza varia a seconda della funzione di perdita scelta. Le funzioni di perdita più comuni sono il Mean Squared Error (usato principalmente nei problemi di regressione, calcola la somma dei quadrati delle differenze tra i valori previsti e quelli osservati) o il Cross-Entropy (misura la differenza tra due distribuzioni di probabilità, comunemente usata per i problemi di classificazione con target in categorie discrete).
Nell'addestramento l'obiettivo del modello è minimizzare la funzione di perdita, aggiornando i suoi parametri interni per ridurre l'errore.
Se i parametri del gradient boosting sono scelti bene, c'è basso rischio di overfitting : anche se un singolo albero decisionale può facilmente sovradattarsi ai dati (overfitting), l'uso di più alberi tramite boosting permette di combinare i punti di forza di ciascun albero, riducendo il rischio di overfitting grazie alla combinazione dei modelli.

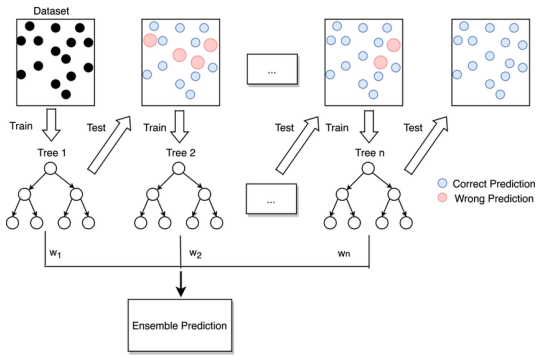


Figure 3: Gradient boosting

2.3.3 Come lavora il Gradient Boosting?

Consideriamo il seguente esempio:

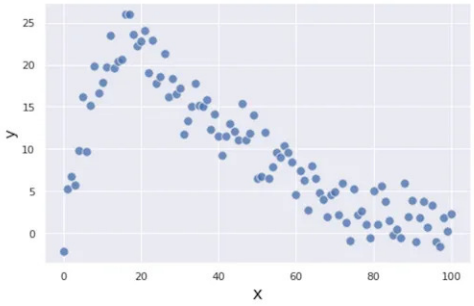


Figure 4: Esempio di dati

Il gradient boosting inizia con un modello semplice:

1. il primo albero di decisione cerca di fare una previsione iniziale sul dataset(solitamente, la media dei target nei dati di addestramento per i problemi di regressione o la probabilità di classe per la classificazione).Ovviamente non sarà perfetto e commetterà degli errori.

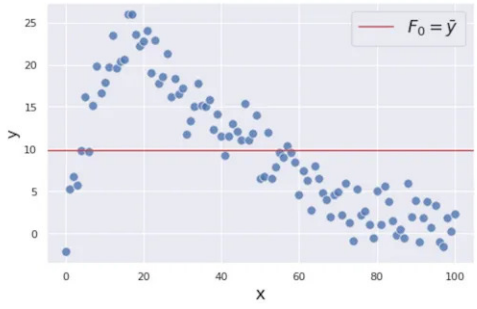
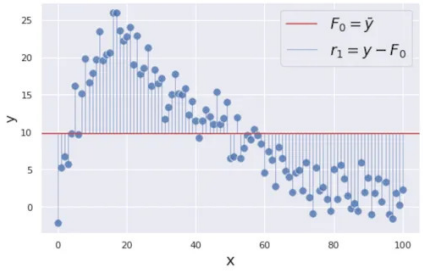
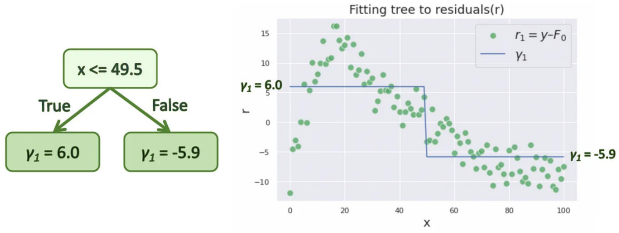


Figure 5: Previsione F0: media dei target

2. Calcolo dei residui (o pseudo-residui): In ogni iterazione, si calcolano i residui o pseudo-residui, che rappresentano la differenza tra il target reale e la previsione corrente. Pertanto, si trovano le differenze tra ogni valore osservato e la nostra previsione iniziale. I residui sono ciò che il nuovo albero cercherà di predire. I modelli successivi (altri alberi) vengono costruiti per correggere gli errori fatti dai modelli precedenti. Ad esempio, se il modello ha sottostimato un valore in una parte dei dati, il nuovo albero cercherà di fare previsioni più accurate per quelle specifiche istanze e cioè tenta di migliorare il modello globale.



(a) Calcolo dei residui r_1 (errore di predizione).



(b) Previsione dei residui: modello ad albero con x come feature e r_1 come target. Valori di esempio

$$\gamma_1 = \{6.0, -5.9\}$$

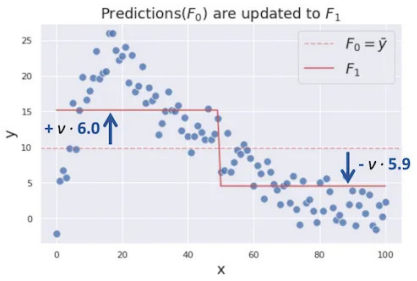
Figure 6: Visualizzazione dei residui e della previsione con modello ad albero.

Aggiungiamo la nuova predizione a quella iniziale per ridurre i residui. ν è il learning rate; γ è la previsione.

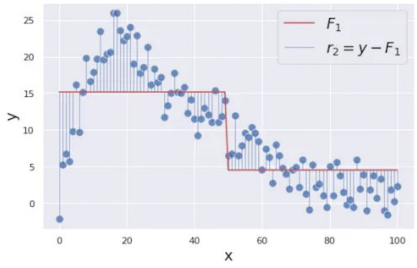
$$F_1 = F_0 + \nu \cdot \gamma_1$$

Nuova previsione F_1:

$$F_1 = \begin{cases} F_0 + \nu \cdot 6.0 & \text{if } x \leq 49.5 \\ F_0 - \nu \cdot 5.9 & \text{otherwise} \end{cases}$$

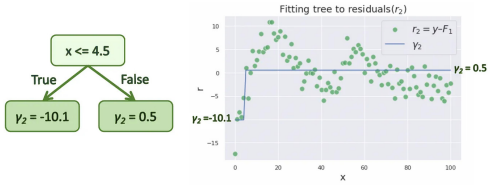


(c) Aggiunta nuova previsione F_1.

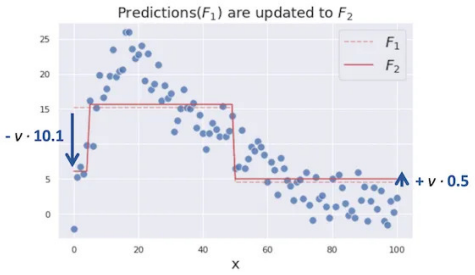


(d) Residui r_2.

Figure 7: Aggiornamento delle previsioni e residui.



(e) Nuovo albero: stessa feature x e residui r_2 come target.



(f) Le previsioni F_1 sono aggiornate a F_2.

Figure 8: Iterazioni con nuovo albero e aggiornamento delle previsioni.

Cioè iterando aggiungiamo i residui di ogni albero alla previsione iniziale per generare quella successiva.

3. Impara usando il gradiente: l'algoritmo utilizza l'idea di ottimizzazione del gradiente, ovvero minimizzare gradualmente gli errori. Dopo che ogni albero è stato addestrato, le sue previsioni vengono combinate con quelle degli alberi precedenti, come si vede nelle immagini precedenti. L'aggiornamento avviene con una somma ponderata delle previsioni, dove il tasso di apprendimento controlla l'importanza di ogni nuovo albero nel modello finale.

Il modello finale è la somma di tutte le previsioni fatte da ciascun albero, ovvero:

$$\hat{y}_i = \hat{y}_i^{(1)} + \nu \cdot f_2(x_i) + \nu \cdot f_3(x_i) + \cdots + \nu \cdot f_M(x_i) \tag{1}$$

dove $f_m(x_i)$ è la previsione dell'albero m-esimo per l'osservazione x_i.

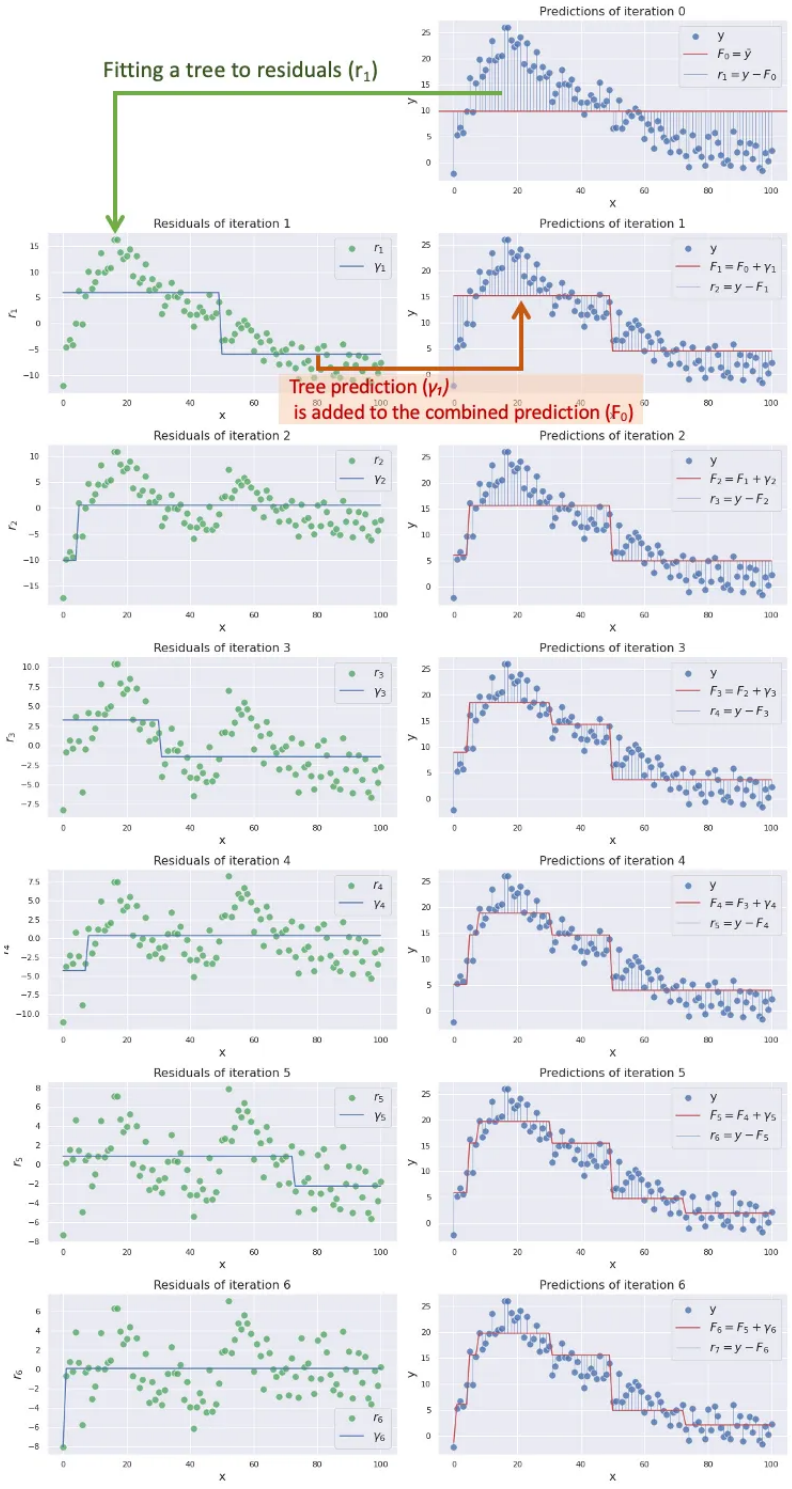


Figure 9: Processo iterativo del gradient boosting

2.3.4 I parametri del gradient boosting

I parametri che fanno parte del gradient boosting controllano un aspetto specifico dell'addestramento del modello e lavorano insieme per controllare la complessità, la capacità di generalizzazione e la velocità di addestramento.

- **n_estimators**:
  - Indica il numero di alberi decisionali da costruire (o il numero di iterazioni).
  - Un valore maggiore implica più alberi, ma aumenta il rischio di overfitting; un valore troppo basso potrebbe non consentire un apprendimento sufficiente. Un valore più alto può migliorare le performance, ma va bilanciato con il learning rate.
- **learning_rate**:
  - Controlla l'ampiezza degli aggiornamenti effettuati da ogni albero.
  - Valori bassi rendono gli aggiornamenti più lenti e robusti ma richiedono più alberi per convergere. Valori alti accelerano l'addestramento ma aumentano il rischio di overfitting.
- **max_depth**:
  - Controlla la profondità massima di ciascun albero.
  - Valori bassi (es. 3 o 5) limitano la complessità dell'albero, prevenendo l'overfitting. Valori più alti permettono alberi più complessi, con un rischio maggiore di sovradattamento.
- **subsample**:
  - Indica la frazione di campioni utilizzati per addestrare ogni albero.
  - Valore 1.0: utilizza tutto il dataset; valori inferiori effettuano sotto-campionamento.
  - Il sotto-campionamento può prevenire l'overfitting, ma riduce l'informazione disponibile per ogni albero.
- **min_samples_split**:
  - È il numero minimo di campioni richiesti per dividere un nodo interno.
  - Valori maggiori impediscono la creazione di nodi piccoli, prevenendo l'overfitting. Valori più bassi consentono alberi più complessi.
- **min_samples_leaf**:
  - È il numero minimo di campioni richiesti in un nodo foglia.
  - Valori maggiori riducono il rischio di adattarsi troppo ai dettagli specifici dei dati.
- **random_state**:
  - Indica il seme per il generatore di numeri casuali.
  - Garantisce la ripetibilità dei risultati e facilita il confronto tra configurazioni di modelli.

2.3.5 RandomizedSearchCV : ottimizzazione iperparametri per il gradient boosting

Al fine di trovare i migliori parametri per il gradient boosting, possiamo utilizzare il RandomizedSearchCV. Il RandomizedSearchCV è una tecnica di ricerca iperparametrica che esplora un sottogruppo casuale dello spazio dei parametri. A differenza della Grid Search, che esplora tutti i possibili valori di una griglia di parametri (il che può essere estremamente costoso computazionalmente), il RandomizedSearchCV sceglie un numero specificato di combinazioni casuali dei parametri, riducendo così il tempo di calcolo. Esegue dunque una ricerca casuale dei parametri per il modello di Gradient Boosting, cercando di ottimizzare i parametri del modello per ottenere la migliore performance possibile sul dataset di addestramento. In questo modo, possiamo ottenere buoni risultati senza dover esplorare completamente tutte le combinazioni. Tuttavia, il RandomizedSearchCV non garantisce di trovare i parametri ottimali poichè è una stima basata sullo spazio campionato.

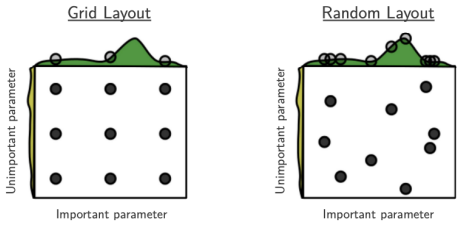


Figure 10: Grid search vs Random search

Come applicare il RandomizedSearchCV al gradient boosting : esempio in scikit learn

1. Viene definito un dizionario di parametri per il modello di Gradient Boosting. Ogni chiave del dizionario rappresenta un parametro del modello, e i valori associati sono i possibili intervalli da esplorare. Nel nostro caso, i parametri utilizzati sono:
   - **n_estimators**: il numero di alberi decisionali da costruire.
   - **learning_rate**: il tasso di apprendimento, che controlla quanto viene modificato ogni albero.
   - **max_depth**: la profondità massima di ciascun albero.
   - **subsample**: la frazione di dati da usare per ogni albero.
   - **min_samples_split**: il numero minimo di campioni necessari per dividere un nodo in un albero.
   - **min_samples_leaf**: il numero minimo di campioni in una foglia di albero.
2. **Definizione del modello**
   `model = GradientBoostingClassifier(random_state=42)`
   Viene creato il modello di Gradient Boosting per la classificazione (`GradientBoostingClassifier`). La proprietà `random_state=42` è impostata per garantire che i risultati siano riproducibili in modo che, ogni volta che si esegue il codice, si ottengano gli stessi risultati.
3. **Creazione di RandomizedSearchCV**
   `random_search = RandomizedSearchCV(`

`model, param_distributions, n_iter=50, cv=3, scoring='accuracy', n_jobs=-1, random_state=42, verbose=1)`
La classe `RandomizedSearchCV` viene utilizzata per eseguire una ricerca casuale dei migliori parametri per il modello. I parametri di configurazione sono:

- **model**: il modello da ottimizzare (il `GradientBoostingClassifier`).
- **param_distributions**: il dizionario che definisce i parametri da esplorare.
- **n_iter=50**: il numero di combinazioni di parametri da provare.
- **cv=3**: il numero di fold nella cross-validation.
- **scoring='accuracy'**: il criterio di valutazione delle performance del modello.
- **n_jobs=-1**: il calcolo viene distribuito su tutti i core della CPU disponibili.
- **random_state=42**: garantisce che la ricerca sia riproducibile.
- **verbose=1**: controlla il livello di dettagli dell'output.

4. **Addestramento del modello con ricerca casuale**
`random_search.fit(X_train, y_train)`
In questa fase, `RandomizedSearchCV` esegue la ricerca dei migliori parametri. Viene addestrato il modello con le combinazioni casuali dei parametri definiti, e per ciascuna combinazione, il modello viene validato usando `la cross-validation`. Viene quindi calcolata l'accuratezza per ciascun set di parametri.

   Durante questo processo, `RandomizedSearchCV` esegue un totale di 50 iterazioni (combinazioni di parametri), per ciascuna delle quali esegue la validazione incrociata (cross-validation) con 3 fold.

5. **Visualizzazione dei migliori parametri**
`print("Migliori parametri:", random_search.best_params_)`
Una volta completata la ricerca, vengono stampati i migliori parametri trovati, che sono quelli che hanno portato alla migliore performance durante il processo di ricerca (in questo caso, misurata tramite accuratezza). Adesso è possibile allenare il modello con i migliori parametri!

2.4 Perceptron & MLP(Multi-Layer Perceptron)

2.4.1 Perceptron

Il percettrone, introdotto da Frank Rosenblatt nel 1958, è uno dei primi modelli di apprendimento supervisionato e rappresenta il fondamento teorico delle moderne reti neurali artificiali. Si tratta di un modello concepito per risolvere problemi di classificazione binaria, basandosi sull'assunzione che i dati siano linearmente separabili. Il percettrone è un sistema semplice ma potente, in grado di apprendere un iperpiano che separa due classi distinte di dati. La struttura del percettrone si basa su un singolo nodo, o neurone, che riceve

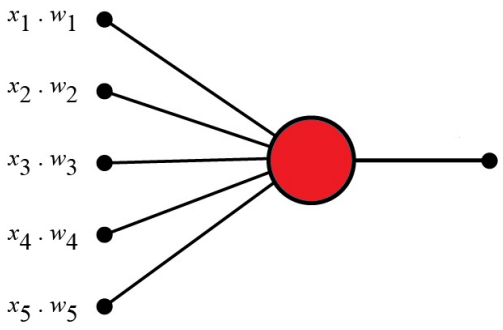


Figure 11: Perceptron.

in ingresso un vettore di caratteristiche, ognuna delle quali è associata a un peso. Questi pesi determinano l'importanza di ciascuna caratteristica nel processo di classificazione. Il modello include inoltre un termine di bias, che consente al percettrone di spostare il piano decisionale, rendendo il sistema più flessibile. Una volta che gli ingressi sono combinati linearmente con i pesi e il bias, il risultato viene passato attraverso una funzione di attivazione, generalmente una funzione *step*. Questa funzione determina se il neurone deve attivarsi, ossia classificare l'input in una delle due categorie previste.

La decisione del percettrone si basa su una semplice equazione matematica:

$$y = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$$

dove $\mathbf{w}$ rappresenta il vettore dei pesi, $\mathbf{x}$ è il vettore degli ingressi, e b è il bias. La funzione sign() restituisce un valore positivo o negativo in base al risultato dell'operazione lineare, assegnando così l'input a una delle due classi.

Per quanto riguarda il processo di apprendimento, il percettrone segue un approccio iterativo che modifica i pesi in modo da minimizzare gli errori di classificazione. Durante l'addestramento, il modello analizza ciascun esempio e confronta l'output predetto con l'etichetta reale. Se la predizione è corretta, i pesi rimangono invariati; al contrario, se vi è un errore, i pesi vengono aggiornati seguendo una regola ben definita:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot (y_i - \hat{y}_i) \cdot \mathbf{x}_i$$

$$b \leftarrow b + \eta \cdot (y_i - \hat{y}_i)$$

dove η è il tasso di apprendimento, un parametro che controlla l'entità degli aggiustamenti. Questo processo si ripete fino a quando tutti gli esempi sono classificati correttamente o viene raggiunto un numero massimo di iterazioni.

Nel caso in cui gli input e gli output sono dello stesso tipo, è possibile creare reti più complesse unendo più percettroni insieme, per esempio usando un gruppo (o strato) di percettroni come input per un secondo gruppo di percettroni, oppure facendo in modo che l'input di ogni percettrone della rete sia dato dall'output di ogni altro percettrone (rete fully-connected). Un aspetto fondamentale del percettrone è che funziona correttamente solo se i dati sono linearmente separabili. In altre parole, il modello è in grado di trovare una soluzione soltanto se esiste un iperpiano che separa esattamente le due classi. Nonostante la sua semplicità, il percettrone ha avuto un impatto significativo nel campo dell'apprendimento automatico. Oltre a fornire una soluzione efficace per i problemi linearmente separabili, ha aperto la strada allo sviluppo di modelli più avanzati, come il *Multi-Layer Perceptron* (MLP).

2.4.2 Estensione del Percettrone per Problemi Multiclasse

Il percettrone classico è progettato per risolvere problemi di classificazione binaria, ma molte applicazioni pratiche richiedono la gestione di problemi con più di due classi. Per affrontare problemi di classificazione multiclasse, è necessario estendere il modello del percettrone utilizzando approcci specifici. Due delle tecniche più comuni sono il metodo *One-vs-All* (OvA) e il metodo *One-vs-One* (OvO).

2.4.3 Metodo One-vs-All (OvA)

Il metodo *One-vs-All*, noto anche come *One-vs-Rest*, consiste nella costruzione di un modello separato per ciascuna classe. Ogni modello è addestrato a distinguere una singola classe (la classe "positiva") da tutte le altre classi combinate (le classi "negative"). Durante la fase di predizione, il modello che restituisce il punteggio più alto o la probabilità più elevata determina l'assegnazione della classe per l'osservazione.

Questa strategia è semplice da implementare e risulta efficace in molti scenari. Tuttavia, può risultare problematica quando le classi sono sbilanciate o quando le classi negative condividono caratteristiche simili alla classe positiva.

2.4.4 Metodo One-vs-One (OvO)

Il metodo *One-vs-One*, invece, suddivide il problema multiclasse in molteplici problemi binari, ognuno dei quali coinvolge due sole classi. Per k classi, vengono costruiti $\binom{k}{2}$ modelli, ciascuno addestrato a distinguere una coppia di classi. Durante la predizione, ogni classificatore vota per una delle due classi che sta distinguendo e la classe con il maggior numero di voti è assegnata all'osservazione.

Questo approccio è particolarmente utile quando il numero di classi è limitato, poiché ogni classificatore lavora con una quantità ridotta di dati. Tuttavia, il numero di modelli da costruire aumenta rapidamente con il numero di classi, rendendolo meno pratico per problemi con molte classi.

2.5 Multi-Layer Perceptron (MLP)

Il *Multi-Layer Perceptron* (MLP) rappresenta un'estensione del modello Percettrone e costituisce una delle forme più comuni e potenti di reti neurali artificiali. A differenza del Percettrone classico, che è limitato alla classificazione binaria di dati linearmente separabili, l'MLP è in grado di risolvere problemi complessi e non linearmente separabili grazie alla sua architettura stratificata e alle funzioni di attivazione non lineari.

2.5.1 Architettura del Modello

Un MLP è composto da:

- **Strato di Input**: Lo strato di ingresso riceve i dati in forma di vettori di caratteristiche. Ogni neurone in questo strato corrisponde a una caratteristica del dataset.
- **Strati Nascosti (Hidden Layers)**: Gli strati nascosti sono costituiti da neuroni che ricevono input dagli strati precedenti e trasmettono output agli strati successivi. Ogni strato nascosto contribuisce a modellare le relazioni complesse tra i dati.
- **Strato di Output**: Lo strato di uscita fornisce la predizione finale. Il numero di neuroni in questo strato dipende dal numero di classi nel problema di classificazione.

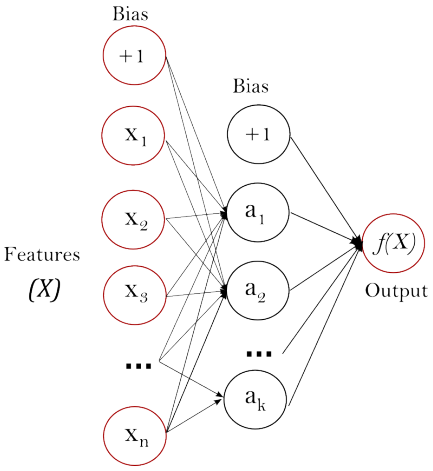


Figure 12: MLP con uno strato nascosto.

Ogni connessione tra i neuroni ha un peso associato, che viene aggiornato durante la fase di addestramento per minimizzare l'errore della rete.

2.5.2 Funzionamento del MLP

L'MLP segue un approccio in due fasi principali:

Forward Propagation: Durante questa fase, i dati vengono propagati dallo strato di input attraverso gli strati nascosti fino allo strato di output. Per ogni neurone, il valore in uscita è calcolato come una combinazione pesata degli input, a cui viene applicata una

funzione di attivazione non lineare, tipicamente la *ReLU* (Rectified Linear Unit), la *sigmoide* o la *tangente iperbolica*. La formula per il calcolo dell'output di un neurone è:

$$y_j = f\left(\sum_{i=1}^{n} w_{ij}x_i + b_j\right)$$

dove:

- x_i: sono gli input del neurone.
- w_{ij}: sono i pesi associati agli input.
- b_j: è il bias del neurone.
- f: è la funzione di attivazione.

Backpropagation: La propagazione all'indietro è il cuore del processo di apprendimento dell'MLP. Durante questa fase, l'errore della rete è calcolato confrontando l'output della rete con il valore atteso (label). Questo errore viene poi propagato all'indietro attraverso la rete per aggiornare i pesi utilizzando l'algoritmo di discesa del gradiente e le sue varianti, come *Stochastic Gradient Descent (SGD)* o *Adam*. La funzione di perdita più comunemente utilizzata per i problemi di classificazione è l'entropia incrociata (*cross-entropy loss*).

2.5.3 Caratteristiche Principali

L'MLP si distingue per alcune caratteristiche fondamentali che lo rendono un modello potente e flessibile. Una delle sue qualità principali è la capacità di apprendere relazioni non lineari nei dati. Questo è reso possibile dalla presenza di strati nascosti e dall'uso di funzioni di attivazione non lineari, che permettono al modello di catturare schemi complessi e di rappresentare relazioni più sofisticate rispetto ai modelli lineari tradizionali.

Inoltre, l'MLP è estremamente versatile, potendo essere utilizzato sia per problemi di regressione, dove si mira a prevedere valori continui, sia per problemi di classificazione, dove l'obiettivo è assegnare gli input a classi discrete.

Un'altra caratteristica rilevante dell'MLP è la capacità di ottimizzare i suoi parametri attraverso tecniche avanzate. Grazie all'algoritmo di *Backpropagation* e a metodi di ottimizzazione come Adam o SGD, il modello riesce ad adattarsi ai dati in modo efficace, migliorando le sue prestazioni attraverso un processo iterativo di apprendimento. Queste caratteristiche fanno dell'MLP uno strumento particolarmente utile per una vasta gamma di applicazioni.

2.6 Random Forest

Il Random Forest è una 'foresta' di molti alberi decisionali. L'albero decisionale è un modello che prende decisioni dividendo i dati in base a regole di decisione. Il processo di decisione continua finché non si raggiunge una foglia. Gli alberi del Random Forest sono tutti diversi fra loro.

2.6.1 Come fare a diversificare gli alberi

Gli alberi sono diversi grazie a due elementi chiave:

1. Sottoinsiemi di dati (Bootstrap Sampling): ogni albero viene addestrato su un sottoinsieme casuale del dataset originale in modo tale che gli alberi non imparino tutti le stesse cose.

2. Sottoinsiemi di caratteristiche: quando un albero decide come dividere i dati, considera solo un sottoinsieme casuale delle caratteristiche. Questo rende ogni albero unico e indipendente.

2.6.2 Come avviene la predizione

Una volta che tutti gli alberi sono stati addestrati, la Random Forest combina le loro predizioni per classificazione (ogni albero 'vota' per una classe, e la classe con più voti vince) o per regressione (si calcola la media delle predizioni degli alberi)

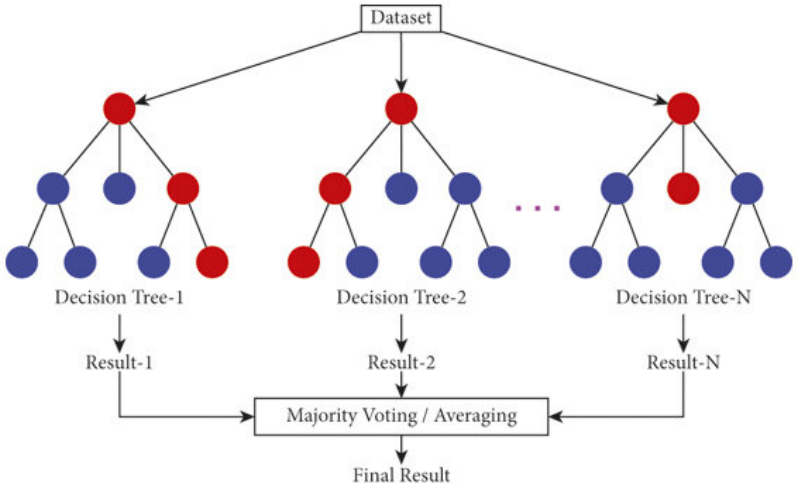


Figure 13: Random Forest.

3 Descrizione degli step

3.1 Step 1

Il primo punto consiste nella definizione manuale di due classificatori e nella valutazione delle prestazioni ottenute sul file 'manuale.csv'.
I classificatori descritti sono il KNN (k-nearest neighbors) e il Gaussian Naive Bayes.

3.1.1 Risultati

La tabella seguente riassume i risultati dei due classificatori, KNN e Gaussian Naive Bayes, in termini di accuratezza e analisi della matrice di confusione, sui dati originali e ripuliti. I valori di k corrispondono al migliore trovato con cross validation. La gestione dei valori anomali per ripulire il dataset è stata fatta sostituendoli con la mediana.

Classificatore	**Dataset**	**Parametro**	**Accuratezza**
KNN	Originale	$k = 5$	0.83
KNN	Ripulito	$k = 4$	0.83
Gaussian Naive Bayes	Originale	N/A	0.77
Gaussian Naive Bayes	Ripulito	N/A	0.83

Table 1: Confronto delle prestazioni di KNN e Gaussian Naive Bayes sui dati originali e ripuliti.

3.2 Step 2

La seconda task richiede di effettuare una pulizia del dataset 'training.csv' per la gestione ad esempio di valori nulli, errati o outlier. Inoltre è richiesta un'analisi relativa a statistiche di base e distribuzione dei dati anche attraverso strumenti grafici come boxplot, pairplot e matrice di correlazione.

3.2.1 Pulizia del dataset

A partire dal file 'training.csv' sono state effettuate le seguenti modifiche:

- rimozione della colonna 'Unnamed:7' dopo aver sistemato errori di formattazione;
- rimozione del sample in cui 'Age'='?' e dei samples in cui il valore dell'età è inconsistente;
- sostituzione dei valori 'NaN' di 'BS' e 'DiastolicPB' con la relativa media;
- sostituzione dei valori inconsistenti di 'HeartRate' e 'BodyTemp' con la relativa media;
- rimozione della riga in cui il campo 'RiskLevel' è inconsistente;

3.2.2 Analisi del dataset

Per svolgere un'analisi del dataset è stato necessario utilizzare librerie che offrono strumenti grafici, come il boxplot, il pairplot e la matrice di correlazione. Inoltre è stata effettuata un'analisi statistica richiamando il metodo describe() prima e dopo aver effettuato la pulizia dei dati. Il boxplot ha fatto emergere valore centrale dei dati ed eventuali outlier e distribuzioni non simmetriche dei dati. Il pairplot ha illustrato relazioni tra variabili ed è stato realizzato l'istogramma che mostra la frequenza di ognuno dei tre livelli di rischio. La matrice di correlazione ha rivelato connessioni tra le variabili del set di dati.

3.3 Step 3

Nel punto 3 sono stati implementati i classificatori progettati al punto 1) in Python e sono state valutate le performance ottenute da ognuno di essi sul dataset "training.csv" o su qualche suo sottoinsieme, cercando di ottimizzare le prestazioni dei classificatori.

Dopo aver importato i classificatori personalizzati, implementati nel punto 1) (le funzioni sono state definite nel file `functions.py`), si procede con la preparazione dei dati. In particolare, il dataset viene separato nelle feature (`X`) e nel target (`y`), dove `RiskLevel` è il target da prevedere. Le etichette del target(high risk, mid risk, low risk) vengono codificate in numeri tramite il `LabelEncoder` di `scikit-learn`, e il dataset viene diviso in due set: `X_train` e `X_test` per le feature, e `y_train` e `y_test` per le etichette.

3.3.1 KNN (K-Nearest Neighbors)

Per il KNN è stata utilizzata la funzione `find_best_k` per determinare il miglior valore di `k` attraverso una validazione incrociata a 5 fold. Effettuata la previsione sul set di test, viene calcolata l'accuratezza finale del modello. Successivamente, vengono calcolati il `precision`, il `recall`, e l'`F1-score`, che sono metriche utili per valutare l'equilibrio tra i falsi positivi e i falsi negativi nel modello. In questo caso, l'`F1-score` è identico all'accuratezza, indicando che il modello è bilanciato e performa bene su entrambe le metriche.

Un altro strumento di valutazione del modello è la matrice di confusione, mostrando come il modello ha classificato correttamente o erroneamente le diverse classi del target.

3.3.2 Gaussian Naive Bayes(GNB)

Applichiamo le funzioni personalizzate per calcolare i parametri del modello e per fare le previsioni. Anche in questo caso, viene visualizzata la matrice di confusione per valutare le prestazioni del modello. Successivamente, viene eseguita una cross-validation per calcolare l'accuratezza media del modello Naive Bayes, utilizzando una validazione incrociata a 5 fold.

Notiamo come il KNN si comporti particolarmente bene con il dataset esteso; un'accuratezza minore risulta invece dal GNB (potrebbe essere poichè non c'è indipendenza nelle features, o per la distribuzione dei dati).

Classificatore	Dataset	Accuratezza
KNN	training$_c$	0.84
Gaussian Naive Bayes	training$_c$	0.63

Table 2: Modelli KNN e GNB utilizzati per il dataset training$_c$.*csv*

Per completezza, i classificatori KNN e Gaussian Naive Bayes sono stati implementati (oltre che manualmente nel file step3manuale.ipynb) anche con scikit learn nel file step3scikit.ipynb.

3.4 Step 4

Il punto 4 si concentra su diversi classificatori implementati con scikit learn e addestrati utilizzando il training set. Ogni classificatore viene valutato in base alle sue prestazioni sul test set, per identificare quale modello riesce a massimizzare l'accuratezza nella previsione del target `RiskLevel`. I classificatori considerati sono gli stessi documentati precedentemente (Gradient Boosting, MLP, Random Forest)

Classificatore	Accuratezza	F1-Score
Gradient Boosting	0.89	0.89
MLP	0.86	0.86
Random Forest	0.87	0.89

Table 3: Confronto tra i modelli GradientBoosting, MLP e RandomForest

3.4.1 Gradient Boosting

Al fine di ottimizzare le prestazioni del gradient boosting, è stato utilizzato il RandomizedSearch per trovare gli iperparametri migliori. L'accuratezza rilevata è di 0.89 e nella matrice di confusione si vede come il modello sia abbastanza preciso. Dai vari strumenti di analisi, si deduce che la feature più influente è SystolicBP; osservando la curva precision-recall, per la classe "high risk" si nota un buon bilanciamento tra precision e recall nella maggior parte dei casi. La precision si mantiene molto elevata (vicina a 1) anche per valori di recall alti (maggiori di 0.8). Ciò suggerisce che il modello è in grado di identificare correttamente molti casi della classe "high risk" mantenendo pochi falsi positivi. Anche per la classe "low risk", la precision è generalmente alta. La classe "mid risk" ha una curva più irregolare rispetto alle altre due, con una precision che cala più rapidamente al crescere di recall. Il modello pertanto fa più fatica a distinguere correttamente la classe "mid risk". La curva ROC indica che il modello ha un'ottima capacità di discriminare le classi "high risk", "low risk" e "mid risk". Le AUC sono tutte vicine a 1, cioè c'è una ottima distinzione delle classi. Per quanto riguarda invece la curva di apprendimento, la training accuracy inizia molto vicina al 100 percento per set di dati piccoli e diminuisce leggermente all'aumentare della dimensione del training set. La validation accuracy invece parte bassa (circa 60 percento) per set molto piccoli e cresce gradualmente all'aumentare della dimensione del training set. Per set piccoli

quindi c'è abbastanza divario tra le curve, e quindi overfitting, ma all'aumento di dati questo è mitigato.

3.4.2 MLP

Inizialmente abbiamo provato ad addestrare un modello Perceptron, caratterizzato da parametri trovati tramite la GridSearchCV. Per la non linearità rilevata nei dati abbiamo scelto di provare ad allenare un modello più complesso(MLP) che permette di modellare relazioni non lineari tra i dati. Il modello risulta avere prestazioni sicuramente migliori rispetto a quelle del Perceptron. Prima del training abbiamo fatto un'analisi per verificare il bilanciamento del dataset. Uno sbilanciamento del dataset potrebbe influire sulle prestazioni complessive del modello. Abbiamo rilevato uno sbilanciamento soprattutto per quanto riguarda la classe `HighRisk`, questo causava un minor `F1-score` per la classe `HighRisk`. Per questo motivo abbiamo utilizzato delle tecniche di bilanciamento di `scikit-learn` come SMOTE e soglia di decisione. Infine sono stati utilizzati strumenti come la matrice di confusione curva di apprendimento e di valutazione per fare un'analisi delle prestazioni del modello.

3.4.3 Random Forest

Per allenare il modello è stato utilizzato l'80% dei dati nel dataset. Il GridSearch ha fornito i migliori iperparametri (numero di alberi, profondità massima, numero minimo di campioni per dividere un nodo, numero minimo di campioni in una foglia). L'accuratezza sul test set di 0.886 risulta essere molto buona. Dal classification report emerge una gestione ottima di tutte le classi anche se la classe 'mid risk' presenta prestazioni leggermente più basse e quindi risulta essere più difficile da identificare. La matrice di confusione riporta una buona qualità di predizione. Sono state realizzate le curve di apprendimento e di validazione in relazione al numero degli alberi. Dalla curva di apprendimento si osserva un peggioramento della performance sul training set con l'aumentare dei dati, ma contemporaneamente un miglioramento della performance sul test set. Questo comportamento indica che il modello sta generalizzando bene.

4 Strumenti utilizzati

In questo progetto sono stati utilizzati diversi strumenti per l'analisi dei dati, la visualizzazione dei risultati e la valutazione delle prestazioni dei modelli di classificazione. Di seguito viene fornita una descrizione di ciascuno di questi strumenti.

4.1 Metriche di valutazione

4.1.1 Accuratezza (Accuracy)

L'**accuratezza** è una metrica di valutazione che misura la proporzione delle previsioni corrette rispetto al totale delle previsioni effettuate. La formula matematica è la seguente:

$$\text{Accuracy} = \frac{\text{Numero di previsioni corrette}}{\text{Totale delle previsioni}} = \frac{TP + TN}{TP + TN + FP + FN}$$

Dove:

- TP = Veri Positivi
- TN = Veri Negativi
- FP = Falsi Positivi
- FN = Falsi Negativi

4.1.2 Precision (Precisione)

La **precisione** misura la proporzione dei risultati positivi previsti che sono effettivamente positivi. Essa è definita come:

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

Questa metrica è utile in scenari in cui i falsi positivi hanno un alto costo.

4.1.3 Recall (Sensibilità o Tasso di Recupero)

Il **recall** misura la capacità del modello di catturare i risultati positivi reali. È definito come:

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

Questa metrica è cruciale quando è importante non perdere risultati positivi, come ad esempio nella diagnosi medica.

4.1.4 F1-Score

L'**F1-score** è la media armonica tra Precision e Recall. Fornisce un bilanciamento tra le due metriche ed è calcolato come segue:

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

L'F1-Score è particolarmente utile quando il dataset è sbilanciato.

4.1.5 Cross-Validation

La **cross-validation** è una tecnica statistica utilizzata per valutare la capacità di generalizzazione di un modello di Machine Learning. L'approccio più comune è la *k-fold cross-validation*, che funziona come segue:

- Il dataset viene suddiviso in k sottoinsiemi (folds) di dimensioni uguali.
- Il modello viene allenato su $k-1$ sottoinsiemi e testato sull'insieme rimanente.
- Il processo viene ripetuto k volte, cambiando ogni volta il sottoinsieme utilizzato per il test.

L'accuratezza finale è data dalla media delle k accuratezze ottenute nei vari passaggi.

$$\text{Accuratezza media} = \frac{1}{k} \sum_{i=1}^{k} \text{Accuratezza}_i$$

Questa tecnica aiuta a mitigare il problema dell'*overfitting* e fornisce una stima più affidabile della performance del modello.

4.2 Plot

4.2.1 Boxplot

Il **boxplot** è uno strumento di visualizzazione che mostra la distribuzione dei dati attraverso i quartili. Esso visualizza i valori minimi, massimi, la mediana e i quartili (primo e terzo) di una distribuzione. È particolarmente utile per identificare la presenza di valori anomali o outliers nei dati, poiché questi vengono rappresentati come punti separati al di fuori dei "baffi" (whiskers) del boxplot.

Nel nostro caso, il boxplot è stato utilizzato per visualizzare la distribuzione delle feature nel dataset, per individuare eventuali valori anomali e capire come le feature si distribuiscono tra le diverse classi del target.

4.2.2 Scatterplot

Lo **scatterplot** (grafico a dispersione) è uno strumento di visualizzazione che mostra la relazione tra due variabili numeriche. Ogni punto nel grafico rappresenta un'osservazione nel dataset, e le sue coordinate sono determinate dai valori delle due variabili in esame. È utile per osservare visivamente come una variabile possa influenzare un'altra e se esistano tendenze, correlazioni o separazioni tra le classi nel dataset.

4.2.3 Pairplot

Il **pairplot** è uno strumento di visualizzazione che mostra tutte le possibili combinazioni di coppie di variabili del dataset. Per ogni coppia di variabili, viene visualizzato un grafico di dispersione (scatterplot), mentre sulla diagonale vengono visualizzati gli istogrammi delle singole variabili. Questo strumento è utile per osservare le relazioni tra le variabili e identificare pattern che potrebbero essere sfruttati dai modelli di classificazione.

Il pairplot è stato utilizzato per esplorare le relazioni tra le feature, aiutando a capire come queste interagiscono tra loro e come possano essere correlate con il target.

4.2.4 Histogram (Hist)

L'**istogramma** è uno strumento di visualizzazione che consente di rappresentare la distribuzione di una singola variabile numerica. Mostra quante volte i valori di una variabile ricadono all'interno di determinate gamme (bin). L'istogramma è stato utilizzato per visualizzare la distribuzione delle feature numeriche, aiutando a comprendere meglio la loro dispersione e la loro forma, utile per applicare trasformazioni appropriate (come la normalizzazione o la standardizzazione).

4.2.5 Heatmap(matrice di correlazione)

La **heatmap** è uno strumento di visualizzazione che utilizza colori per rappresentare valori numerici in una matrice o tabella. È particolarmente utile per visualizzare la matrice di correlazione tra le variabili del dataset. In particolare, la heatmap permette di identificare rapidamente le variabili che sono altamente correlate, aiutando nella selezione delle feature e nella comprensione delle interazioni tra le variabili.

Nel nostro caso, la heatmap è stata utilizzata per visualizzare la matrice di correlazione tra le feature, permettendo di identificare eventuali correlazioni forti che avrebbero potuto influire sulle performance dei modelli.

In questo progetto, lo scatterplot è stato utilizzato per esplorare le relazioni tra le variabili indipendenti e il target, cercando pattern che potessero suggerire come le variabili influenzano le classi.

4.2.6 Confusion matrix

La **matrice di confusione** è una tabella che mostra il numero di previsioni corrette e errate di un classificatore rispetto a ciascuna classe del target. Ogni cella rappresenta un conteggio delle predizioni fatte dal modello. Le righe della matrice rappresentano le classi reali, mentre le colonne rappresentano le classi predette. La matrice di confusione è uno strumento fondamentale per valutare le performance di un modello, poiché permette di vedere se il modello tende a commettere errori sistematici, come confondere certe classi.

Nel nostro caso, è stata utilizzata per visualizzare le performance dei modelli di classificazione, per valutare quanto bene ogni modello sia riuscito a predire ciascuna classe.

4.2.7 Validation Curve

La **validation curve** è uno strumento di visualizzazione che mostra come cambiano le prestazioni di un modello (in termini di accuratezza, precisione, etc.) al variare di un parametro di iperparametro del modello. Di solito, si esegue una validazione incrociata (cross-validation) per calcolare le prestazioni del modello su diverse configurazioni. La curva aiuta a identificare la gamma ottimale di valori per un iperparametro, evitando il sovra-adattamento o l'underfitting.

In questo progetto, la validation curve è stata utilizzata per ottimizzare gli iperparametri dei modelli di classificazione, come il valore di k per il classificatore KNN.

4.2.8 Learning Curve

La **learning curve** è uno strumento che mostra come le prestazioni del modello migliorano al variare della dimensione del training set. In altre parole, mostra come l'accuratezza aumenta quando vengono utilizzati più dati per addestrare il modello. Questo strumento è

utile per capire se il modello ha bisogno di più dati per migliorare le prestazioni o se sta già imparando in modo efficace con il dataset corrente.

Nel nostro caso, la learning curve è stata utilizzata per osservare l'andamento delle prestazioni del modello con l'aumento della dimensione del training set, identificando eventuali problemi legati alla quantità di dati disponibili.

4.2.9 ROC-AUC curve

La **curva ROC (Receiver Operating Characteristic)** è un grafico che mostra la relazione tra il tasso di veri positivi (True Positive Rate) e il tasso di falsi positivi (False Positive Rate) per diversi valori di soglia del modello. La **AUC (Area Under the Curve)** è l'area sotto la curva ROC e rappresenta la capacità del modello di distinguere tra le classi. Un AUC maggiore di 0.5 indica che il modello ha una capacità migliore di distinguere tra le classi rispetto a un modello casuale.

La curva ROC-AUC è stata utilizzata per valutare la capacità discriminante del modello sui dati di test. Un modello perfetto ha un punto (0, 1) sulla curva ROC. Questo indica che il modello ha zero falsi positivi (FPR = 0) e tutti i positivi sono correttamente classificati (TPR = 1). Questo è l'ideale. Se la curva ROC è vicina alla diagonale, il modello sta classificando casualmente. Un modello che ha un AUC vicino a 0.0 è praticamente inutile, in quanto sta sbagliando costantemente nel classificare le istanze.

4.2.10 Precision-Recall curve

La **curva precision-recall** è simile alla curva ROC, ma si concentra su precisione e recall piuttosto che sui tassi di falsi positivi e veri positivi. È particolarmente utile quando i dati sono sbilanciati, poiché fornisce una visione più chiara delle prestazioni del modello sulle classi minoritarie. La precisione misura la frazione di veri positivi tra tutte le previsioni positive, mentre il recall misura la frazione di veri positivi tra tutti i veri positivi.

Nel nostro caso, la curva precision-recall è stata utilizzata per esaminare il bilanciamento tra precisione e recall, specialmente per le classi meno rappresentate nel dataset.

4.3 Ottimizzazione

4.3.1 GridSearch & RandomizedSearch

GridSearch è un metodo per la ricerca automatica degli iperparametri di un modello. Esso esplora tutte le possibili combinazioni di valori predefiniti per un set di iperparametri. È utile per ottimizzare il modello, cercando la combinazione di parametri che massimizza le prestazioni. Tuttavia, può essere computazionalmente costoso se il numero di iperparametri da ottimizzare è grande.

La **RandomSearch**, d'altra parte, esplora casualmente le possibili combinazioni di iperparametri. Sebbene non garantisca una ricerca esaustiva come il GridSearch, è più veloce e può comunque identificare buone configurazioni per i parametri.

In questo progetto, sono stati utilizzati entrambi i metodi per ottimizzare gli iperparametri dei modelli e migliorare le loro prestazioni.