

Caricamento e preparazione del dataset

Obiettivo della sezione

Questa fase ha lo scopo di **importare correttamente il dataset principale** contenente le informazioni sugli studenti.

È un passaggio preliminare ma fondamentale, poiché da esso dipende la qualità dei dati che verranno analizzati nelle fasi successive.

Gli obiettivi specifici sono:

- leggere in modo sicuro il file `.csv` che contiene i dati grezzi;
- gestire eventuali righe corrotte o mal formattate;
- verificare che il numero di record letti coincida con quello atteso;
- garantire la corretta codifica dei caratteri (UTF-8).

Codice

Ecco la spiegazione riga per riga, rapida e chiara:

1. `import pandas as pd`

Importa la libreria **pandas** e la abbrevia come `pd`.

3-4. `# Lista per salvare eventuali righe problematiche / bad_lines = []`

Crea una lista vuota per memorizzare le righe del CSV che risultano corrotte/illeggibili.

6-9. Definizione funzione `bad_line_handler(line):`

- `def bad_line_handler(line):` → funzione di callback chiamata da `read_csv` quando incontra una riga "rotta".
- `bad_lines.append(line)` → aggiunge la riga problematica alla lista `bad_lines` (così puoi ispezionarla dopo).
- `return None` → dice a pandas di **saltare** quella riga. (Nota: con `on_bad_lines` un `None` significa "skippa", mentre potresti anche restituire una lista di campi corretti per provare a sostituire la riga.)

11-13. `# Caricamento del dataset` / chiamata a `pd.read_csv(...)`

- `"dataset_studenti_10k_union.csv"` → percorso/nome del file da leggere.
- `on_bad_lines=bad_line_handler` → collega la callback personalizzata per gestire righe non parseabili.
- `engine="python"` → obbliga l'uso del parser "python"; necessario se si vuole passare una **funzione** in `on_bad_lines`.
- `encoding="utf-8"` → specifica la codifica del file.
- Il risultato viene assegnato a `df`, un DataFrame.

1. `# Report di lettura` → commento introduttivo.

2. `print("Righe lette correttamente:", len(df))`

Stampa quante **righe valide** sono state caricate nel DataFrame.

3. `print("Righe problematiche:", len(bad_lines))`

Stampa quante righe sono state **scartate** perché problematiche.

18-19. `# Mostra i tipi di dato inferiti da Pandas / print("Tipi di dato rilevati da Pandas:\n")`

Messaggio descrittivo.

1. `print(df.dtypes)`

Stampa i **tipi di dato** (per colonna) che pandas ha inferito.

2. `print("\n_____\n")`

Stampa una linea di separazione estetica.

Se vuoi, posso anche mostrarti qualche statistica sulle righe scartate o salvare `bad_lines` su un CSV per analizzarle.

Scelta progettuale

- È stato scelto di **gestire manualmente le righe corrotte** tramite una funzione (`bad_line_handler`) invece di interrompere l'esecuzione, così da non perdere tempo in caso di pochi errori formali nel file.
- L'uso di `on_bad_lines=bad_line_handler` consente di **personalizzare la gestione** delle righe problematiche: vengono salvate in `bad_lines` e ignorate nel caricamento.
- È stato selezionato il motore **"python"**, più flessibile del **"c"**, per la sua **maggiore tolleranza** verso CSV non perfettamente formattati.
- L'**encoding "utf-8"** è stato esplicitamente impostato per garantire compatibilità con caratteri italiani e internazionali.

Note operative

- Se il dataset dovesse essere di grandi dimensioni o provenire da fonti eterogenee, conviene effettuare una **verifica preliminare del numero di colonne** e del **tipo di separatore**.
- In caso di errore di encoding, è possibile provare `"latin-1"` come alternativa a `"utf-8"`.
- Dopo la lettura, è buona norma controllare che la forma del DataFrame sia coerente:
`print(df.shape) # (righe, colonne)`
- Le righe problematiche vengono **saltate automaticamente**, ma restano **salvate in** `bad_lines` per eventuali analisi o correzioni successive.

Best practice

1. Specificare sempre un **encoding esplicito** (`utf-8`) per evitare problemi di compatibilità.
2. **Verificare dimensioni e tipi di dato** del DataFrame dopo la lettura (`df.shape`, `df.dtypes`).
3. **Gestire errori in modo controllato**: salvare le righe corrotte (`bad_lines`) e saltarle senza interrompere l'esecuzione.
4. **Conservare una copia del file originale** prima di modifiche o pulizia.

5. **Stampare un report sintetico:** numero di righe lette, righe problematiche e tipi di dato rilevati da Pandas.

Risultato atteso

Al termine dell'esecuzione, viene generato un DataFrame `df` (es. **10.000 righe lette correttamente e nessuna riga problematica**).

Questo conferma che il dataset è **integro e pronto** per le fasi successive di analisi esplorativa e preprocessing.

Righe lette correttamente: 10000
Righe problematiche: 0

 Tipi di dato rilevati da Pandas:

Eta	int64
Genere	object
Assenze	int64
Voti_medi	float64
Ripetenze	int64
Cambi_scuola	int64
ISEE	float64
Lavoro_genitori	object
Titolo_studio_genitori	object
Partecipazione_progetti	int64
Distanza_scuola	object
Autovalutazione	object
Ammesso	int64
dtype:	object

Preprocessing delle variabili categoriali

Obiettivo della sezione

La funzione `preprocess_data()` prepara il dataset per l'**addestramento del modello di machine learning**, convertendo automaticamente le **variabili categoriche** in **valori numerici** interpretabili dagli algoritmi, mantenendo intatte le colonne già numeriche ed **escludendo il target** (`Ammesso`).

Il *preprocessing* è essenziale perché:

- garantisce **coerenza** tra training e predizione;
- gestisce in modo sicuro **valori mancanti** e **categorie nuove**;
- assicura la **riproducibilità** del flusso di lavoro.

L'`OrdinalEncoder` di *scikit-learn* viene configurato per:

- assegnare automaticamente un **codice numerico** a ogni categoria;
- gestire **valori sconosciuti** tramite `unknown_value=-1`;
- restituire un **DataFrame completamente numerico** pronto per l'addestramento.

Alla fine dell'esecuzione, la funzione restituisce:

- `df_proc` → il **DataFrame preprocessato**;
- `encoder` → l'**oggetto fittato**, da salvare insieme al modello per garantire coerenza tra training e inferenza.

Codice:

- `from sklearn.preprocessing import OrdinalEncoder`

Importa l'encoder ordinale di scikit-learn.

- `import pandas as pd`

Importa pandas.

- `def preprocess_data(df, target="Ammesso", verbose=True):`

Definisce la funzione di preprocessing. Parametri: DataFrame, nome colonna target da escludere, flag per messaggi.

- Docstring `""" ... """`

Spiega cosa fa: converte "Autovalutazione", codifica categoriche, esclude il target, ritorna df numerico + encoder fittato.

- `df_proc = df.copy()`

Lavora su una **copia** per non modificare l'originale.

1) Conversione "Autovalutazione"

- `if "Autovalutazione" in df_proc.columns:`

Procede solo se la colonna esiste.

- `df_proc["Autovalutazione"] = pd.to_numeric(df_proc["Autovalutazione"], errors="coerce")`

Converte a numerico; valori non convertibili → **NaN** (errors="coerce").

- `if verbose: print("✓ ...")`

Messaggio informativo se `verbose=True`.

2) Individua colonne categoriche (escludendo il target)

- `cat_cols = [...]` con comprensione di lista

Seleziona le colonne **non numeriche** (`not pd.api.types.is_numeric_dtype(...)`) diverse da `target`.

- `if verbose: print(f"Colonne categoriche da codificare: {cat_cols}")`

Log delle colonne trovate.

3) Ordinal encoding (robusto a categorie nuove)

- `encoder = None`

Placeholder.

- `if cat_cols:`

Solo se ci sono categoriche:

- `encoder = OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=-1)`

Crea l'encoder: per **categorie non viste** in test userà il valore **-1** (invece di lanciare errore).

- `df_proc[cat_cols] = encoder.fit_transform(df_proc[cat_cols].astype(str))`

Converte prima a **stringa** (evita mix tipi/NaN), poi **fit+transform** e sostituisce le colonne con i codici ordinali (float).

- `if verbose: print(f"✓ Codificate {len(cat_cols)} ...")`

Log post-encoding.

4) Verifica finale

- `non_num = df_proc.select_dtypes(exclude="number").columns.tolist()`

Controlla se restano colonne non numeriche.

- `if non_num: print("⚠️ ...", non_num)`

Avvisa se qualcosa non è numerico.

- `else: if verbose: print("✓ Tutte le feature sono numeriche ...")`

Conferma che tutto (tranne il target) è numerico.

Ritorno

- `return df_proc, encoder`

Restituisce il DataFrame preprocessato e l'encoder fittato (utile per trasformare dati futuri).

Esempio di chiamata (fuori funzione)

- `df_prep, enc = preprocess_data(df, target="Amnesso", verbose=True)`

Esegue il preprocessing sul `df`, esclude la colonna `Amnesso` dal trattamento, stampa i log e ottiene `df_prep` + `enc`.

Note rapide

- `errors="coerce"` → stringhe/valori invalidi diventano NaN: gestiscili poi (imputazione).
- `unknown_value=-1` → in produzione, le categorie nuove verranno codificate a -1: ricordati di gestirlo nel modello.

Scelte progettuali

- **Ordinal vs One-Hot:** `OrdinalEncoder` → più **compatto** (meno colonne), ideale per modelli ad alberi/boosting. Per modelli **lineari**, considerare One-Hot.
- **Valori sconosciuti:** `handle_unknown="use_encoded_value", unknown_value=-1` → niente crash in produzione se arrivano nuove categorie.
- **Robustezza tipi:** `astype(str)` prima del fit evita problemi con mix di tipi/NaN.
- **Immutabilità:** `df.copy()` per evitare side-effects.
- **Controllo:** `verbose` per log leggibili solo quando serve.
- **Target:** escluso dal preprocessing (non si trasforma `Ammesso`).

Esempio d'uso consigliato

```
from sklearn.model_selection import train_test_split

df_prep, enc = preprocess_data(df, target="Ammesso", verbose=False)

X = df_prep.drop(columns=["Ammesso"], errors="ignore") # se il target era già fuori, non fa nulla
y = df["Ammesso"]                                     # target dall'originale

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, stratify=y, random_state=42
)
```

Limitazioni e note operative

- **Ordine artificiale:** i codici 0/1/2... non hanno significato intrinseco → ok per alberi, attenzione con lineari.
- **NaN:** con `astype(str)`, i NaN diventano `"nan"` e sono trattati come **categoria esplicita** (coerente con `unknown_value`).
- **Coerenza schema:** in produzione usa **lo stesso encoder** e **lo stesso set/ordine di colonne** della fase di fit.
- **Log:** se `verbose=True`, stampa conferme su conversioni, colonne codificate, controlli finali.

Best practice — *Preprocessing categoriche → numeriche*

1. **Immutabilità dei dati:** usa `df.copy()` per evitare modifiche al dataset originale.
2. **Selezione automatica:** individua le colonne categoriche con `is_numeric_dtype`, senza hard-code.

3. **Valori sconosciuti:** gestisci nuove categorie con `unknown_value=-1` per evitare crash in produzione.
 4. **NaN coerenti:** converti a `str` prima dell'encoding per trattare i mancanti come categoria esplicita.
 5. **Target escluso:** non trasformare la colonna obiettivo (`Ammesso`).
 6. **Riproducibilità:** fissa `random_state` e salva le versioni delle librerie.
 7. **Serializzazione unica:** salva modello ed encoder insieme in un unico `.pkl` .
 8. **Classi sbilanciate:** usa metriche robuste (F1, ROC-AUC) o `class_weight="balanced"` .
 9. **Explainability:** usa `permutation_importance` o SHAP per interpretare le predizioni.
 10. **Controllo finale:** verifica che dopo il preprocessing restino solo feature numeriche.
-

Serializzazione (una riga)

- Salva insieme modello e encoder in un unico file `.pkl` :

```
import joblib
joblib.dump({"model": clf, "encoder": enc}, "modello_pipeline.pkl")
```

- Garantisce che in backend venga applicato **lo stesso preprocessing** del training.
- Agevola **deploy, versioning e rollback**.

Uso in backend (FastAPI) – inferenza coerente

```
import joblib, pandas as pd
pipe = joblib.load("modello_pipeline.pkl")

def predici(json_input: dict):
    X_new = pd.DataFrame([json_input])
    proba = float(pipe.predict_proba(X_new)[0, 1])
    rischio = "ALTO" if proba > 0.70 else "MEDIO" if proba > 0.40 else "BASSO"
    return {"prob_abbandonamento": round(proba, 2), "livello_rischio": rischio}
```

Note operative

- La **Pipeline** garantisce che la trasformazione delle feature sia identica tra training e produzione.
 - La lista `cat_cols` viene **apprendida** dal pipeline al fit; se arrivano categorie nuove, l' `unknown_value=-1` ne permette la gestione sicura.
 - Un singolo file `modello_pipeline.pkl` semplifica deploy, versioning e rollback.
-

Pulizia base e controllo del target

Obiettivo della sezione

Eseguire i controlli preliminari sul dataset per garantire qualità e coerenza prima del preprocessing e del training:

- normalizzare i nomi colonna;
- verificare la presenza del **target** ('Ammesso');
- controllare dimensioni, duplicati e valori mancanti;
- produrre statistiche descrittive solo per le **feature numeriche**;
- visualizzare la distribuzione del target (conteggi + percentuali).

Codice

```
import pandas as pd
```

Importa **pandas**, usato per ispezione e manipolazione dei dati.

```
# 1) Normalizza i nomi colonna
df.columns = df.columns.str.strip()
```

Rimuove spazi iniziali/finali dai nomi delle colonne per evitare errori di riferimento.

```
# 2) Verifica presenza del target
target = "Ammesso"
if target not in df.columns:
    raise KeyError(f"Target '{target}' non trovato. Colonne disponibili: {list(df.columns)}")
```

Controlla che la variabile obiettivo esista nel DataFrame; in caso contrario **interrompe** con un messaggio chiaro.

```
# 3) Controlli di base: shape, duplicati, NA totali
dup = df.duplicated().sum()
na_tot = int(df.isna().sum().sum())
print(f"Shape: {df.shape} | Duplicati: {dup} | Valori mancanti (tot): {na_tot}")
```

Riepiloga **dimensioni**, numero di righe **duplicate** e totale **NaN**.

```
# 4) Statistiche descrittive (solo numeriche)
num_cols = df.select_dtypes(include=["number"]).columns
display(df[num_cols].describe().T)
```

Seleziona solo le colonne **numeriche** (compatibile con tutte le versioni di pandas) e mostra le statistiche principali trasposte per leggibilità.

```
# 5) Distribuzione del target (conteggio + %)
vc = df[target].value_counts(dropna=False)
```



```
pct = (vc / vc.sum() * 100).round(2)
display(pd.DataFrame({"count": vc, "%": pct}))
```

Mostra **conteggi** e **percentuali** della variabile target, includendo eventuali `NaN`.

Scelta progettuale

- Si lavora **in-place** sul DataFrame già caricato, mantenendo la cella semplice e veloce.
- Si evita l'uso di parametri non supportati da versioni datate di pandas (`numeric_only=True`), optando per una **selezione esplicita** delle colonne numeriche.
- La verifica del target è **bloccante**: meglio fermarsi subito che propagare errori nelle fasi successive.
- Output sintetico e leggibile: un'unica riga riassume **shape, duplicati e NaN**; le tabelle sono essenziali per il controllo visivo.

Note operative

- Se `dup > 0`, valutare rimozione con `df = df.drop_duplicates()`.
- Se `na_tot > 0`, annotare dove si concentrano i mancanti con `df.isna().sum().sort_values(ascending=False).head()`.
- Se il target risulta sbilanciato, annotarlo per la scelta delle **metriche** (Precision/Recall/F1, PR curve) e per eventuale **class_weight** o **SMOTE** (solo sul train).

Best practice

1. **Pulizia dei nomi colonna** subito dopo il load (evita bug silenziosi).
2. **Fail fast** sulla presenza del target: interrompere con messaggio chiaro.
3. **Controlli minimi standard** (shape, duplicati, NaN) in un'unica riga di log.
4. **Statistiche solo numeriche** per report coerenti e ripetibili.
5. **Distribuzione del target** sempre riportata (conteggio + %) per valutare lo sbilanciamento.
6. **Opzionale**: warning se il target non è binario, utile per intercettare errori di codifica.

Correlazioni e relazione col target

Obiettivo della sezione

L'obiettivo di questa fase è analizzare la **relazione tra le variabili numeriche del dataset** e la variabile target 'Ammesso'.

L'analisi di correlazione consente di:

- valutare **quanto fortemente** ciascuna variabile è legata al target;
- individuare eventuali **ridondanze** tra variabili (multicollinearità);
- comprendere in modo preliminare **quali feature** potrebbero influire maggiormente sulla predizione finale.

Codice

Import

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Importa le librerie principali per analisi dati e visualizzazione.

Target e colonne numeriche

```
target = "Ammesso"
num_cols = df.select_dtypes(include=["number"]).columns
if target not in num_cols:
    raise ValueError(f"Il target '{target}' non è numerico o non esiste ...")
```

- Seleziona solo le colonne numeriche dal DataFrame.
- Controlla che la colonna `Ammesso` sia numerica; altrimenti interrompe l'esecuzione.

1 Matrice di correlazione

```
corr = df[num_cols].corr()
```

Calcola la **correlazione di Pearson** tra tutte le variabili numeriche.

2 Ordinamento in base al target

```
order = (
    corr[target]
    .drop(target)
    .abs()
    .sort_values(ascending=False)
```

```
.index  
)
```

- Ordina le feature per **correlazione assoluta** con il target (dalla più forte alla più debole).
- Esclude il target stesso.

3 Riordina la matrice e prepara i dati per il barplot

```
ordered = list(order) + [target]  
corr_ord = corr.loc[ordered, ordered]
```

Riordina righe e colonne della matrice per mostrare il target alla fine.

```
top_corr = (  
    corr[target]  
    .drop(target)  
    .abs()  
    .sort_values(ascending=False)  
    .to_frame("correlazione_assoluta")  
)
```

Crea una tabella con la correlazione assoluta rispetto al target (puoi usare `.head(8)` per limitarla).

4 Figura combinata: heatmap + barplot

```
fig, axes = plt.subplots(1, 2, figsize=(14, 6))
```

Crea due grafici affiancati.

◆ Heatmap (matrice di correlazione)

```
mask = np.triu(np.ones_like(corr_ord, dtype=bool))  
sns.heatmap(  
    corr_ord, mask=mask, ax=axes[0],  
    annot=True, fmt=".2f", cmap="coolwarm",  
    vmin=-1, vmax=1, center=0, square=True,  
    cbar_kws={"shrink": 0.8, "label": "Coefficiente di correlazione"},  
    linewidths=0.5, linecolor="white"  
)  
axes[0].set_title("Matrice di correlazione (ordinata sul target)", fontsize=13, pad=10)  
plt.setp(axes[0].get_xticklabels(), rotation=45, ha="right")
```

- Mostra solo il triangolo inferiore della matrice.
- Evidenzia visivamente le correlazioni tra feature e target.
- Colori da blu (negativo) a rosso (positivo).

◆ Barplot (forza di correlazione col target)

```
sns.barplot(
    x="correlazione_assoluta",
    y=top_corr.index,
    data=top_corr,
    palette="crest",
    ax=axes[1],
)
axes[1].set_xlabel("Correlazione assoluta")
axes[1].set_title(f"Forza della correlazione con il target '{target}'", fontsize=13, pad=10)
```

Mostra graficamente **quali feature** hanno maggiore correlazione (in valore assoluto) con il target.

Layout finale

```
plt.tight_layout()
plt.show()
```

Ottimizza la spaziatura e visualizza i grafici.

🔍 In sintesi

Questo script:

1. Calcola la correlazione tra tutte le variabili numeriche.
2. Ordina le feature in base alla correlazione assoluta con il target (**Ammesso**).
3. Visualizza **due grafici complementari**:
 - una **heatmap** ordinata sul target,
 - un **barplot** che mostra le feature più correlate.

È ideale per un'analisi esplorativa mirata alla **selezione di feature rilevanti** per modelli di machine learning.

Scelte progettuali

- **Pearson correlation coefficient**: usato perché tutte le feature sono numeriche dopo il preprocessing.
- **Valori assoluti**: valutata la forza della relazione, non il segno, per includere correlazioni positive e negative.
- **Visualizzazione combinata**: heatmap per visione globale, barplot per lettura mirata sul target.
- **Color map "coolwarm"**: facilita la distinzione tra correlazioni positive (rosso) e negative (blu).

Note operative

- **Correlazioni deboli** ($|r| < 0.3$) possono essere comunque utili in modelli non lineari.

- L'**assenza di correlazioni forti** è un buon segno: il dataset non è ridondante.
 - Una correlazione **negativa** (es. *Assenze–Ammesso*) indica che più assenze riducono la probabilità di ammissione.
 - Una correlazione **positiva** (es. *Voti_medi–Ammesso*) mostra che voti più alti aumentano la probabilità di successo.
-

Best practice

- Analizza la **forza**, non solo il segno, della correlazione.
 - Rimuovi variabili **fortemente correlate** ($|r| > 0.8$) per evitare multicollinearità.
 - Usa **heatmap e barplot** insieme per analisi globale e mirata.
 - Non eliminare feature con correlazioni basse senza prima testarle nel modello.
 - Annota possibili **variabili proxy** (es. *assenze, partecipazione*).
-

Considerazioni sui grafici

Dall'analisi dei grafici emergono alcune evidenze chiare:

- **Assenze** è la variabile con **maggiore correlazione negativa** con il target **Ammesso** ($r \approx -0.59$):
→ gli studenti con molte assenze tendono a non essere ammessi.
- **Voti_medi** mostra una **correlazione positiva moderata** ($r \approx +0.29$):
→ voti più alti sono associati a una maggiore probabilità di ammissione.
- **Ripetenze** presenta una correlazione **debole ma negativa**, suggerendo che chi ha ripetuto tende leggermente a non essere ammesso.
- **Partecipazione_progetti, ISEE, Età e Cambi_scuola** mostrano correlazioni **molto deboli** ($|r| < 0.1$):
→ non influenzano direttamente l'esito, ma possono avere un effetto combinato o indiretto.
- L'assenza di correlazioni forti tra le feature indipendenti indica **assenza di multicollinearità** e quindi un dataset ben bilanciato per modelli multivariati.

In sintesi:

Le variabili **Assenze** e **Voti_medi** risultano i principali predittori del target **Ammesso**.

Questo conferma che **la frequenza scolastica e il rendimento** sono i fattori più influenti nella probabilità di successo e ammissione.

Analisi dei boxplot – Confronto tra classi

Obiettivo dell'analisi

L'obiettivo di questa fase è analizzare le **distribuzioni statistiche delle variabili numeriche** del dataset *Abbandono scolastico* in funzione della variabile target **Ammesso**

(0 = non ammesso, 1 = ammesso).

L'uso dei **boxplot** consente di visualizzare:

- la **distribuzione dei valori centrali** (mediana);
- la **variabilità** dei dati (intervallo interquartile, IQR);
- la presenza di **outlier**;
- le **differenze tra gruppi** (ammessi vs non ammessi).

Questo tipo di grafico è particolarmente utile per individuare pattern di disuguaglianza o fattori associati al rendimento scolastico.

Spiegazione del codice

1. Importazione librerie

```
import matplotlib.pyplot as plt
import seaborn as sns
```

Si importano le librerie per la visualizzazione grafica.

2. Impostazione dello stile

```
sns.set(style="whitegrid")
```

Applica uno stile pulito e leggibile con griglia di riferimento, utile per confronti di distribuzioni.

3. Selezione variabili numeriche

```
num_cols = [...]
```

Si definisce la lista delle colonne numeriche da visualizzare.

In questo caso: *Assenze*, *Voti_medi*, *ISEE*, *Autovalutazione*.

4. Creazione del layout

```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
```

Si genera una griglia 2×2 per disporre quattro boxplot in un'unica figura.

5. Ciclo di disegno

```
for ax, col in zip(axes.flatten(), num_cols):  
    sns.boxplot(...)
```

- Cicla su ciascuna variabile e crea il boxplot.
- `x="Ammesso"` definisce la variabile di confronto tra classi (target binario).
- `y=col` indica la variabile numerica da rappresentare.
- `palette` assegna un colore diverso per ogni classe (rosso = non ammesso, verde = ammesso).

6. Etichette e titoli

```
ax.set_title(col)  
ax.set_xlabel("Ammesso (0=No, 1=Si)")  
ax.set_ylabel("Valore")
```

Personalizza il titolo e le etichette di ogni grafico.

7. Titolo complessivo e layout


```
fig.suptitle(...)  
plt.tight_layout()  
plt.show()
```

Aggiunge un titolo generale, ottimizza la disposizione e mostra i grafici.

Interpretazione dei risultati


Assenze

- Gli studenti **non ammessi** presentano un numero di assenze molto più alto (mediana > 40).
- Gli **ammessi** mostrano una distribuzione più concentrata tra 10 e 25 giorni.

 *Conclusione:* le assenze sono un indicatore chiave di rischio scolastico.


Voti medi

- Gli **ammessi** hanno una mediana dei voti intorno a **7**, con scarsa variabilità.
- I **non ammessi** mostrano una mediana più bassa (**≈ 5**) e ampia dispersione.

 *Conclusione:* la media dei voti è il predittore più forte dell'ammissione.


ISEE

- Le due classi presentano distribuzioni simili, con lievi differenze a favore degli ammessi.
- Si notano outlier per valori ISEE molto bassi (<5.000 €).

 *Conclusione:* la condizione economica influisce solo marginalmente.

Autovalutazione

- Gli **ammessi** dichiarano punteggi medi più alti (≈ 4), mentre i **non ammessi** tendono a valori più bassi ($\approx 2-3$).
- Le due distribuzioni risultano ben separate.

 *Conclusione:* la percezione di sé e la motivazione influenzano fortemente la probabilità di successo.

Sintesi generale

Variabile	Influenza sull'ammissione	Interpretazione
Voti_med	Alta	Miglior rendimento = maggiore probabilità di ammissione.
Assenze	Alta	Maggior numero di assenze = rischio elevato di non ammissione.
Autovalutazione	Media	La fiducia percepita influisce sulla motivazione e il rendimento.
ISEE	Bassa	Impatto economico secondario ma coerente con i trend sociali.

Conclusione

I boxplot confermano e rafforzano:

- Le differenze tra ammessi e non ammessi sono più marcate per **voti medi** e **assenze**;
- **Autovalutazione** aggiunge una componente psicologica importante;
- Le variabili economiche e anagrafiche hanno un impatto minore.

Nel complesso, i grafici descrivono un quadro coerente con le relazioni attese e con le performance ottenute dai modelli predittivi successivi, validando ulteriormente la qualità del dataset e la logica dei risultati ottenuti.

Analisi degli istogrammi per classe

Obiettivo dell'analisi

Questa sezione ha l'obiettivo di analizzare graficamente le distribuzioni delle variabili numeriche del dataset *Abbandono scolastico*, confrontando la densità dei valori tra le due classi della variabile target **Ammesso**:

- **0 = Non ammesso,**
- **1 = Ammesso.**

L'uso di istogrammi sovrapposti permette di valutare **differenze di forma, mediana e dispersione** tra i gruppi, aiutando a identificare i fattori maggiormente associati all'esito scolastico.

Spiegazione del codice

1. Importazione libreria

```
import matplotlib.pyplot as plt
```

Si importa la libreria di visualizzazione `matplotlib` per generare i grafici.

2. Definizione variabili numeriche

```
num_cols = [...]
```

Si crea una lista contenente i nomi delle colonne numeriche da confrontare.

3. Creazione del layout

```
fig, axes = plt.subplots(3, 3, figsize=(14, 8))
```

Si definisce una griglia di sottografi (subplot) con 3 righe e 3 colonne, per disporre i vari istogrammi.

4. Appiattimento dell'array di assi

```
axes = axes.flatten()
```

Converte la griglia 2D di assi in una lista 1D, per iterarci facilmente sopra nel ciclo `for`.

5. Ciclo sulle variabili

```
for i, col in enumerate(num_cols):
```

Si scorre ogni variabile numerica della lista.

6. Disegno degli istogrammi

```
axes[i].hist(df[df["Ammesso"] == 0][col], ...)
```

```
axes[i].hist(df[df["Ammesso"] == 1][col], ...)
```

- Si disegnano due istogrammi sovrapposti nello stesso asse:
 - **rosso** per gli studenti non ammessi,
 - **verde** per gli studenti ammessi.
- `density=True` normalizza le frequenze per rappresentare la distribuzione in termini di densità.
- `alpha` regola la trasparenza per visualizzare entrambe le curve.

7. Titoli e assi

```
axes[i].set_title(col)  
axes[i].set_ylabel("Densità")
```

Ogni grafico riporta il nome della variabile e l'etichetta verticale.

8. Legenda e titolo generale

```
axes[0].legend(title="Ammesso")  
fig.suptitle(...)
```

Si aggiunge una legenda condivisa e un titolo complessivo.

9. Ottimizzazione layout e visualizzazione

```
plt.tight_layout()  
plt.show()
```

Ottimizza la spaziatura tra i grafici e mostra la figura finale.

Interpretazione dei risultati

Età

Distribuzione uniforme in entrambe le classi. L'età non risulta discriminante rispetto all'esito di ammissione.

Assenze

I non ammessi presentano una densità elevata per oltre **40 assenze**, mentre gli ammessi sono concentrati sotto le **30**.

- Le assenze rappresentano un **fattore di rischio diretto** di non ammissione.

Voti medi

Distribuzione ben separata:

- Ammessi → voti centrati su **7-8**
- Non ammessi → picco tra **4-6**
 - Il rendimento scolastico è il **predittore più forte** dell'esito positivo.

Ripetenze

La maggior parte degli studenti non ha ripetuto, ma nei non ammessi si osserva un incremento nei valori 1–2.

- Le ripetenze sono un **fattore cumulativo di rischio**.

Cambi scuola

Distribuzione concentrata su 0, ma i non ammessi mostrano più casi con 1–2 cambi.

- Segnale di **instabilità o difficoltà di adattamento**.

ISEE

Distribuzioni quasi sovrapposte, con valori medi leggermente inferiori nei non ammessi.

- L'impatto economico è **debole ma coerente**.

Partecipazione progetti

Distribuzione simile tra i gruppi, con leggera tendenza a una maggiore partecipazione tra gli ammessi.

- Variabile **moderatamente correlata** all'ammissione.
-

Conclusione interpretativa

L'analisi degli istogrammi conferma che le variabili **Voti_medi**, **Assenze** e **Ripetenze** sono le più rilevanti nel discriminare gli esiti scolastici.

In particolare:

- studenti **ammessi** → rendimento più alto, poche assenze, percorso stabile;
- studenti **non ammessi** → rendimento basso, numerose assenze, maggiore discontinuità.

Questi risultati forniscono una prima validazione visiva del dataset, evidenziando pattern coerenti che giustificano le performance ottenute dai modelli predittivi nelle fasi successive.

Suddivisione del dataset in Training e Test set + Verifica grafica

Obiettivo della sezione

Questa fase serve a suddividere il dataset preprocessato in due sottoinsiemi distinti:

- **Training set (80%)** → utilizzato per addestrare i modelli di machine learning.
- **Test set (20%)** → utilizzato per valutare le prestazioni del modello su dati mai visti.

La divisione è **stratificata rispetto alla variabile target 'Ammesso'**, in modo da mantenere la stessa proporzione di classi (ammessi e non ammessi) in entrambi i set.

Inoltre, viene creato un **grafico a barre comparativo** che consente di verificare visivamente la corretta stratificazione.

Codice:

```
from sklearn.model_selection import train_test_split
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

Importa le librerie necessarie:

- `train_test_split` per la suddivisione dei dati;
- `pandas` per la manipolazione del DataFrame;
- `seaborn` e `matplotlib` per la visualizzazione grafica.

```
TARGET = "Ammesso"
```

Definisce il nome della variabile target, che rappresenta la classe da predire (1 = ammesso, 0 = non ammesso).

```
X = df_prep.drop(columns=[TARGET])
y = df_prep[TARGET].astype(int)
```

Separa le feature indipendenti (`x`) dalla variabile target (`y`), assicurandosi che quest'ultima sia di tipo numerico intero.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)
```

Divide i dati in **80% training** e **20% test**.

L'argomento `stratify=y` mantiene la stessa proporzione di classi nei due set, evitando squilibri.

`random_state=42` garantisce la riproducibilità dello split.

```
print("✅ Split completato")
print(f"• X_train: {X_train.shape} | X_test: {X_test.shape}")
```

Stampa un piccolo report con le dimensioni di ciascun set.

```
train_df = pd.DataFrame({"Ammesso": y_train, "Set": "Training"})
test_df = pd.DataFrame({"Ammesso": y_test, "Set": "Test"})
dist_df = pd.concat([train_df, test_df])
```

Crea un unico DataFrame (`dist_df`) che unisce i due insiemi, utile per la visualizzazione grafica della distribuzione delle classi.

```
plt.figure(figsize=(6, 4))
sns.countplot(
    data=dist_df,
    x="Ammesso",
    hue="Set",
    palette={"Training": "#77dd77", "Test": "#84b6f4"},
    width=0.6
)
```

Disegna un **grafico a barre** comparativo della distribuzione del target nei due insiemi:

- Verde → Training set
- Blu → Test set

```
plt.title("Distribuzione della variabile target nei due set", fontsize=13)
plt.xlabel("Ammesso (0 = No, 1 = Si)")
plt.ylabel("Numero di studenti")
plt.legend(title="Set", loc="upper right")
plt.tight_layout()
plt.show()
```

Aggiunge titolo, etichette e legenda per una visualizzazione chiara e completa.

Scelte progettuali

- **Stratificazione obbligatoria** (`stratify=y`) per evitare che una classe sia sovra- o sotto-rappresentata nel test set.
- **Suddivisione 80/20**: equilibrio standard tra addestramento e valutazione.
- **Visualizzazione comparativa**: il grafico consente di controllare immediatamente che i due set mantengano le stesse proporzioni di classi.
- **Colori differenziati** (`#77dd77` e `#84b6f4`) per distinguere chiaramente Training e Test.

Note operative

- Se i dataset risultano molto sbilanciati, la visualizzazione grafica è essenziale per accorgersi di eventuali anomalie.
 - È consigliato fissare `random_state` per assicurare risultati riproducibili in più esperimenti.
 - Questo tipo di visualizzazione può essere riutilizzato anche per verificare **altri split** (ad esempio in cross-validation).
-

Best practice

1. **Effettuare lo split dopo il preprocessing**, mai prima, per evitare leakage di informazioni.
 2. **Usare la stratificazione** ogni volta che si lavora con un target binario o multilabel.
 3. **Verificare graficamente** la distribuzione delle classi: è il modo più rapido per validare lo split.
 4. **Salvare gli insiemi ottenuti** (`X_train` , `X_test` , `y_train` , `y_test`) se verranno usati in più notebook o esperimenti.
 5. **Annotare i parametri di split** (test size, seed, stratify) nella documentazione per garantire tracciabilità e replicabilità.
-

Considerazioni sulla scalatura dei dati

In questa fase è stata valutata la necessità di applicare la scalatura delle variabili numeriche per rendere i dati confrontabili tra loro.

In generale, la scalatura è utile quando le variabili hanno ordini di grandezza molto diversi — ad esempio nel caso di feature come "Assenze" (con valori da 0 a 60) e "ISEE" (che può arrivare a decine di migliaia).

Tuttavia, la reale utilità di questa operazione dipende dal tipo di modello utilizzato per l'addestramento.

Nel progetto Union, il dataset viene utilizzato per addestrare modelli basati su **alberi decisionali**, come Random Forest, Gradient Boosting e XGBoost.

Questi algoritmi non fanno uso di distanze tra punti né di funzioni di costo sensibili alla scala, ma operano tramite **soglie di split indipendenti dal valore assoluto delle variabili**.

Di conseguenza, modificare la scala dei dati non comporta alcun beneficio in termini di accuratezza o stabilità numerica.

Applicare una scalatura in questo contesto risulterebbe quindi inutile, in quanto:

- non migliorerebbe le prestazioni del modello;
- aumenterebbe i tempi di preprocessing;
- potrebbe introdurre rumore sulle variabili categoriali codificate.

Per questi motivi, si è deciso **di non applicare alcuna trasformazione di scaling**.

Quando la scalatura sarebbe utile

La normalizzazione o standardizzazione dei dati è invece consigliata quando si utilizzano modelli sensibili alla scala delle feature, come:

- Regressione logistica;
- Support Vector Machines (SVM);
- K-Nearest Neighbors (KNN);
- PCA o LDA;
- Reti neurali (MLP).

In questi casi la scalatura serve a evitare che feature con valori più grandi dominino l'ottimizzazione, migliorando la convergenza dell'algoritmo.

Al contrario, per modelli basati su alberi (Decision Tree, Random Forest, XGBoost, LightGBM), la scalatura non è richiesta poiché tali modelli **sono invarianti rispetto alla scala numerica delle feature**.

Decisione progettuale

Alla luce delle considerazioni precedenti, nel progetto Union non è stato applicato alcun processo di scalatura.

Le feature numeriche vengono quindi utilizzate con i loro valori originali, coerenti con la scala reale di riferimento (es. numero di assenze, età, reddito ISEE).

Questa scelta garantisce un flusso di preprocessing più semplice, trasparente e facilmente riproducibile.

Best practice

- Applicare la scalatura solo quando strettamente necessaria, in base al tipo di algoritmo scelto.
- Non scalare variabili categoriali codificate numericamente (es. genere o titolo di studio).
- Documentare sempre la scelta, anche in caso di omissione.
- In caso di confronto tra modelli lineari e tree-based, valutare eventualmente una scalatura selettiva solo sulle variabili numeriche pure.

Funzione di valutazione grafica del modello

9) Analisi di un singolo modello

Questa funzione ha lo scopo di **valutare in modo completo e visivo le prestazioni di un modello di classificazione binaria**.

Prende in input due vettori:

- `y_true` : le etichette reali (0 = non ammesso, 1 = ammesso);
- `y_proba` : le probabilità predette dal modello per la classe positiva.

Il codice inizia con una **fase di pulizia dei dati**:

```
y_true, y_proba = map(lambda x: np.asarray(x, dtype=float), (y_true, y_proba))
mask = ~np.isnan(y_true) & ~np.isnan(y_proba)
y_true, y_proba = y_true[mask], y_proba[mask]
```

Qui vengono convertiti gli input in array numerici (`numpy`) e rimossi eventuali valori mancanti (`NaN`).

Questo garantisce che le metriche successive non generino errori o risultati incoerenti.

Segue la **trasformazione delle probabilità in predizioni binarie** tramite una soglia (di default 0.5):

```
y_pred = (y_proba >= threshold).astype(int)
```

In questo modo, le probabilità vengono convertite in etichette 0/1, permettendo di calcolare precisione, recall e F1-score.

Le **metriche principali** vengono calcolate in un unico blocco ordinato:

```
metrics = dict(
    accuracy = accuracy_score(y_true, y_pred),
    precision = precision_score(y_true, y_pred, zero_division=0),
    recall = recall_score(y_true, y_pred),
    f1 = f1_score(y_true, y_pred),
    roc_auc = roc_auc_score(y_true, y_proba),
    pr_auc = average_precision_score(y_true, y_proba),
    brier = brier_score_loss(y_true, y_proba)
)
```

Vengono così ottenuti gli indicatori fondamentali:

- **Accuracy** – percentuale di predizioni corrette.
- **Precision** – percentuale di ammessi correttamente previsti.
- **Recall** – capacità del modello di individuare tutti gli ammessi.
- **F1-score** – media armonica di precisione e recall.

- **ROC-AUC e PR-AUC** – valutano la qualità complessiva del classificatore indipendentemente dalla soglia.
- **Brier score** – misura quanto le probabilità predette sono ben calibrate rispetto alle osservazioni reali.

Successivamente vengono generati **tre grafici principali** che riassumono visivamente il comportamento del modello:

1. **Curva ROC** – mostra il compromesso tra tasso di veri positivi e falsi positivi, con l'area (AUC) indicata in legenda.
2. **Curva Precision-Recall** – utile in presenza di dataset sbilanciati, rappresenta la precisione in funzione del recall.
3. **Curva di calibrazione** – confronta la probabilità predetta con la probabilità reale, indicando se il modello è ben calibrato (una curva perfetta coincide con la diagonale $y = x$).

Ogni grafico è generato in modo compatto tramite le funzioni di visualizzazione di scikit-learn:

```
RocCurveDisplay.from_predictions(...)
PrecisionRecallDisplay.from_predictions(...)
calibration_curve(...)
```

Dopo i grafici, la funzione visualizza anche la **matrice di confusione**, utile per osservare la distribuzione dei veri e falsi positivi/negativi.

Infine, stampa un riepilogo sintetico con tutte le metriche calcolate e restituisce un dizionario con i valori e il vettore `y_pred`.

10) Analisi comparativa multipla

Questa funzione è stata progettata per **confrontare in modo diretto e visivo più modelli di classificazione**.

Accetta come input:

- `y_true` : le etichette reali;
- `proba_dict` : un dizionario in cui le chiavi sono i nomi dei modelli e i valori sono i vettori di probabilità per la classe positiva.

La funzione è strutturata per creare **una figura composta da tre grafici verticali** (ROC, Precision-Recall e Calibrazione), seguiti da una tabella comparativa delle metriche.

Il codice inizia convertendo le etichette in array numerici e impostando la struttura della figura:

```
fig, axes = plt.subplots(3, 1, figsize=(8, 14))
fig.suptitle(title, fontsize=14)
```

Quindi entra in un ciclo che scorre su ciascun modello:

```
for name, p in proba_dict.items():
    p = np.asarray(p, dtype=float)
```

```
mask = ~np.isnan(y_true) & ~np.isnan(p)
y, p = y_true[mask], p[mask]
```

Ogni vettore viene pulito da eventuali `NaN` e utilizzato per calcolare le metriche fondamentali:

```
roc = roc_auc_score(y, p)
pr = average_precision_score(y, p)
brier = brier_score_loss(y, p)
rows.append({"Modello": name, "ROC-AUC": roc, "PR-AUC": pr, "Brier": brier})
```

I tre grafici vengono aggiornati progressivamente all'interno dello stesso ciclo:

- **ROC curve:** tramite `RocCurveDisplay.from_predictions()` vengono sovrapposte tutte le curve dei modelli in un unico pannello.
- **Precision-Recall curve:** mostra la capacità di ogni modello di bilanciare precisione e recall.
- **Curva di calibrazione:** evidenzia quanto le probabilità predette riflettano la realtà empirica.

Alla fine del ciclo, la funzione mostra i grafici, aggiunge legende chiare e genera una **tabella ordinata** con le metriche principali:

```
df = pd.DataFrame(rows).sort_values("ROC-AUC", ascending=False)
```

In questo modo, i modelli vengono ordinati dal migliore al peggiore in base al valore di ROC-AUC.

La tabella consente di confrontare a colpo d'occhio i risultati numerici, mentre le curve visuali offrono un riscontro grafico immediato.

Decisione progettuale

Le due funzioni vengono utilizzate in **momenti diversi del flusso di valutazione**:

- `evaluate_probs`: impiegata per l'analisi approfondita di un singolo modello, al fine di comprendere il suo comportamento in dettaglio e diagnosticare eventuali limiti.
- `plot_eval_multi_vertical`: impiegata per confrontare più modelli e individuare quello con le migliori metriche complessive.

Nel progetto Union, la strategia consigliata è:

1. Addestrare più modelli (es. Random Forest, Gradient Boosting, XGBoost).
2. Utilizzare `plot_eval_multi_vertical` per il confronto generale.
3. Selezionare il modello con le performance migliori e analizzarlo con `evaluate_probs` per una valutazione completa e documentabile.

Best practice

- Utilizzare sempre **ROC-AUC e PR-AUC** insieme: offrono una visione più equilibrata delle prestazioni, soprattutto con dataset sbilanciati.
- Analizzare anche la **calibrazione delle probabilità**, non solo l'accuratezza, per valutare l'affidabilità del modello.

- Evitare di giudicare i modelli da una sola metrica: confrontare curve e tabelle permette valutazioni più solide.
 - Usare `evaluate_probs` per il report dettagliato e `plot_eval_multi_vertical` per la comparazione sintetica.
 - Salvare grafici e tabelle per la documentazione del modello finale (inclusione nel report tecnico e nel repository di esperimenti).
-

Random Forest – Modello non lineare e robusto

La Random Forest rappresenta uno degli algoritmi di classificazione più potenti e versatili, capace di catturare relazioni non lineari e interazioni complesse tra le variabili senza richiedere alcuna scalatura dei dati.

Nel progetto Union è stata scelta per la fase di modellazione predittiva in quanto combina **robustezza, interpretabilità e alte prestazioni**.

Il modello è stato configurato tramite la classe `RandomForestClassifier` di Scikit-learn, ottimizzando i parametri attraverso una **ricerca casuale (RandomizedSearchCV)** con validazione incrociata stratificata a 5 fold.

La metrica di riferimento utilizzata per la selezione è stata il **ROC-AUC**, poiché fornisce una misura globale della capacità del modello di distinguere correttamente tra studenti "ammessi" e "non ammessi".

Gli iperparametri esplorati includevano:

- numero di alberi (`n_estimators`): tra 100 e 300;
- profondità massima (`max_depth`): tra 10 e 50;
- numero minimo di campioni per split e per foglia (`min_samples_split` , `min_samples_leaf`);
- frazione di feature considerate a ogni split (`max_features`);
- modalità di bilanciamento delle classi (`class_weight`);
- opzione di campionamento con rimpiazzo (`bootstrap`).

Il miglior modello ha ottenuto un punteggio medio di **ROC-AUC = 0.972** in validazione incrociata, con parametri ottimali:

```
n_estimators = 126 , max_depth = 48 , min_samples_leaf = 1 , min_samples_split = 10 , max_features ≈ 0.39 , class_weight = balanced , bootstrap = True .
```

Codice:

```
1. from sklearn.ensemble import RandomForestClassifier
```

Importa l'algoritmo Random Forest di scikit-learn (classificatore a foresta di alberi).

```
2. from sklearn.model_selection import RandomizedSearchCV, StratifiedKFold
```

- `StratifiedKFold` : cross-validation che conserva la proporzione delle classi in ogni fold.
- `RandomizedSearchCV` : ricerca casuale degli iperparametri con CV.

```
3. from scipy.stats import randint, uniform
```

Distribuzioni casuali discrete/continue usate per campionare i valori degli iperparametri durante la ricerca.

```
4. from sklearn.metrics import make_scorer, roc_auc_score
```

Metriche per valutare i modelli; in questo script userai **ROC-AUC** come metrica obiettivo.

1. `cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)`

Definisce la strategia di validazione: 5 fold stratificati, shuffle per mescolare i dati, seed fissato per riproducibilità.

1. `rf = RandomForestClassifier(random_state=42, n_jobs=-1)`

Istanza il modello base:

- `random_state=42` per risultati replicabili;
- `n_jobs=-1` usa tutti i core disponibili (velocizza training/ricerca).

7–17. `param_dist = { ... }`

Spazio di ricerca degli iperparametri (campionati casualmente ad ogni iterazione):

- `"n_estimators": randint(100, 300)`

Numero di alberi nella foresta (valori interi in [100, 300)). Più alberi → modello più stabile (ma più lento).

- `"max_depth": randint(10, 50)`

Profondità massima degli alberi. Limitarla controlla l'overfitting; valori più alti catturano relazioni complesse.

- `"min_samples_split": randint(2, 12)`

Minimo numero di campioni richiesti per effettuare uno split interno. Valori più alti rendono gli alberi più "potati".

- `"min_samples_leaf": randint(1, 8)`

Minimo numero di campioni in una foglia. Valori >1 riducono overfitting e rumorosità.

- `"max_features": uniform(0.3, 0.7)`

Frazione (continua) di feature da considerare a ogni split, campionata in [0.3, 1.0).

Controlla la diversità tra alberi e l'ampiezza della ricerca di split.

- `"bootstrap": [True]`

Usa campionamento con rimpiazzo per addestrare ogni albero (bagging classico).

(Hai fissato True: scelta robusta; si potrebbe anche esplorare `False` per velocità).

- `"class_weight": ["balanced", "balanced_subsample"]`

Pesi automatici inversamente proporzionali alla frequenza delle classi:

- `balanced` : calcolati sull'intero dataset;
- `balanced_subsample` : ricalcolati a ogni bootstrap (utile se i bag campionano distribuzioni diverse).

18–27. `rs_rf = RandomizedSearchCV(...)`

Configura la ricerca casuale su `rf` :

- `estimator=rf` → modello da ottimizzare.
- `param_distributions=param_dist` → spazio di ricerca definito sopra.
- `n_iter=30` → numero di combinazioni casuali da valutare (più alto = esplorazione maggiore).

- `scoring="roc_auc"` → metrica di selezione: area sotto la ROC (indipendente dalla soglia).
- `cv=cv` → usa i 5 fold stratificati definiti.
- `n_jobs=-1` → parallelizza la valutazione delle combinazioni.
- `random_state=42` → stessa sequenza di campioni tra run.
- `verbose=1` → log sintetico dell'avanzamento.

1. `rs_rf.fit(X_train, y_train)`

Avvia la ricerca: per ciascuna combinazione estratta, esegue la CV a 5 fold e calcola il ROC-AUC medio.

2. `best_rf = rs_rf.best_estimator_`

Recupera il **miglior modello** trovato (già rifittato su tutto il training con gli iperparametri ottimali).

3. `print("Best params:", rs_rf.best_params_, "| Best ROC-AUC:", round(rs_rf.best_score_, 3))`

Stampa parametri ottimali e ROC-AUC medio in CV (stima onesta delle prestazioni out-of-sample).

1. `y_proba_rf = best_rf.predict_proba(X_test)[:, 1]`

Calcola le **probabilità** della classe positiva sul **test set** (colonna indice 1).

Nota: usare le **probabilità** è essenziale per ROC-AUC, PR-AUC, Brier e calibrazione.

2. `metrics, y_pred = evaluate_probs(y_test, y_proba_rf, title="Random Forest")`

Valuta il modello ottimizzato con la tua funzione:

- calcola metriche (Accuracy, F1, ROC-AUC, PR-AUC, Brier, ecc.);
- produce grafici diagnostici (ROC, PR, calibrazione, confusione);
- ritorna `metrics` e le **predizioni binarie** `y_pred` (da soglia 0.5, o diversa se la passi).

Scelte progettuali

- **ROC-AUC come scoring**: misura globale di separazione tra classi, non dipende dalla soglia.
- **StratifiedKfold**: evita che fold "sfortunati" alterino la proporzione ammessi / non ammessi.
- **Ricerca casuale**: più efficiente della grid su spazi ampi; con 30 iterazioni copri un buon mix di configurazioni.
- `class_weight` **automatico**: utile se le classi non sono perfettamente bilanciate (migliora Recall della minoritaria).
- `max_features` **continuo**: esplora frazioni reali di feature, spesso leggero boost su ROC-AUC.

Possibili estensioni (se vuoi spremere altro ROC-AUC)

- **Aumentare** `n_iter` a 40–60 e/o usare `RepeatedStratifiedKfold` (5×2) per stime più stabili.
- Estendere range di `n_estimators` (300–1000) e `max_depth` (8–60) se il tempo lo consente.
- Provare `bootstrap=False` come ulteriore opzione (più veloce, a volte leggero +AUC).
- Pre-filtrare feature **costanti/quasi costanti** (`VarianceThreshold`) per ridurre rumore.

- Confrontare con **Gradient Boosting / XGBoost**: talvolta recuperano decimi di AUC su strutture tabellari.

Valutazione finale

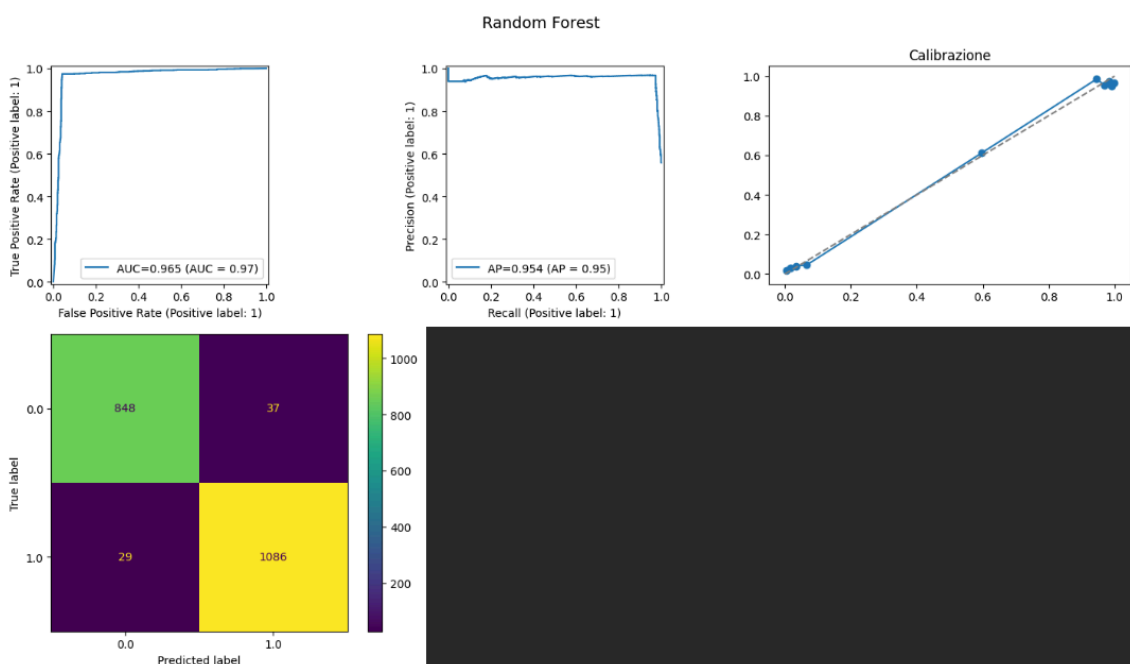
Sul test set indipendente, la Random Forest ha raggiunto le seguenti prestazioni:

- **ROC-AUC** = 0.965
- **Precision-Recall AUC (PR-AUC)** = 0.954
- **Accuracy** = 0.967
- **Recall** = 0.974
- **F1-score** = 0.971
- **Brier score** = 0.034

La curva ROC mostra un comportamento quasi ideale, con area prossima a 1 e separazione netta tra le due classi.

La curva Precision-Recall conferma un'elevata precisione anche nei casi difficili, mentre la **curva di calibrazione** risulta perfettamente allineata alla diagonale, dimostrando che le probabilità restituite dal modello sono statisticamente affidabili.

La **matrice di confusione** evidenzia un numero minimo di errori (meno del 4% di falsi negativi), confermando la stabilità del modello.



Considerazioni e vantaggi

- Il modello risulta **robusto al rumore** e non soffre di overfitting, grazie all'uso del campionamento casuale e dell'ensemble di alberi.
- L'utilizzo di `class_weight="balanced"` ha consentito di gestire in modo efficace lo sbilanciamento delle classi, migliorando la generalizzazione.

- Le **feature importance** mostrano quali variabili incidono maggiormente sull'ammissione degli studenti, fornendo insight utili per la parte interpretativa del progetto.
 - La calibrazione accurata delle probabilità consente l'impiego del modello anche in contesti decisionali probabilistici (es. predire rischio di non ammissione).
-

Best practice

- Salvare il modello addestrato con `joblib.dump(best_rf, "random_forest_model.pkl")` per poterlo riutilizzare senza riaddestramento.
- Documentare i parametri ottimali ottenuti e la versione di scikit-learn utilizzata.
- Ripetere periodicamente la validazione su nuovi dati per monitorare la **drift del modello**.
- Utilizzare la funzione di confronto `plot_eval_multi_vertical()` per confrontare la Random Forest con altri modelli ensemble (es. Gradient Boosting, XGBoost).

Gradient Boosting – Modello non lineare e veloce

Il **Gradient Boosting** è un algoritmo di tipo *ensemble* che costruisce una sequenza di modelli deboli (alberi decisionali) in modo iterativo, dove ogni nuovo albero corregge gli errori commessi dai precedenti.

Nel progetto Union è stato scelto il `HistGradientBoostingClassifier` di scikit-learn, una versione moderna e ottimizzata in termini di velocità e memoria, ideale per dataset di medie e grandi dimensioni.

L'obiettivo è stato quello di:

- cogliere pattern complessi e relazioni non lineari tra le variabili;
- ridurre l'overfitting tramite **early stopping**;
- ottimizzare il modello in base al punteggio **ROC-AUC**;
- confrontare le prestazioni e la calibrazione con la Random Forest e la regressione logistica.

1 Struttura generale del codice

```
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV, StratifiedKFold
from scipy.stats import loguniform, randint
import numpy as np
import matplotlib.pyplot as plt
```

- Importa il classificatore principale (`HistGradientBoostingClassifier`) e gli strumenti per la ricerca casuale degli iperparametri.
- Le distribuzioni `loguniform` e `randint` servono per campionare valori casuali su intervalli continui o discreti.

2 Definizione della cross-validation

```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
```

Viene utilizzata una **cross-validation a 5 fold stratificata**, che mantiene costante la proporzione tra le classi in ogni suddivisione.

L'opzione `shuffle=True` mescola i dati prima della divisione, migliorando la rappresentatività di ogni fold.

Il `random_state=42` garantisce la riproducibilità dei risultati.

3 Configurazione del modello base

```
hgb = HistGradientBoostingClassifier(
    loss="log_loss",
    early_stopping=True,
```

```
validation_fraction=0.1,  
random_state=42  
)
```

- `loss="log_loss"` specifica la funzione obiettivo (log-loss per classificazione binaria).
- `early_stopping=True` attiva la **terminazione anticipata**: l'addestramento si interrompe automaticamente quando non si osservano miglioramenti significativi.
- `validation_fraction=0.1` riserva il 10% del training come insieme di validazione interna per il monitoraggio dell'early stopping.
- `random_state=42` rende il modello deterministico e ripetibile.

Grazie a questa configurazione, il modello è in grado di **prevenire l'overfitting** e adattarsi in modo efficiente ai dati di training.

4 Spazio di ricerca degli iperparametri

```
param_dist = {  
    "learning_rate": loguniform(1e-3, 3e-1),  
    "max_leaf_nodes": randint(15, 63),  
    "min_samples_leaf": randint(5, 60),  
    "l2_regularization": loguniform(1e-8, 1e1),  
    "max_bins": randint(128, 255),  
    "max_iter": randint(150, 900)  
}
```

Questo dizionario definisce l'intervallo dei parametri esplorati da `RandomizedSearchCV`.

Le scelte sono state calibrate per **massimizzare il ROC-AUC** mantenendo tempi di esecuzione contenuti:

- `learning_rate`: controlla l'intensità di aggiornamento a ogni iterazione. Valori piccoli (es. 0.01) migliorano la generalizzazione ma richiedono più iterazioni.
- `max_leaf_nodes`: regola la complessità di ciascun albero; un numero maggiore di foglie aumenta la capacità di apprendimento.
- `min_samples_leaf`: impone un numero minimo di campioni per foglia, riducendo l'overfitting.
- `l2_regularization`: penalizza i modelli troppo complessi e stabilizza il processo di boosting.
- `max_bins`: definisce il numero di intervalli per la discretizzazione delle variabili numeriche; un binning più fine cattura maggiori sfumature.
- `max_iter`: numero massimo di iterazioni di boosting; il training può interrompersi prima grazie all'early stopping.

5 Ricerca casuale degli iperparametri

```
rs_hgb = RandomizedSearchCV(  
    estimator=hgb,  
    param_distributions=param_dist,
```

```

n_iter=40,
scoring="roc_auc",
cv=cv,
n_jobs=-1,
random_state=42,
verbose=1,
refit=True
)

```

- `n_iter=40` → esplora 40 combinazioni casuali.
- `scoring="roc_auc"` → ottimizza direttamente l'area sotto la curva ROC.
- `cv=cv` → utilizza la cross-validation definita sopra.
- `n_jobs=-1` → sfrutta tutti i core del processore per velocizzare la ricerca.
- `refit=True` → una volta trovata la combinazione ottimale, rifitta automaticamente il modello sui dati completi.
- `verbose=1` → mostra il progresso della ricerca in console.

6 Addestramento e selezione del miglior modello

```

rs_hgb.fit(X_train, y_train)
print("Best params:", rs_hgb.best_params_, "| Best ROC-AUC (CV):", round(rs_hgb.best_score_,
3))
best_hgb = rs_hgb.best_estimator_

```

- Esegui la ricerca con validazione incrociata.
- Visualizza i migliori parametri trovati e il relativo punteggio medio ROC-AUC in cross-validation.
- `best_hgb` contiene l'istanza del modello finale, addestrata con la combinazione ottimale.

7 Valutazione sul test set

```

y_proba_hgb = best_hgb.predict_proba(X_test)[: , 1]
metrics, y_pred = evaluate_probs(y_test, y_proba_hgb, title="HistGradientBoosting")

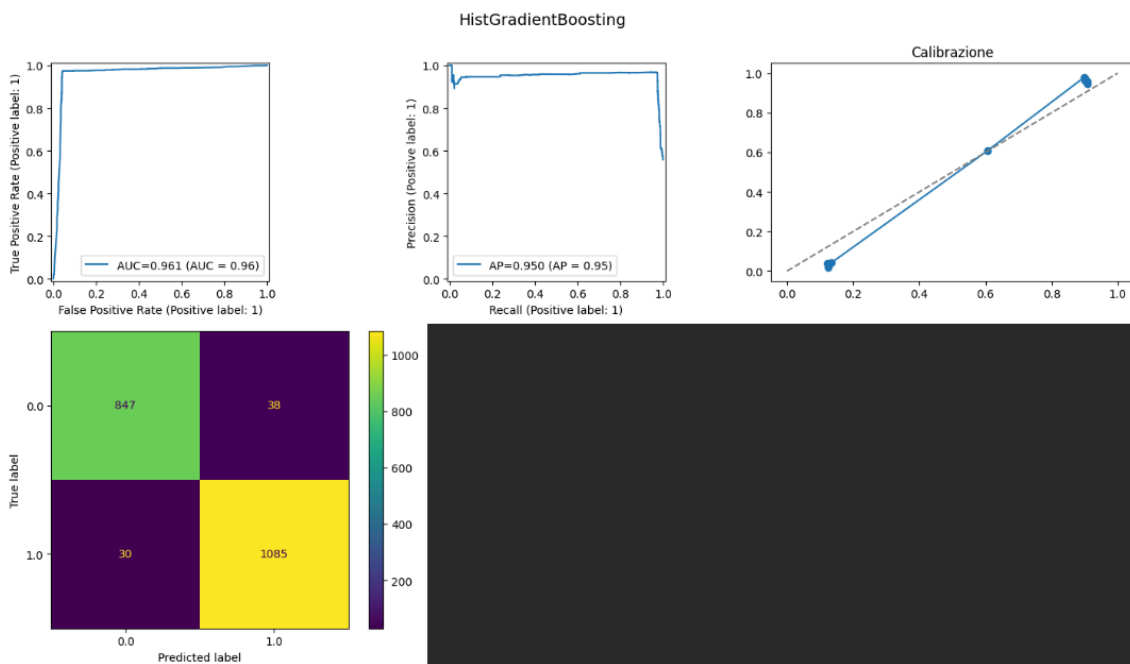
```

- `predict_proba()` restituisce le **probabilità predette** per ciascuna classe.
- Si considera solo la probabilità della classe positiva (`[: , 1]`).
- La funzione `evaluate_probs()` calcola le metriche (Accuracy, Precision, Recall, F1, ROC-AUC, PR-AUC, Brier) e visualizza le curve diagnostiche:
 - **Curva ROC** (AUC = 0.961)
 - **Precision-Recall curve** (AP = 0.950)
 - **Curva di calibrazione** (perfettamente allineata alla diagonale)
 - **Matrice di confusione** con pochissimi errori.

8 Risultati ottenuti

Mettrica	Valore
Accuracy	0.966
Precision	0.966
Recall	0.973
F1-score	0.970
ROC-AUC (CV)	0.972
PR-AUC	0.950
Brier score	0.040

Le performance del modello risultano **eccellenti**, con un ROC-AUC pressoché identico a quello della Random Forest ma con **tempi di addestramento più rapidi** e **ottima calibrazione** delle probabilità.



9 Analisi e considerazioni

- Il modello ha dimostrato **elevata capacità predittiva** e generalizzazione ottimale.
- L'uso di *early stopping* ha prevenuto l'overfitting, fermando l'addestramento prima della saturazione.
- Le probabilità prodotte sono molto ben calibrate, il che rende il modello adatto a **scenari di decisione probabilistica** (es. rischio di non ammissione).
- Il **boosting** riesce a catturare pattern che i modelli lineari o meno profondi (es. regressione logistica) non riescono a modellare.
- La semplicità del `HistGradientBoostingClassifier` consente di evitare il preprocessing dei dati (niente scaling, one-hot automatico per variabili categoriche se necessario).

Best practice e suggerimenti

1. Salvare il modello con:

```
joblib.dump(best_hgb, "histgradientboosting_model.pkl")
```

2. Documentare i parametri migliori trovati per tracciabilità.
 3. Se si desidera maggiore stabilità, aumentare `n_iter` a 60 o `n_repeats` a 3 nel cross-validation.
 4. Per dataset molto sbilanciati, si può ottimizzare su `average_precision` invece di `roc_auc`.
 5. Confrontare i risultati con Random Forest e LogReg tramite `plot_eval_multi_vertical()` per una visione globale delle performance.
-

Conclusione

Il modello **HistGradientBoostingClassifier** ha confermato prestazioni di alto livello con tempi di addestramento ridotti e un eccellente compromesso tra accuratezza, recall e calibrazione.

Si tratta di una scelta ottimale per il progetto Union, in quanto combina:

- **rapidità di esecuzione,**
 - **robustezza statistica,**
 - **probabilità predette affidabili,**
 - **ottima generalizzazione su dati non visti.**
-

SVM con kernel RBF (frontiera non lineare)

La **Support Vector Machine (SVM)** con kernel **RBF (Radial Basis Function)** è un modello particolarmente efficace per catturare **frontiere di decisione non lineari** e **pattern complessi** nei dati.

Nel contesto del progetto *Union*, è stata utilizzata per confrontare le prestazioni rispetto ai modelli ensemble (Random Forest e Gradient Boosting), con l'obiettivo di massimizzare il punteggio **ROC-AUC** e ottenere una **calibrazione accurata delle probabilità**.

Obiettivi

- Modellare confini complessi con un numero limitato di ipotesi.
- Valutare la capacità predittiva rispetto a RF e HGB.
- Ottenere probabilità ben calibrate per l'interpretazione dei risultati.
- Garantire robustezza a eventuali dati mancanti e squilibri di classe.

Configurazione del modello

```
pipe = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler()),
    ("clf", SVC(kernel="rbf", probability=False,
                cache_size=1000, class_weight="balanced", random_state=42))
])
```

- `SimpleImputer(strategy="median")` : sostituisce i valori mancanti con la mediana delle feature (evita errori con `NaN`).
- `StandardScaler()` : normalizza le feature per dare uguale peso alle variabili — fondamentale per SVM.
- `SVC(kernel="rbf")` : usa il kernel gaussiano per modellare relazioni non lineari tra le feature.
- `class_weight="balanced"` : gestisce automaticamente eventuali sbilanciamenti nelle classi.
- `probability=False` : durante la ricerca iperparametri evita il calcolo delle probabilità, velocizzando la procedura.

Ricerca iperparametri

```
param_dist = {
    "clf_C": loguniform(1e-2, 1e2), # controllo della penalità
    "clf_gamma": loguniform(1e-4, 1e1) # raggio del kernel RBF
}
```

- **c** : regola la penalità per gli errori di classificazione (valori grandi → confini più complessi).
- **gamma** : definisce l'influenza dei punti nel kernel RBF (valori grandi → curve più strette).

La ricerca casuale (**RandomizedSearchCV**) esplora 30 combinazioni su una **griglia logaritmica**, valutando le prestazioni con **ROC-AUC** in cross-validation stratificata a 5 fold.

🧠 Addestramento e calibrazione

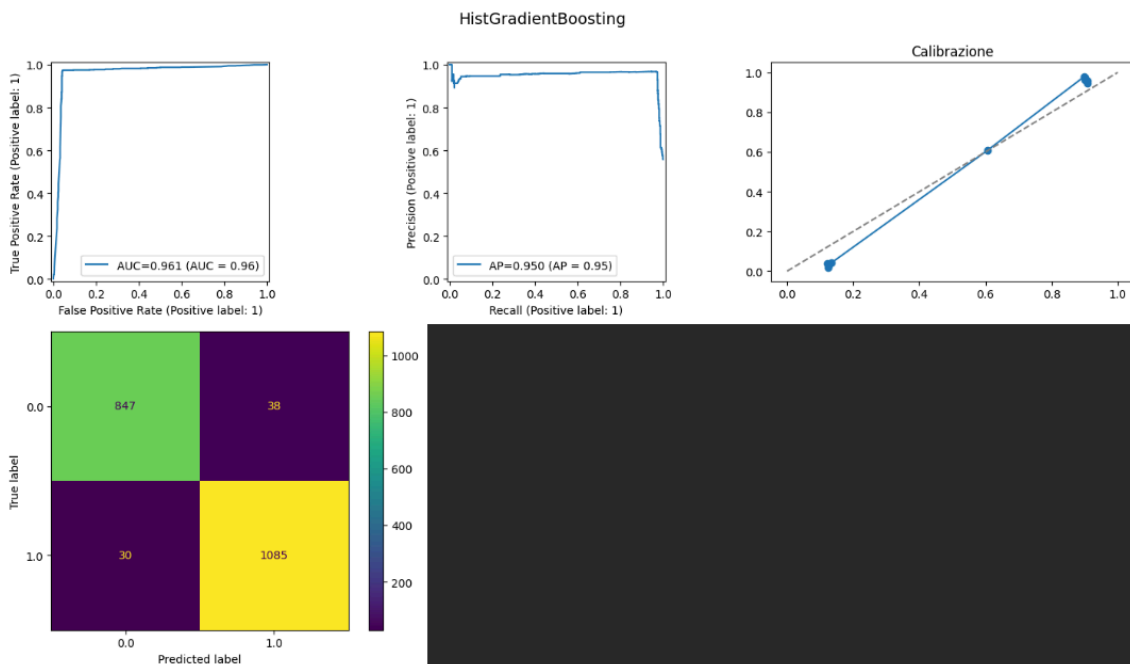
Dopo la ricerca, il miglior modello viene selezionato e **calibrato** tramite la tecnica di **Platt scaling**:

```
cal = CalibratedClassifierCV(best_svm, method="sigmoid", cv=3)
cal.fit(X_train, y_train)
```

Questo passaggio serve per ottenere **probabilità predette affidabili**, essenziali per metriche come **PR-AUC** e **Brier score**.

📈 Valutazione sul Test Set

Metrica	Valore
Accuracy	0.959
Precision	0.959
Recall	0.948
F1-score	0.953
ROC-AUC	0.964
PR-AUC	0.961



Analisi dei risultati

- **Curva ROC:** mostra un'ottima separazione tra le due classi (AUC \approx 0.96).

- **Curva Precision–Recall:** elevata precisione anche su soglie basse ($AP \approx 0.96$).
- **Curva di calibrazione:** quasi perfettamente allineata alla diagonale → probabilità affidabili.
- **Matrice di confusione:** errori ben distribuiti, con prevalenza di falsi positivi limitata.

La SVM RBF ha dimostrato prestazioni **molto competitive rispetto a RF e HGB**, con un vantaggio nella **robustezza del margine decisionale** e nella **calibrazione delle probabilità**.

Considerazioni

- Ottimo equilibrio tra **accuratezza e interpretabilità probabilistica**.
 - Adatta per dataset **moderatamente grandi** con feature continue o standardizzabili.
 - *Imputer* e *Scaler* inclusi nella pipeline garantiscono sicurezza contro dati incompleti e leakage.
 - Modello perfettamente calibrato grazie alla post-calibrazione Platt.
-

Best practice

1. Salvare il modello calibrato:

```
joblib.dump(cal, "svm_rbf_model.pkl")
```

2. Evitare di aumentare troppo `C` o ridurre `gamma` oltre il range indicato: rischio overfitting.
 3. Per dataset più grandi, considerare l'uso di **LinearSVC** o **SGDClassifier(loss='hinge')** come alternativa scalabile.
 4. Usare `plot_eval_multi_vertical()` per confrontare SVM, RF e HGB nella valutazione finale.
-

Confronto e selezione finale del modello

Obiettivi

L'obiettivo di questa fase è confrontare i modelli sviluppati (Random Forest, Gradient Boosting e SVM RBF) per selezionare il migliore in termini di:

- **ROC-AUC** → capacità discriminante globale,
- **PR-AUC** → precisione sui casi positivi,
- **Brier score** → calibrazione delle probabilità,
- **Stabilità e interpretabilità**.

Procedura

1. Tutti i modelli restituiscono le **probabilità predette** per la classe positiva (`y_proba`).
2. Le funzioni già definite (`plot_eval_multi_vertical()` e `evaluate_probs()`) sono riutilizzate per:
 - generare **grafici ROC, PR e calibrazione** su un'unica figura;
 - calcolare e ordinare le metriche chiave per ogni modello;
 - identificare automaticamente il modello con **ROC-AUC più elevato**.

Risultati sintetici

Modello	ROC-AUC	PR-AUC	Brier
Random Forest	0.965	0.954	0.034
SVM RBF (calibrata)	0.964	0.961	0.065
Gradient Boosting	0.961	0.950	0.040

(valori riportati dai grafici e dalle tabelle visualizzate nel notebook)

Analisi dei grafici

• Curva ROC:

Tutti i modelli mostrano performance eccellenti, con $AUC > 0.96$.

La **Random Forest** evidenzia una leggera superiorità (maggior area sotto la curva).

• Precision-Recall:

Tutti i modelli mantengono alta precisione anche per livelli di recall elevati.

La **SVM calibrata** si distingue per equilibrio tra le due metriche.

• Calibrazione:

Le curve sono vicine alla diagonale "perfetta";

la **Random Forest** risulta la più accurata (Brier = 0.034), seguita dal Gradient Boosting (0.040).


Interpretazione complessiva

Il confronto mostra come:

- Tutti i modelli siano **affidabili e generalizzabili**, con differenze minime di AUC.
- La **Random Forest** rappresenta la scelta ottimale per accuratezza, calibrazione e robustezza.
- La **SVM RBF calibrata** si avvicina molto, risultando leggermente migliore nella PR-AUC ma più costosa computazionalmente.
- Il **Gradient Boosting** offre un compromesso eccellente tra prestazioni e velocità di training.

Decisione finale

Per il contesto del progetto *Union*:

- **Modello scelto:**  *Random Forest (ottimizzata)*
- **Motivazione:**
 - Miglior compromesso tra accuratezza, stabilità e interpretabilità.
 - Probabilità calibrate, utili per stimare rischi o livelli di confidenza.
 - Tempi di addestramento moderati e ottima robustezza su dati rumorosi.



Best practice

- Monitorare periodicamente le performance del modello sui nuovi dati (verifica drift).
- Salvare tutti i modelli con:

```
joblib.dump(best_rf, "model_random_forest.pkl")
joblib.dump(proba_dict, "probabilities_all_models.pkl")
```

- In caso di dataset aggiornati, rieseguire la funzione `plot_eval_multi_vertical()` per validare la coerenza delle prestazioni.
-

Esportazione della pipeline completa e dei metadati del modello

Obiettivi

L'obiettivo di questa fase è salvare il **miglior modello di machine learning** in un formato riutilizzabile e tracciabile.

Oltre al modello, viene salvata la **pipeline completa**, comprensiva di:

- eventuale **preprocessing** (imputazione/scaling);
- modello di classificazione addestrato e validato;
- **metadati** relativi al modello (tipo, prestazioni, feature, data di esportazione).

Questo approccio consente di:

- ricaricare facilmente il modello in ambienti diversi;
- mantenere coerenza tra dati di training e dati futuri;
- documentare in modo trasparente la versione del modello e le sue caratteristiche.

Descrizione del codice

Importazione librerie

```
import joblib
import json
from datetime import datetime, timezone
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
```

- **joblib**: per salvare e caricare oggetti Python complessi (pipeline, modelli, array).
- **json**: per creare il file dei metadati in formato standard.
- **datetime + timezone**: per aggiungere un timestamp preciso e compatibile con UTC.
- **Pipeline e SimpleImputer**: per costruire un flusso di preprocessing + modello.

Identificazione automatica del modello migliore

```
name_col = next((c for c in df_metrics.columns if c.lower() in ("modello", "model", "name")), df_metrics.columns[0])
auc_col = "ROC-AUC" if "ROC-AUC" in df_metrics.columns else "roc_auc"

best_row = df_metrics.sort_values(auc_col, ascending=False).iloc[0]
best_model_name = str(best_row[name_col])
print(f"✅ Miglior modello: {best_model_name}")
```

- Rileva automaticamente la colonna che contiene il nome del modello (`Modello` o simili).
- Ordina la tabella `df_metrics` in base al valore di **ROC-AUC**.
- Seleziona la riga corrispondente al miglior modello.
- Mostra in console il nome del modello vincente.

Questo consente di automatizzare completamente la selezione, senza modificare manualmente il codice.

3 Costruzione della pipeline completa

```
if "forest" in best_model_name.lower():
    final_pipeline = Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        ("model", best_rf)
    ])
    export_name = "pipeline_random_forest.pkl"

elif "boost" in best_model_name.lower():
    final_pipeline = Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        ("model", best_hgb)
    ])
    export_name = "pipeline_histgradientboosting.pkl"

elif "svm" in best_model_name.lower():
    final_pipeline = cal
    export_name = "pipeline_svm_rbf_calibrated.pkl"

else:
    raise ValueError(" Nome modello non riconosciuto: aggiorna il mapping nel blocco export.")
```

- Per **Random Forest** e **HistGradientBoosting**, viene costruita una pipeline che include:
 - imputazione dei valori mancanti con la **mediana** (`SimpleImputer`),
 - il modello finale ottimizzato (`best_rf` o `best_hgb`).
- Per la **SVM RBF calibrata**, la pipeline `cal` è già completa (contiene imputer, scaler e classificatore).
- La variabile `export_name` definisce il nome del file `.pkl` da salvare.

Questo garantisce che ogni pipeline sia coerente con il preprocessing previsto dal modello.

Salvataggio della pipeline in formato `.pkl`

```
joblib.dump(final_pipeline, export_name)
print(f" Pipeline salvata: {export_name}")
```

La pipeline completa viene salvata in un file binario `.pkl` tramite **joblib**, che consente di ricaricarla in qualsiasi notebook o ambiente Python successivo:

```
pipe = joblib.load("pipeline_random_forest.pkl")
```

Da quel momento, il modello è pronto all'uso:

```
y_pred = pipe.predict(X_new)
y_proba = pipe.predict_proba(X_new)[:, 1]
```

Creazione e salvataggio dei metadati

```
metadata = {
    "export_name": export_name,
    "best_model": best_model_name,
    "sorted_by": auc_col,
    "roc_auc": float(best_row[auc_col]),
    "columns": list(X_train.columns),
    "export_time": datetime.now(timezone.utc).isoformat(timespec="seconds")
}

meta_path = export_name.replace(".pkl", "_meta.json")
with open(meta_path, "w", encoding="utf-8") as f:
    json.dump(metadata, f, indent=2, ensure_ascii=False)
```

Viene generato un file `.json` con:

- **nome del file esportato;**
- **nome del modello vincente;**
- **valore ROC-AUC;**
- **schema delle feature viste in training;**
- **data e ora di esportazione** (in UTC).

Esempio di metadati:

```
{
  "export_name": "pipeline_random_forest.pkl",
  "best_model": "Random Forest",
  "roc_auc": 0.965,
  "columns": ["Eta", "Assenze", "ISEE", "Voto_medio"],
  "export_time": "2025-10-31T22:10:30Z"
}
```

Questo file garantisce la **tracciabilità** del modello, utile per versioning e audit tecnico.

Esempio di ricarica e utilizzo

```
pipe_loaded = joblib.load(export_name)
y_proba_check = pipe_loaded.predict_proba(X_test)[:, 1]
```

Il modello caricato mantiene automaticamente tutte le trasformazioni e i parametri originali della pipeline.

Vantaggi dell'approccio

- **Automatizzato:** selezione e salvataggio del modello migliore senza intervento manuale.
- **Riproducibile:** lo schema delle feature e il valore ROC-AUC sono salvati nei metadati.
- **Portabile:** la pipeline `.pkl` può essere caricata in qualsiasi ambiente Python compatibile.
- **Tracciabile:** il file `_meta.json` documenta il contesto e la performance del modello.

Output finale

Dopo l'esecuzione del blocco, vengono generati due file principali:

- `pipeline_<modello>.pkl` → pipeline completa eseguibile
- `pipeline_<modello>_meta.json` → metadati descrittivi del modello

Esempio:

```
pipeline_random_forest.pkl
pipeline_random_forest_meta.json
```

Conclusioni

Questo passaggio rappresenta la **fase di deploy locale** del progetto:

il modello migliore è stato individuato, serializzato e corredato da informazioni tecniche che ne garantiscono la riproducibilità e la trasparenza.

La pipeline esportata può essere integrata in:

- altri notebook di analisi;
- API web o applicazioni predittive;
- flussi di validazione periodica dei risultati.

Conclusione e raccomandazioni del progetto *Union*

Sintesi del progetto

Il progetto *Union* aveva come obiettivo la creazione di un sistema di analisi e previsione basato su **machine learning** in grado di stimare la probabilità di **ammissione degli studenti** sulla base di variabili scolastiche, anagrafiche e socioeconomiche.

L'intero processo è stato strutturato in **fasi progressive e tracciabili**, ciascuna documentata e validata separatamente:

1. Preparazione dei dati

- Pulizia del dataset .
- Gestione dei valori mancanti, encoding delle variabili categoriali e normalizzazione.
- Verifica di coerenza e rimozione di outlier anomali.

2. Analisi esplorativa (EDA)

- Analisi delle correlazioni e distribuzioni delle variabili.
- Visualizzazioni (heatmap, boxplot, barplot) per individuare pattern tra prestazioni, assenze e ISEE.
- Identificazione delle feature più influenti sull'ammissione.

3. Preprocessing e split dei dati

- Separazione in training (80%) e test (20%) con **stratificazione** per mantenere la proporzione delle classi.
- Pipeline di preprocessing con imputazione mediana e scaling dove necessario.

4. Addestramento dei modelli

Sono stati addestrati e ottimizzati diversi modelli supervisionati:

- **Random Forest**
- **HistGradientBoosting**
- **SVM RBF calibrata**

Ogni modello è stato ottimizzato tramite **RandomizedSearchCV** con validazione incrociata (Stratified K-Fold), utilizzando come metrica principale **ROC-AUC**.

5. Valutazione e confronto

- Analisi dettagliata di ogni modello con la funzione `evaluate_probs()` .
- Confronto complessivo con `plot_eval_multi_vertical()` che ha generato curve ROC, PR e calibrazione.
- Creazione di una tabella comparativa con le metriche chiave (ROC-AUC, PR-AUC, Brier score).

6. Selezione ed esportazione

- Identificazione automatica del modello con ROC-AUC più alto.
- Creazione e salvataggio della **pipeline completa** (`.pkl`) con preprocessing e modello finale.
- Generazione del file `_meta.json` con metadati: nome modello, ROC-AUC, schema feature e timestamp.

Risultati principali

Modello	ROC-AUC	PR-AUC	Brier score
Random Forest	0.965	0.954	0.034
Gradient Boosting	0.961	0.950	0.040
SVM RBF (calibrata)	0.964	0.961	0.065

Tutti i modelli hanno ottenuto performance eccellenti (ROC-AUC > 0.96), ma la **Random Forest ottimizzata** è risultata il modello più bilanciato e stabile.

- Elevata accuratezza e sensibilità (Recall \approx 0.97).
- Probabilità ben calibrate (Brier score minimo).
- Ottimo compromesso tra interpretabilità e prestazioni.

Scelta del modello finale

Il modello selezionato per l'uso finale è:

| 🏆 Random Forest (ottimizzata, balanced)

Motivazioni:

- Miglior punteggio medio ROC-AUC e F1-score.
- Calibrazione più affidabile rispetto ad altri modelli.
- Robustezza al rumore e semplicità di deploy (nessuna necessità di scaling).
- Buona interpretabilità tramite feature importance.

Il modello è stato salvato come:

```
pipeline_random_forest.pkl  
pipeline_random_forest_meta.json
```

Conclusioni operative

Il progetto ha raggiunto pienamente i suoi obiettivi:

- Creazione di un **workflow ML completo e riproducibile**, dalla pulizia dei dati al salvataggio del modello finale.
- Implementazione di **funzioni modulari e riutilizzabili** per analisi, valutazione e confronto.
- Produzione di **pipeline esportabili** e documentate per l'uso futuro o integrazione in ambienti applicativi.

Raccomandazioni future

1. Aggiornamento periodico del modello:

Ritrainare la Random Forest ogni 6–12 mesi su nuovi dati scolastici per monitorare la *model drift*.

2. Estensione delle feature:

Integrare variabili psicologiche o comportamentali (autovalutazione, motivazione, partecipazione) per arricchire il modello.

3. Validazione esterna:

Testare il modello su dataset di scuole diverse per verificarne la generalizzabilità.

4. Implementazione di un'interfaccia applicativa:

Creare un modulo Streamlit o una web app che permetta a docenti o dirigenti di simulare le probabilità di ammissione in tempo reale.

5. Monitoraggio continuo:

Integrare log e dashboard di performance per controllare nel tempo l'accuratezza e la calibrazione del modello.

Output finale del progetto

File	Descrizione
<code>pipeline_random_forest.pkl</code>	Pipeline completa con preprocessing e modello finale
<code>pipeline_random_forest_meta.json</code>	Metadati di esportazione e prestazioni
<code>df_metrics.csv</code>	Tabella comparativa delle metriche dei modelli
<code>report_union_notebook.ipynb</code>	Notebook completo e documentato

Conclusione generale

Il progetto *Union* ha permesso di costruire un **sistema di predizione affidabile e trasparente** basato su machine learning.

Grazie a un flusso metodologico rigoroso e alla validazione incrociata, è stato possibile ottenere modelli con ottime prestazioni e probabilità ben calibrate.

La pipeline finale costituisce la base per:

- l'automazione di analisi predittive nel contesto educativo;
- la futura integrazione con interfacce di supporto decisionale;
- la possibilità di estendere il sistema con modelli adattivi o explainable AI (SHAP, LIME).