

Data and Web Mining

Kaggle ft Child Institute: problematic internet use

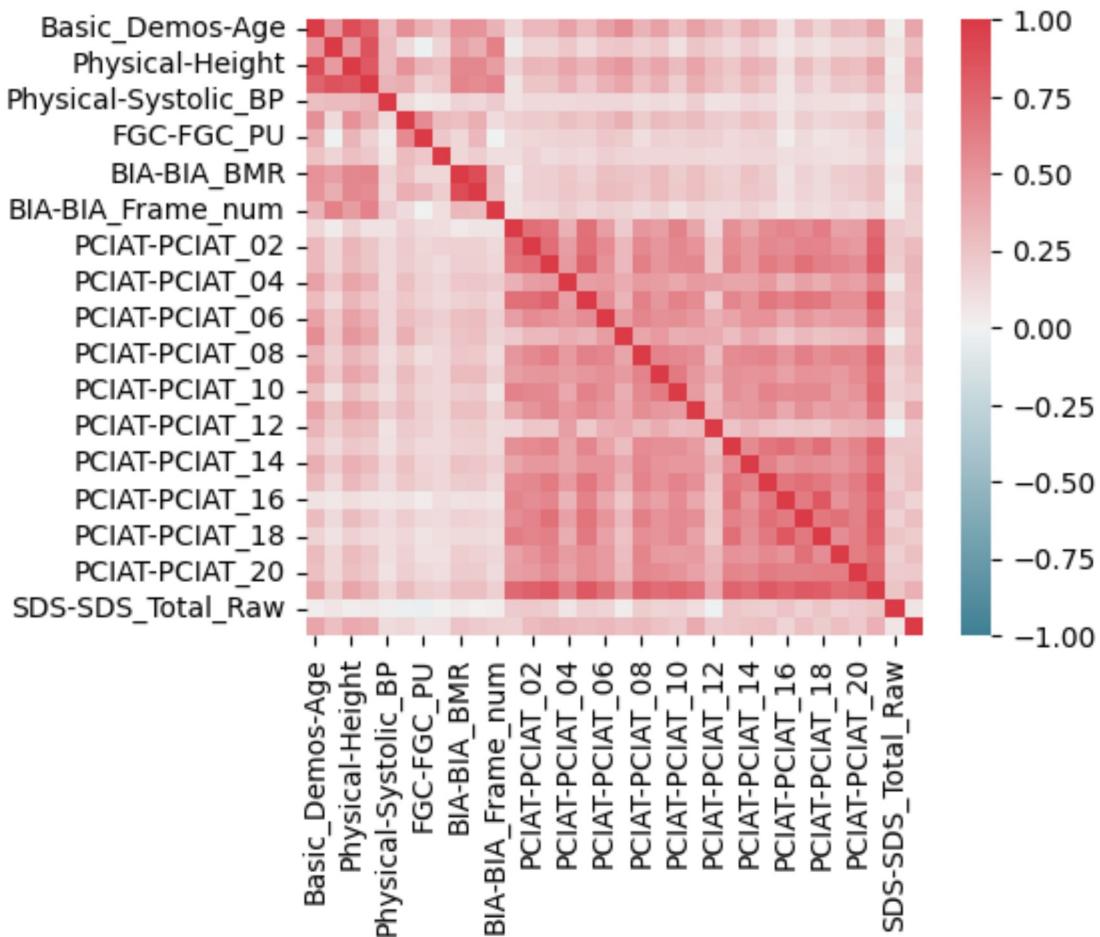
alleneremo e metteremo a confronto due modelli. uno ad albero e uno basato su rete neurale.

```
In [ ]: import pandas as pd
from sklearn.model_selection import train_test_split
from data_cleaning_module import prepare_data, augment_data, correlation_matrix1
from model0_baseline_module import run_baseline
from model1_XGBoost_module import tune_and_train_xgboost, test_xgboost
from model2_ann_module import run_nn, prepare_X, impute_missing_values, rf_cascade
SEED = 42 # global random seed for reproducibility
```

Preparazione dati

Le prime cosa da fare a partire dai dati grezzi sono: split tra train/validation/test, trasformazione delle variabili categoriche in variabili numeriche e eliminazione delle features con troppi NaN, non predittive o ridondanti. Nel train abbiamo eliminato tutte le colonne con una percentuale di missing values > 50%. Abbiamo deciso di adottare un approccio ordinale per mappare le stagioni in numeri: winter:0, spring:1, summer:2, fall:3, NaN:NaN. Dopodiché abbiamo fatto la matrice di correlazione tra tutte le colonne rimanenti, eliminando le features ridondanti (correlazione > 0.9).

```
In [2]: X_train, y_train, X_val, y_val, X_test, y_test = prepare_data("../data/train.csv")
X_train_filtered = correlation_matrix1(X_train, y_train)
```



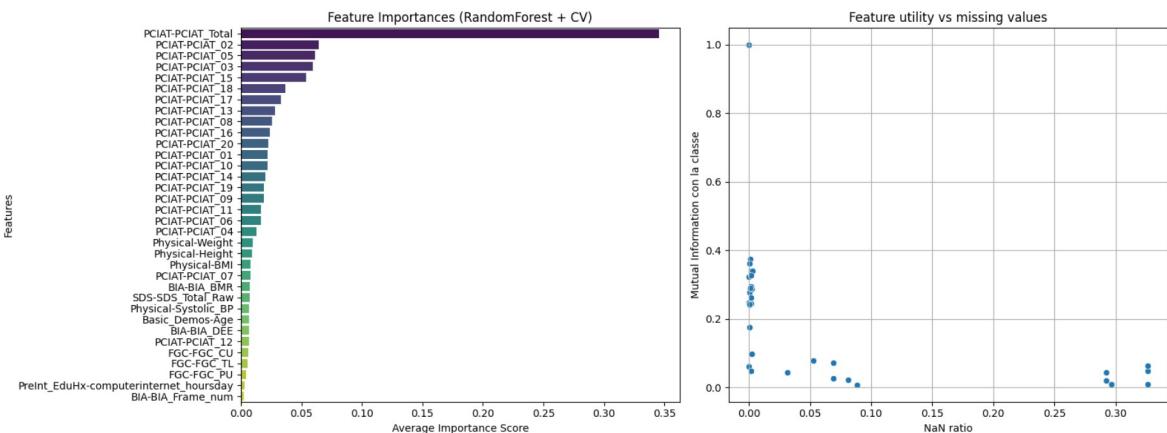
Per scremare ulteriormente le features (per avere maggior interpretabilità e minore complessità computazionale), abbiamo allenato una random forest in cross validation, per valutare la features importance. Abbiamo anche dato un'occhiata all'analisi statistica "mutual information", che non dipende dal modello, ma misura "quanto sapere la feature aiuta a prevedere la classe". Messa in grafico col NaN ratio, dà anch'essa una buona misura di cosa sia significativo o meno, per confrontarlo coi risultati ottenuti dalla feature importance appena calcolata.

```
In [3]: feat_imp_cv, feat_imp_cv_sorted = feature_importance_and_mutual_info(X_train_fil
```

```
c:\Users\David\Uni_Lavoro\data-web-mining-project\src\data_cleaning_module.py:153:
FutureWarning:
```

```
Passing `palette` without assigning `hue` is deprecated and will be removed in v0
.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effec
t.
```

```
sns.barplot(
```



Dall'analisi della feature importance e della mutual information emergono alcune evidenze. La variabile più predittiva per l'etichetta Sii è PCIAT_Total, seguita da alcune specifiche domande (02, 05, 03, 15, 17). Le due analisi concordano: le feature più rilevanti coincidono, mentre quelle con un alto numero di valori mancanti o con bassa MI risultano poco utili e possono essere eliminate, ad esempio tramite una random forest. Tuttavia, le variabili PCIAT introducono un problema di data leakage, perché sono fortemente correlate o addirittura derivate direttamente dal target, quindi non sarebbero disponibili in scenari reali. Per questo motivo è necessario escluderle dal training del modello. Allo stesso tempo, proprio per la loro forte correlazione, esse possono essere sfruttate in un secondo momento per stimare i valori mancanti di Sii e arricchire così il dataset.

Proxy Model

Procediamo quindi con la creazione del modello di imputazione. Dovremo quindi rifare i passaggi già eseguiti, in quanto il primo passo fatto è stato eliminare le righe con sii ignoto.

1. a Partire dal dataset originale, ripetiamo i passi base di preparazione dati:
 - A. separiamo train/test
 - B. A partire da X_train creiamo due df distinti, di cui uno contiene le righe del training set dove sii è noto, uno dove è NaN
 - C. uso le PCIAT come features e sii come target per allenare una semplice ma efficace random forest, che poi uso sul df con sii NaN.
2. A questo punto, imputiamo i valori di sii mancanti.
3. Eliminiamo quindi direttamente le colonne PCIAT, ripetendo la feature selection come fatta prima, senza che queste colonne siano coinvolte.

```
In [4]: X_train_final, y_train_final, X_test, y_test = augment_data()
```

```
Righe di X_training con 'sii' noto: 2193
Righe di X_training da imputare: 975
Dataset salvati in formato CSV nella cartella 'data/'.
```

- X_train_final : contiene il set di training, totale: senza le colonne PCIAT e aumentato con le righe che inizialmente avevano sii nullo
- y_train_final : target del training, con tutte le righe, anche quelle imputate

- `X_test`, `y_test`: set di test, intoccati, con dati ancora grezzi

A questo punto, ci sono quasi 1000 righe in più su cui allenare i modelli. Procediamo a rifare i passi eseguiti in precedenza riguardanti pulizia dei dati e feature importance. (cambiare le stagioni, matrice di correlazione, feature importance, nan ratio)

```
In [5]: correlation_matrix2(X_train_final, y_train_final)
```

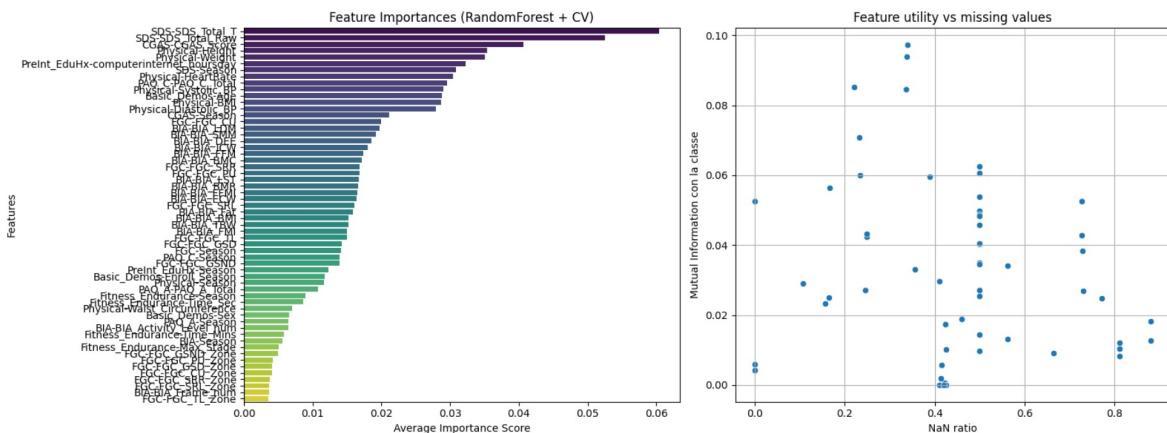


```
In [6]: feat_imp, feat_imp_sorted = feature_importance_and_mutual_info(X_train_final, y_
```

```
c:\Users\David\Uni_Lavoro\data-web-mining-project\src\data_cleaning_module.py:153:
FutureWarning:
```

```
Passing `palette` without assigning `hue` is deprecated and will be removed in v0
.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effect.
```

```
sns.barplot(
```

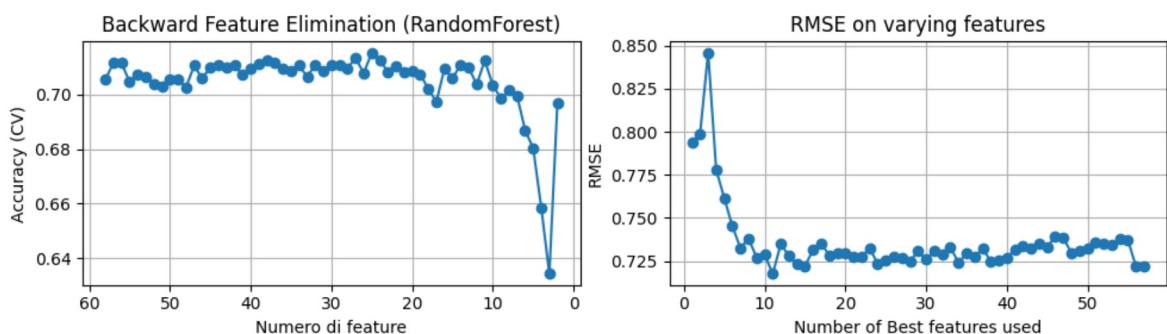


l'importance score è in generale molto più basso e decisamente meno sbilanciato di prima. Le features con NaN ratio alto non sono quelle con importance in assoluto minore, ma sono comunque nella bassa classifica, come atteso. Ugualmente alta mutual information corrisponde ad alta importance.

Feature subset selection by elimination

Abbiamo applicato una backward feature selection utilizzando come modello una Random Forest. Siamo partiti dal set completo di variabili e abbiamo rimosso iterativamente quelle con minor feature importance, valutando a ogni passo le performance sul validation set. Questo approccio ha il vantaggio di gestire bene le feature correlate: infatti, la Random Forest tende a dividere l'importanza tra variabili simili, ma con la rimozione progressiva una di esse può rivelarsi molto più rilevante, compensando così la penalizzazione dovuta alla ridondanza.

```
In [7]: X_train_final, features_list = backward_elimination(X_train_final, y_train_final)
```



Top 11 features (RandomForest CV):

1. SDS-SDS_Total_T
2. SDS-SDS_Total_Raw
3. CGAS-CGAS_Score
4. Physical-Height
5. Physical-Weight
6. PreInt_EduHx-computerinternet_hoursday
7. SDS-Season
8. Physical-HeartRate
9. PAQ_C-PAQ_C_Total
10. Physical-Systolic_BP
11. Basic_Demos-Age

'data/X_train_final.csv' sovrascritto coi nuovi dati.

Da questi due grafici si evince che è possibile eliminare ripetutamente le features senza

perdite in termini di accuracy: è chiaro come avere una decina di features sia tanto significativo quanto averle tutte, la perdita di accuracy è totalmente giustificata dalla semplificazione del modello. nel range 20-10 c'è una diminuzione leggermente più marcata, con maggiore varibilità, ma forse ugualmente accettabile. Da 11 in giù le performance calano sensibilmente.

Abbiamo scelto di tenerne quindi 11. proseguiamo quindi con:

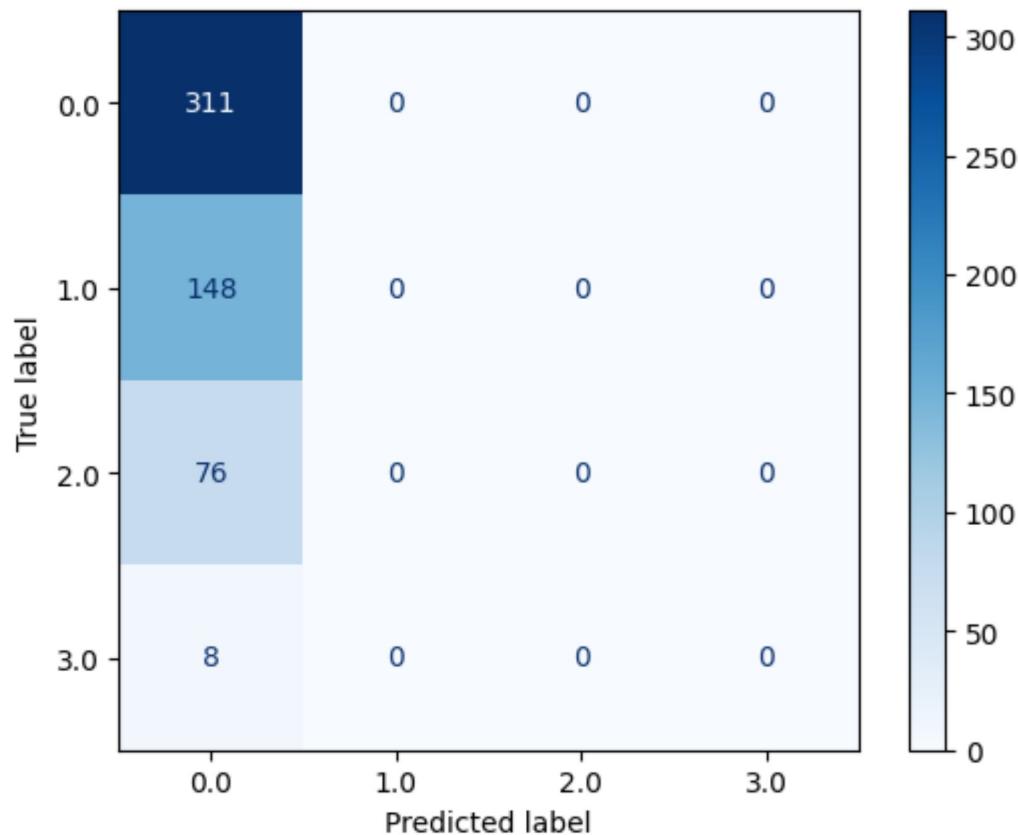
1. Mostrare le feature ordinate per importanza
2. Selezionare le prime 11 e droppare dal dataset le features non utilizzate

Modello 0: baseline

Uno dei modelli più semplici che si possono utilizzare per eseguire delle predizioni è la moda.

```
In [9]: print("== Model 0: Baseline ==")
run_baseline(X_train_final, y_train_final, X_test, y_test)
```

```
== Model 0: Baseline ==
Baseline accuracy: 0.572744014732965
```



Si nota che le classi sono fortemente sbilanciate e, essendo un test medico, è essenziale minimizzare il numero di falsi negativi della categoria 3 (corrispondente a "severe"). Infatti, un falso positivo è meno problematico, in quanto dopo la predizione con il modello, si può eseguire un test reale sulla persona. In caso di un falso negativo, invece, si scarta a priori l'ipotesi ed è più improbabile che venga effettuato un ulteriore test.

Il test **non verrà mai usato** se non alla fine, per valutare le performance dei due modelli

e confrontarle con la baseline

```
In [10]: y_train_final = y_train_final.astype(int)
y_test = y_test.astype(int)
X_tr, X_val, y_tr, y_val = train_test_split(X_train_final, y_train_final, test_size=0.2, random_state=42)

X_tr_scaled, X_val_scaled, class_weights, scaler = prepare_X(X_tr, X_val, y_tr, y_val)
X_tr_scaled, X_val_scaled, imputer = impute_missing_values(X_tr, X_val, SEED, scale=True)
```

Modello 1: XGBoost

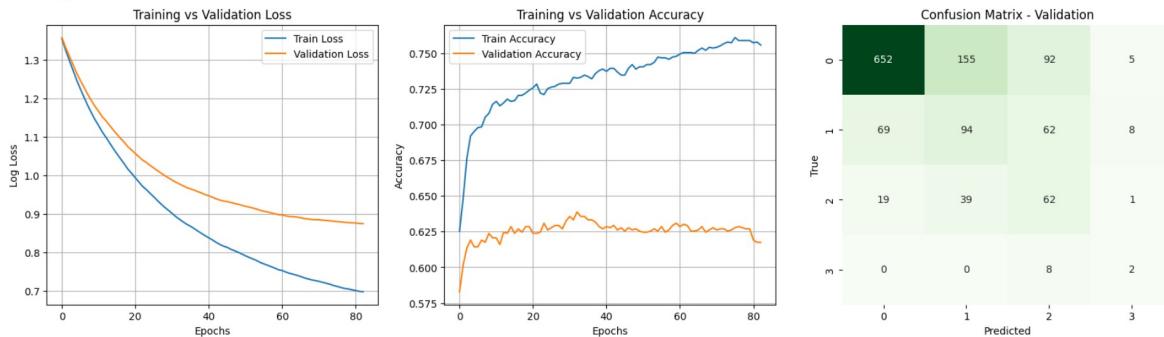
Xgboost è un algoritmo di gradient boosting ottimizzato che costruisce modelli ensemble di alberi decisionali in modo molto efficiente. Nel nostro lavoro abbiamo seguito diversi passaggi per costruire e addestrare un modello di classificazione multiclasse basato su XGBoost. Per prima cosa abbiamo impostato il modello in modo da gestire più categorie, adattandolo quindi al contesto del nostro problema. Successivamente ci siamo occupati dello sbilanciamento delle classi: invece di lasciare che quelle più frequenti influenzassero troppo l'addestramento, abbiamo calcolato e applicato dei pesi proporzionati alla distribuzione.

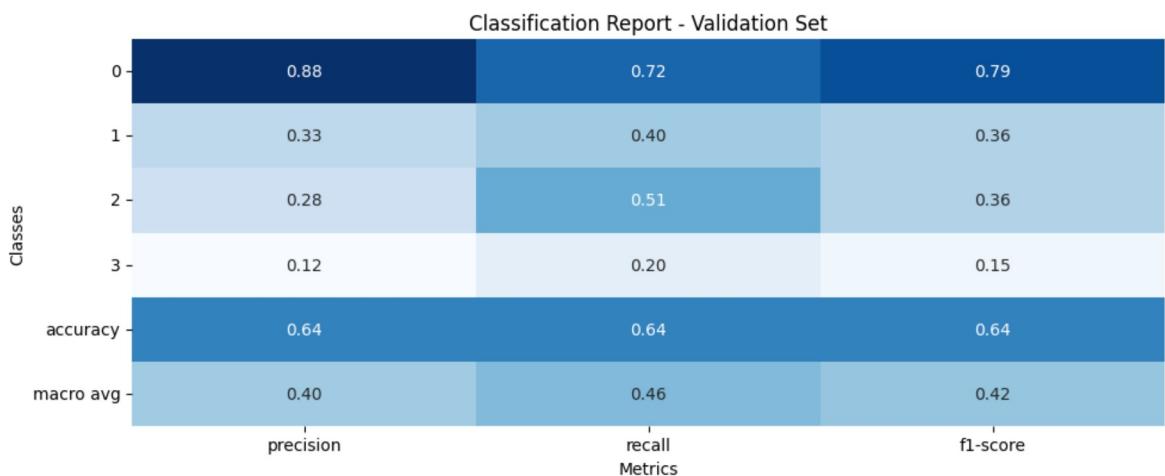
A questo punto abbiamo introdotto Optuna, una libreria che permette di automatizzare l'ottimizzazione degli iperparametri. In questo modo il modello ha potuto essere testato con diverse configurazioni e noi abbiamo potuto individuare quella che massimizzava la metrica F1 ponderata sul validation set.

Infine, con gli iperparametri migliori selezionati, abbiamo addestrato il modello finale, ottenendo così una versione più robusta e meglio calibrata sulle nostre esigenze.

```
In [11]: print("== Model 1 ==")
model1, study = tune_and_train_xgboost(X_tr_scaled, y_tr, X_val_scaled, y_val, n_estimators=100, max_depth=6, learning_rate=0.04482378180660554, min_child_weight=9, subsample=0.6305190980496068, colsample_bytree=0.8843113093738098, gamma=1.2342431082747214, reg_alpha=0.7241029073314151, reg_lambda=0.12583887940614819)

== Model 1 ==
Best hyperparameters: {'max_depth': 6, 'learning_rate': 0.04482378180660554, 'min_child_weight': 9, 'subsample': 0.6305190980496068, 'colsample_bytree': 0.8843113093738098, 'gamma': 1.2342431082747214, 'reg_alpha': 0.7241029073314151, 'reg_lambda': 0.12583887940614819}
== Validation Set ==
Accuracy: 0.638801261829653
Weighted F1: 0.6673079294741331
```





Modello 2: Rete Neurale, ensemble con RF

Siamo partiti dall'addestramento di una rete neurale, cercando di migliorarla variando diversi iperparametri, tra cui il numero di neuroni nei layer nascosti, le funzioni di attivazione, il dropout, la regolarizzazione L2 e il learning rate. Per affrontare il forte sbilanciamento del dataset, abbiamo anche provato a aumentare i pesi delle classi più rare. Successivamente, abbiamo testato altre strategie di miglioramento, come aggiungere neuroni o layer con regolarizzazione e utilizzare metriche alternative come MAE e MSE in virtù del rapporto ordinale tra le classi, senza però ottenere miglioramenti significativi.

Per la gestione dei valori mancanti, abbiamo utilizzato una prima Random Forest come imputer "intelligente", stimando i NaN tenendo conto delle relazioni tra le variabili, anziché ricorrere a media, mediana o moda.

Infine, abbiamo adottato un approccio di classificazione a cascata: una seconda Random Forest viene allenata per predire la probabilità di appartenenza a due macrocategorie ("probabilmente sano - 0,1" vs "probabilmente problematico 2,3"), e la rete neurale utilizza questa nuova feature insieme alle altre per la classificazione finale. Questo approccio ha portato a un miglioramento significativo sulle prestazioni del modello sui dati di validazione.

```
In [12]: # 4. Modello 2
print("== Model 2: Neural Network ==")

# primo step random forest
X_tr_aug, X_val_aug, n_features_aug, rf = rf_cascade(X_tr_scaled, y_tr, X_val_sc
```

```
==== Model 2: Neural Network ====
==== Confusion Matrix - TRAIN ====
[[1670  33]
 [ 1 196]]

==== Confusion Matrix - VAL ====
[[1069  68]
 [ 86  45]]

==== Classification Report - TRAIN ====
      precision    recall   f1-score   support
0         1.00     0.98     0.99    1703
1         0.86     0.99     0.92    197

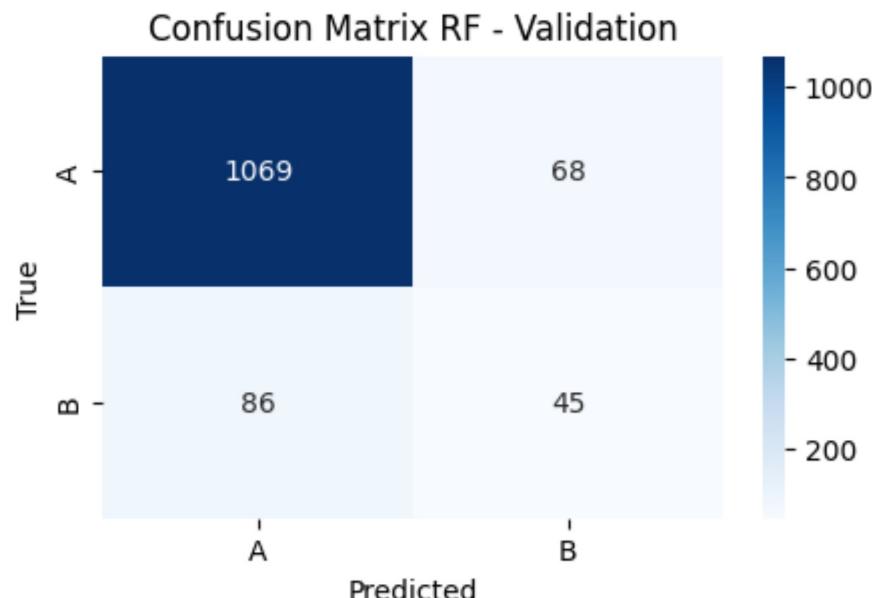
accuracy                           0.98    1900
macro avg       0.93     0.99     0.96    1900
weighted avg    0.98     0.98     0.98    1900
```

```
==== Classification Report - VAL ====
      precision    recall   f1-score   support
0         0.93     0.94     0.93    1137
1         0.40     0.34     0.37    131

accuracy                           0.88    1268
macro avg       0.66     0.64     0.65    1268
weighted avg    0.87     0.88     0.87    1268
```

TRAIN Accuracy RF: 0.9821

VAL Accuracy RF: 0.8785



Shape train: (1900, 12)

Shape val: (1268, 12)

```
In [13]: # secondo step tuning automatico
final_model = tune_hyperparameters(X_tr_aug, y_tr, X_val_aug, y_val, n_features_
```

```
Reloading Tuner from my_keras_tuner\dwm_tuning\tuner0.json
```

```
--- RICERCA COMPLETATA ---
```

```
I migliori iperparametri trovati sono:  
- Unità primo layer: 32  
- Unità secondo layer: 64  
- Funzione di attivazione: tanh  
- Tasso di Dropout: 0.50  
- Regolarizzazione L2: 0.0000  
- Learning Rate: 0.01
```

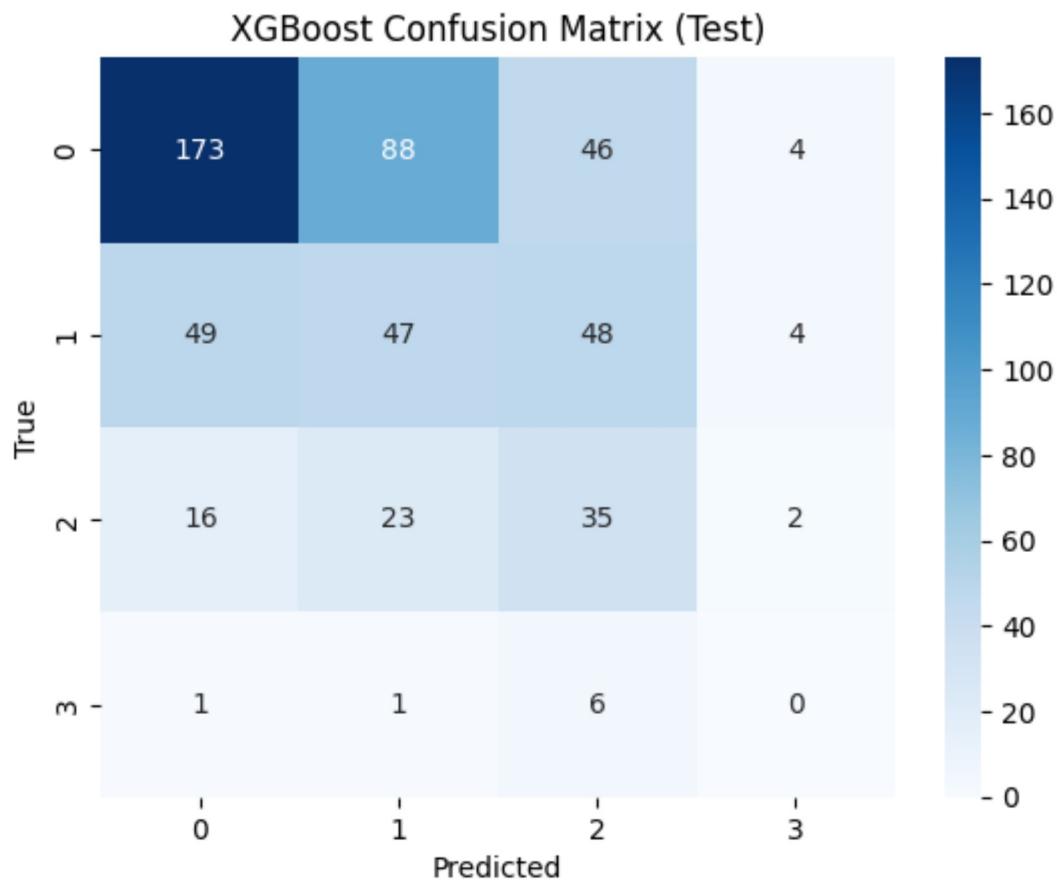
👉 Addestramento del modello finale con i parametri ottimali...

Test dei modelli

```
In [14]: X_test11 = X_test.copy()  
X_test11 = X_test11[features_list]  
  
X_test11 = season_map(X_test11)  
  
X_test_imputed = imputer.transform(X_test11)  
X_test_scaled = scaler.transform(X_test_imputed)
```

```
In [26]: test_xgboost(model1, X_test_scaled, y_test)
```

```
==== XGBoost - Classification Report (Test) ====  
precision    recall    f1-score   support  
  
          0       0.7238     0.5563     0.6291      311  
          1       0.2956     0.3176     0.3062      148  
          2       0.2593     0.4605     0.3318       76  
          3       0.0000     0.0000     0.0000        8  
  
accuracy                  0.4696      543  
macro avg       0.3197     0.3336     0.3168      543  
weighted avg     0.5314     0.4696     0.4902      543
```



```
In [ ]: test_ann_pipeline(X_test_scaled, y_test, final_model, rf)
```

```
17/17 ━━━━━━━━ 0s 7ms/step
==== ANN Ensemble - Classification Report (Test) ====
      precision    recall   f1-score   support
0       0.7003   0.7363   0.7179     311
1       0.3643   0.3446   0.3542     148
2       0.3261   0.1974   0.2459      76
3       0.1000   0.3750   0.1579       8

accuracy                           0.5488     543
macro avg       0.3727   0.4133   0.3690     543
weighted avg    0.5475   0.5488   0.5444     543
```

