

# Data and Web Mining

## Kaggle ft Child Institute: problematic internet use

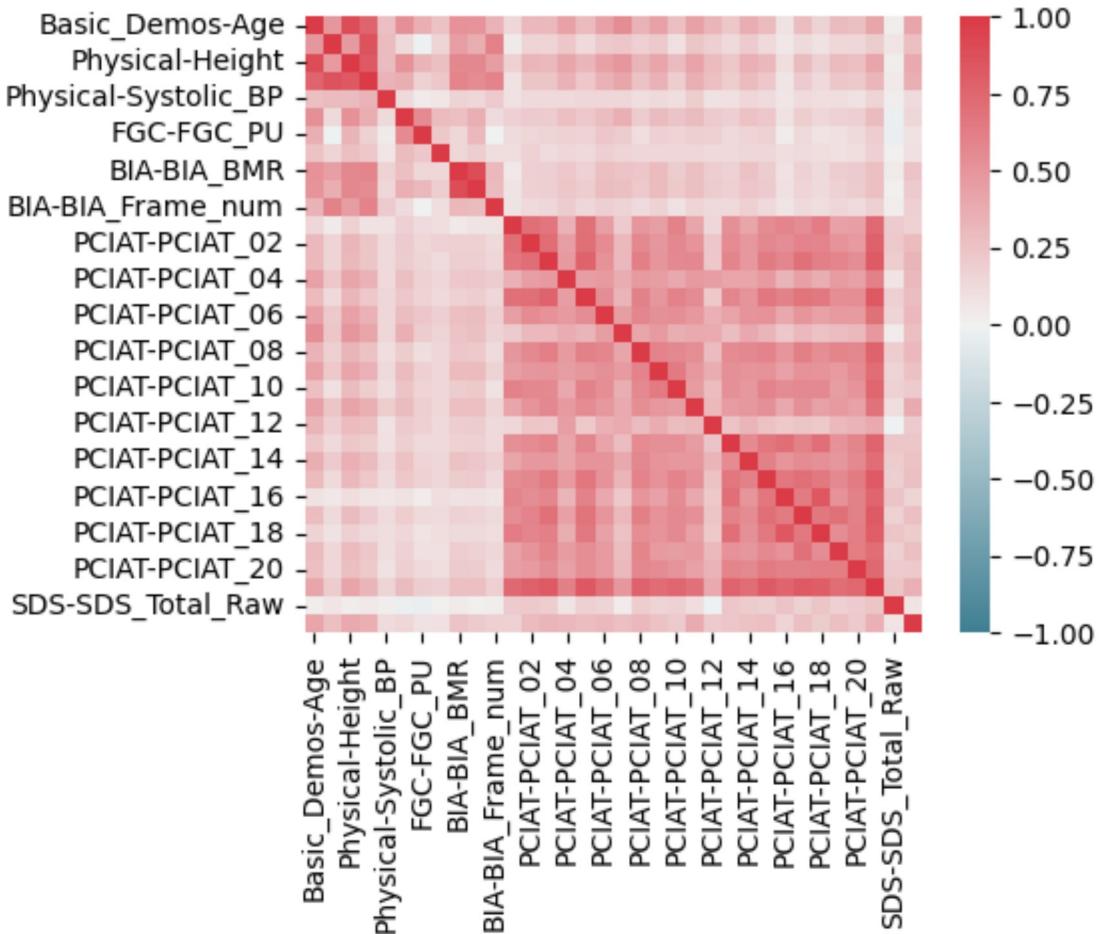
alleneremo e metteremo a confronto due modelli. uno ad albero e uno basato su rete neurale.

```
In [1]: 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)
```



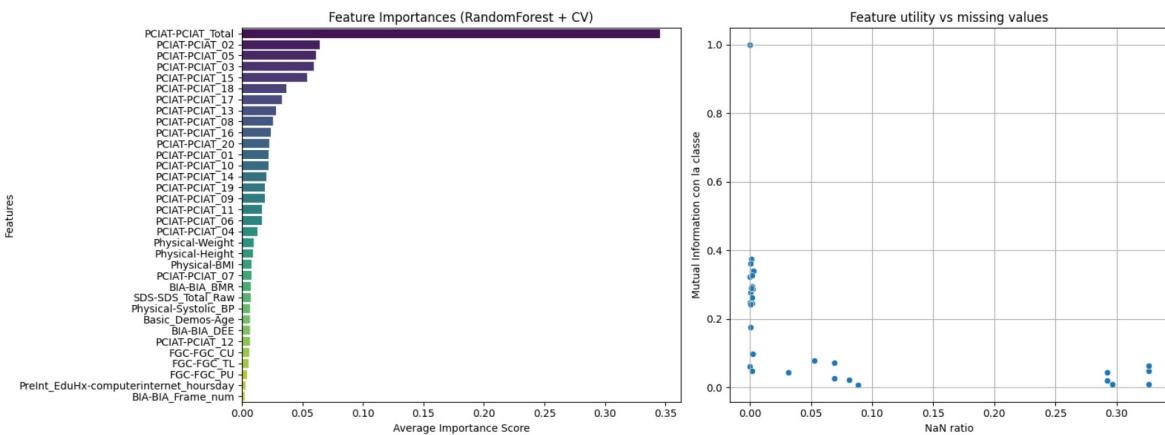
Per scremare ulteriormente le features (per avere maggior interpretabilità e minore complessità computazionale), abbiamo allenato una random forest in cross validation, per valutare la features importance. Abbiamo anche dato un'occhiata all'analisi statistica "mutual information", che non dipende dal 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)
```

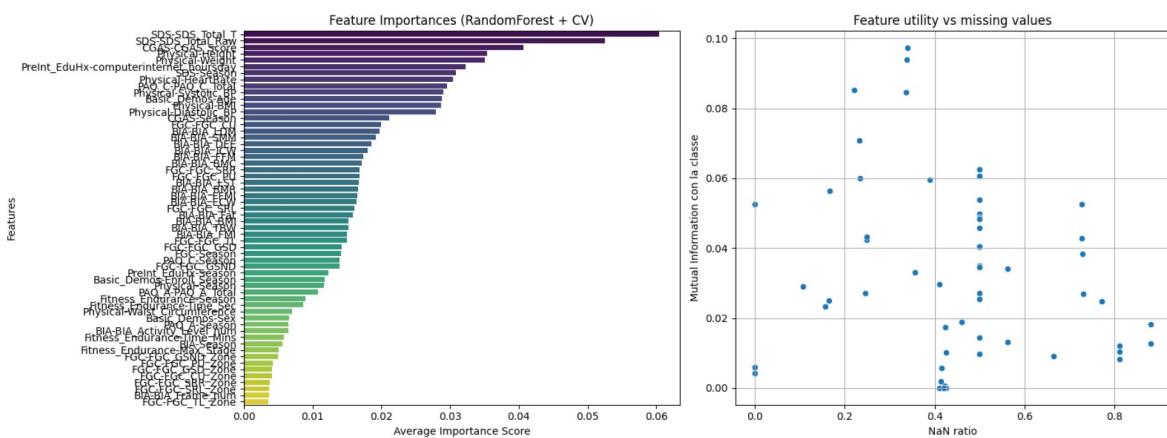


```
In [6]: feat_imp, feat_imp_sorted = feature_importance_and_mutual_info(X_train_final, y_
```

```
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(
```

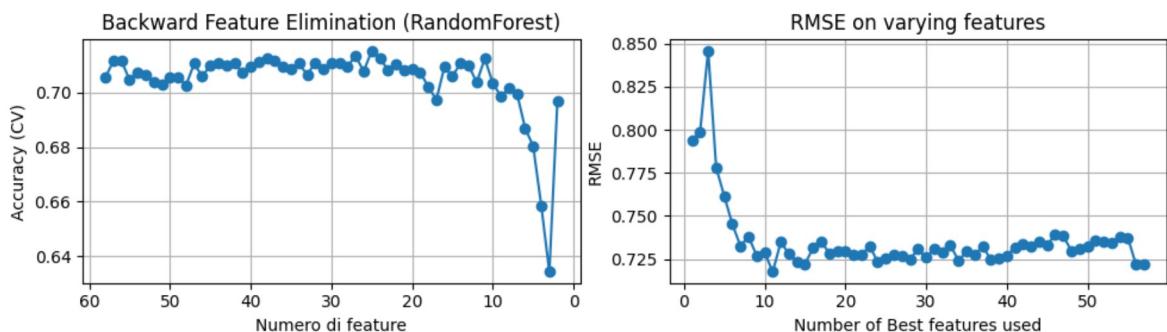


l'importance score è in generale molto più basso e decisamente meno sbilanciato di prima. Le features con NaN ratio alto non sono quelle con importance in assoluto minore, ma sono comunque nella bassa classifica, come atteso. Ugualmente alta mutual information corrisponde ad alta importance.

## 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

## roba da eliminaere

questa cella di codice in realtà è l'unione dei moduli, così da poterli modificare direttamente qui, in produzione va tolta

```
In [8]: from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import optuna
from xgboost import XGBClassifier
from sklearn.metrics import f1_score, accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from data_cleaning_module import season_map
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.ensemble import RandomForestRegressor
from sklearn.preprocessing import StandardScaler
import logging
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import joblib
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import classification_report, confusion_matrix, precision_recall_fscore_support
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import keras_tuner as kt
from tensorflow.keras.models import load_model

def tune_and_train_xgboost(X_train, y_train, X_val, y_val, num_classes, n_trials
```

```
class_counts = np.bincount(y_train)
total = len(y_train)
weights = {i: total / (len(class_counts) * c) for i, c in enumerate(class_counts)}
sample_weights = np.array([weights[label] for label in y_train])

def objective(trial):
    params = {
        'objective': 'multi:softprob',
        'num_class': num_classes,
        'tree_method': 'hist',
        'eval_metric': ['mlogloss', 'merror'],
        'max_depth': trial.suggest_int('max_depth', 2, 6),
        'learning_rate': trial.suggest_float('learning_rate', 0.001, 0.05, 1),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 20),
        'subsample': trial.suggest_float('subsample', 0.6, 1.0),
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.6, 1.0),
        'gamma': trial.suggest_float('gamma', 0, 5),
        'reg_alpha': trial.suggest_float('reg_alpha', 0, 5),
        'reg_lambda': trial.suggest_float('reg_lambda', 0, 5),
    }

    model = XGBClassifier(**params, n_estimators=1000, verbosity=0, early_stopping_rounds=10, eval_set=[(X_val, y_val)], eval_metric='mlogloss', max_depth=6, learning_rate=0.01, min_child_weight=1, subsample=0.6, colsample_bytree=0.6, gamma=0, reg_alpha=0, reg_lambda=0)
    model.fit(X_train, y_train, sample_weight=sample_weights, eval_set=[(X_val, y_val)], verbose=False)
    y_pred_val = model.predict(X_val)
    return f1_score(y_val, y_pred_val, average='weighted')

optuna.logging.set_verbosity(optuna.logging.WARNING)
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=n_trials)
print("Best hyperparameters:", study.best_params)

best_params = study.best_params
best_params.update({
    'objective': 'multi:softprob',
    'num_class': num_classes,
    'tree_method': 'auto',
    'eval_metric': ['mlogloss', 'merror']
})

model_final = XGBClassifier(**best_params, n_estimators=1000, verbosity=0, eval_set=[(X_train, y_train), (X_val, y_val)], eval_metric='mlogloss', max_depth=6, learning_rate=0.01, min_child_weight=1, subsample=0.6, colsample_bytree=0.6, gamma=0, reg_alpha=0, reg_lambda=0)
model_final.fit(X_train, y_train, sample_weight=sample_weights, eval_set=[(X_train, y_train), (X_val, y_val)], verbose=False)

model_final = XGBClassifier(**best_params, n_estimators=1000, verbosity=0, eval_set=[(X_train, y_train), (X_val, y_val)], eval_metric='mlogloss', max_depth=6, learning_rate=0.01, min_child_weight=1, subsample=0.6, colsample_bytree=0.6, gamma=0, reg_alpha=0, reg_lambda=0)
model_final.fit(X_train, y_train, sample_weight=sample_weights, eval_set=[(X_train, y_train), (X_val, y_val)], verbose=False)

evals_result = model_final.evals_result()

epochs = range(len(evals_result['validation_0']['mlogloss']))
fig, axes = plt.subplots(1, 3, figsize=(20,5))

axes[0].plot(epochs, evals_result['validation_0']['mlogloss'], label='Train')
axes[0].plot(epochs, evals_result['validation_1']['mlogloss'], label='Validation')
axes[0].set_xlabel("Epochs")
axes[0].set_ylabel("Log Loss")
axes[0].set_title("Training vs Validation Loss")
axes[0].legend()
axes[0].grid(True)
```

```
train_acc = [1 - x for x in evals_result['validation_0']['merror']]
val_acc = [1 - x for x in evals_result['validation_1']['merror']]
axes[1].plot(epochs, train_acc, label='Train Accuracy')
axes[1].plot(epochs, val_acc, label='Validation Accuracy')
axes[1].set_xlabel("Epochs")
axes[1].set_ylabel("Accuracy")
axes[1].set_title("Training vs Validation Accuracy")
axes[1].legend()
axes[1].grid(True)

y_pred_val = model_final.predict(X_val)
cm_val = confusion_matrix(y_val, y_pred_val)
sns.heatmap(cm_val, annot=True, fmt="d", cmap="Greens", cbar=False, ax=axes[2])
axes[2].set_title("Confusion Matrix - Validation")
axes[2].set_xlabel("Predicted")
axes[2].set_ylabel("True")

print("== Validation Set ==")
print("Accuracy:", accuracy_score(y_val, y_pred_val))
print("Weighted F1:", f1_score(y_val, y_pred_val, average='weighted'))

# Classification report come tabella grafica
report_dict = classification_report(y_val, y_pred_val, output_dict=True)
report_df = pd.DataFrame(report_dict).transpose().round(2)

plt.figure(figsize=(10, report_df.shape[0]*0.6))
sns.heatmap(report_df.iloc[:-1, :-1], annot=True, cmap="Blues", cbar=False,
plt.title("Classification Report - Validation Set")
plt.ylabel("Classes")
plt.xlabel("Metrics")

plt.tight_layout()
plt.show()

return model_final, study

def test_xgboost(model, X_test, y_test, classes=["0", "1", "2", "3"], SEED=42):

    X_test = season_map(X_test)
    Imputer = IterativeImputer(
        estimator=RandomForestRegressor(n_estimators=50, random_state=SEED),
        max_iter=10,
        random_state=SEED
    )

    X_test_imputed = Imputer.fit_transform(X_test)

    scaler = StandardScaler()
    X_test_scaled = scaler.fit_transform(X_test_imputed)

    y_pred = model.predict(X_test_scaled)
    print("== XGBoost - Classification Report (Test) ==")
    print(classification_report(y_test, y_pred, digits=4))
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
                xticklabels=classes, yticklabels=classes)
    plt.xlabel("Predicted")
    plt.ylabel("True")
```

```
plt.title("XGBoost Confusion Matrix (Test)")
plt.show()

def prepare_X (X_tr, X_val, y_tr, y_val, SEED):
    tf.random.set_seed(SEED)

    scaler = StandardScaler()

    # uso il train per calcolare media e deviazione standard, e standardizzo tutto
    # su train e validation

    X_tr_filled = X_tr.fillna(-9999).copy()
    X_val_filled = X_val.fillna(-9999).copy()

    X_tr_scaled = scaler.fit_transform(X_tr_filled)
    X_val_scaled = scaler.transform(X_val_filled)

    classes = np.unique(y_tr) # array delle classi presenti nel training set
    num_classes = len(classes)

    auto_weights = compute_class_weight(class_weight='balanced',
                                         classes=classes,
                                         y=y_tr)
    class_weights = {cls: w for cls, w in zip(classes, auto_weights)}

    return X_tr_scaled, X_val_scaled, class_weights

def build_model(n_features,
                n_classes = 4,
                hidden_units=[64,32],
                activation='relu',
                lr=1e-3,
                dropout=0.0,
                l2_reg=0.0,
                loss_case_binary = "binary_crossentropy",
                loss_case_multiclass = "sparse_categorical_crossentropy",
                loss_case_regression="mse",
                metrics=['accuracy', ]):

    # Definisce il Layer di input, dimensione = numero di feature
    inputs = layers.Input(shape=(n_features,))

    x = inputs    # Variabile temporanea per "costruire" i layer successivi
    for units in hidden_units:

        x = layers.Dense(units, # denso -> fully-connected
                         activation = activation,
                         kernel_regularizer = keras.regularizers.l2(l2_reg))(x)
        if dropout and dropout > 0.0:
            x = layers.Dropout(dropout)(x) # applico il dropout

    if n_classes == None:
        out = layers.Dense(1, activation='linear')(x)
        loss = loss_case_regression
    elif n_classes <= 2: # classificazione binaria
        out = layers.Dense(1, activation='sigmoid')(x)
        loss = loss_case_binary
    else: # classificazione multoclasse
        out = layers.Dense(n_classes, activation='softmax')(x)
        loss = loss_case_multiclass
```

```
model = keras.Model(inputs=inputs, outputs=out)
model.compile(optimizer=keras.optimizers.Adam(learning_rate=lr),
              loss=loss,
              metrics=metrics)

return model

def run_nn (X, Xv, y, yv, _num_features = 11, _num_classes = 4, hidden_units=[64
    """
    * _num_features: int, numero di feature in input
    * _num_classes: int, numero di classi in output (None per regressione)
    * hidden_units: lista di interi, numero di neuroni per layer nascosti
    * activation: stringa, funzione di attivazione da usare nei layer nascosti
    * lr: float, learning rate iniziale
    * dropout: float, percentuale di dropout da applicare
    * l2_reg: float, coefficiente di regolarizzazione L2
    * patience_early_stopping: int, numero di epoches senza miglioramento prima di
    * patience_reduce_lr: int, numero di epoches senza miglioramento prima di ridurre
    * factor: float, fattore di riduzione del learning rate se la loss di validazione
    * min_lr: float, learning rate minimo a cui ridurre
    * epochs: int, numero massimo di epoches per l'allenamento
    * batch_size: int, numero di campioni per batch
    * loss_case_binary: stringa, funzione di loss da usare per il caso binario
    * loss_case_multiclass: stringa, funzione di loss da usare per il caso multiclasse
    * loss_case_regression: stringa, funzione di loss da usare per il caso di regressione
    * metrics: lista di stringhe, metriche da usare per valutare il modello
    * weights: dizionario, pesi delle classi per il bilanciamento
    * classes: lista, nomi delle classi per la confusion matrix
    """
# -----
# modellazione effettiva
# -----
model = build_model(
    n_features=_num_features,           # input
    n_classes=_num_classes,             # output
    hidden_units=hidden_units,          # due con 64 e 32 neuroni
    activation=activation,              # f di attivazione
    lr=lr,                            # learning rate iniziale
    dropout=dropout,                  # dropout inizialmente 0
    l2_reg=l2_reg,                    # regolarizzazione L2 idem
    loss_case_binary=loss_case_binary, # loss per caso binario
    loss_case_multiclass=loss_case_multiclass, # loss per caso multiclasse
    loss_case_regression=loss_case_regression, # loss per caso regressione
    metrics=metrics,                  # metriche da usare
)
callbacks = [
    # early stop se la loss di validazione non migliora per tot epoches consecutive
    keras.callbacks.EarlyStopping(monitor='val_loss', patience=patience_early_stopping),

    # Se la loss di validazione non migliora per tot epoches, riduce il Learning Rate
    keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=factor, patience=patience_reduce_lr),
]
history = model.fit(
    X,
    y,
    validation_data=(Xv, yv),
    epochs=epochs,
```

```
batch_size=batch_size, # numero di campioni per batch
class_weight=weights, # pesi classi
callbacks=callbacks,
verbose=0
)

# -----
# risultati
# -----


print("training terminato")

plt.figure(figsize=(5,3))
plt.subplot(1,2,1)
plt.plot(history.history['loss'], label='train loss')
plt.plot(history.history['val_loss'], label='val loss')
plt.title('Loss per epoca'); plt.legend()

# Se è regressione, grafico MAE invece che accuracy
if _num_classes > 0:
    plt.subplot(1,2,2)
    plt.plot(history.history['accuracy'], label='train acc')
    plt.plot(history.history['val_accuracy'], label='val acc')
    plt.title('Accuracy per epoca'); plt.legend()
else:
    if 'mae' in history.history:
        plt.subplot(1,2,2)
        plt.plot(history.history['mae'], label='train MAE')
        plt.plot(history.history['val_mae'], label='val MAE')
        plt.title('MAE per epoca'); plt.legend()
plt.show()

# Predizioni di probabilità sul validation set
y_val_pred_proba = model.predict(Xv)

# Conversione da probabilità a classe predetta
if _num_classes == None:
    y_val_pred = np.int(y_val_pred_proba).astype(int).ravel()
    y_val_pred = np.clip(y_val_pred, 0, 3)
elif _num_classes <= 2:
    # Caso binario: threshold 0.5
    y_val_pred = (y_val_pred_proba.ravel() >= 0.5).astype(int)
else:
    # Caso multiclasse: argmax
    y_val_pred = np.argmax(y_val_pred_proba, axis=1)

# --- Report di classificazione ---
print("Classification report (validation):")
print(classification_report(yv, y_val_pred, digits=4))

# --- Confusion matrix ---
cm = confusion_matrix(yv, y_val_pred)
plt.figure(figsize=(5,3))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.xlabel('Predicted')          # asse X: predizioni
plt.ylabel('True')               # asse Y: valori reali
plt.title('Confusion Matrix (validation)')
plt.show()
```

```
if _num_classes and _num_classes > 3:
    # --- Recall specifico per classe 3 ---
    rec_severe = recall_score(yv, y_val_pred, labels=[3], average='macro')
    print(f"Recall classe severa (3): {rec_severe:.4f}")

return model

def impute_missing_values(X_tr, X_val, SEED):
    # Usa IterativeImputer con RandomForestRegressor per imputare i valori manca
    imputer = IterativeImputer(
        estimator=RandomForestRegressor(n_estimators=50, random_state=SEED),
        max_iter=10,
        random_state=SEED
    )
    X_tr_imputed = imputer.fit_transform(X_tr)
    X_val_imputed = imputer.transform(X_val)

    # eseguo quindi la standardizzazione
    scaler = StandardScaler()
    X_tr_scaled = scaler.fit_transform(X_tr_imputed)
    X_val_scaled = scaler.transform(X_val_imputed)

    return X_tr_scaled, X_val_scaled

def rf.Cascade(X_tr, y_tr, X_val, y_val, SEED):
    # y come classe binaria
    y_bin_tr = np.where(np.isin(y_tr, [0, 1]), 0, 1)
    y_bin_val = np.where(np.isin(y_val, [0, 1]), 0, 1)

    rf = RandomForestClassifier(
        random_state=SEED,
        class_weight='balanced',
        n_estimators=100,
        max_depth=10
    )
    rf.fit(X_tr, y_bin_tr)

    # Predizione RF: probabilità
    rf_probs_train = rf.predict_proba(X_tr)[:, 1].reshape(-1, 1)
    rf_probs_val = rf.predict_proba(X_val)[:, 1].reshape(-1, 1)

    # Predizione RF: classificazione (probabilità discretizzata)
    rf_pred_train = (rf_probs_train >= 0.5).astype(int).ravel()
    rf_pred_val = (rf_probs_val >= 0.5).astype(int).ravel()

    cm_train = confusion_matrix(y_bin_tr, rf_pred_train)
    cm_val = confusion_matrix(y_bin_val, rf_pred_val)

    print("== Confusion Matrix - TRAIN ==")
    print(cm_train)
    print("\n== Confusion Matrix - VAL ==")
    print(cm_val)

    print("\n== Classification Report - TRAIN ==")
    print(classification_report(y_bin_tr, rf_pred_train))
    print("\n== Classification Report - VAL ==")
    print(classification_report(y_bin_val, rf_pred_val))

    acc_train = accuracy_score(y_bin_tr, rf_pred_train)
```

```
acc_val = accuracy_score(y_bin_val, rf_pred_val)
print(f"TRAIN Accuracy RF: {acc_train:.4f}")
print(f"VAL Accuracy RF: {acc_val:.4f}")

plt.figure(figsize=(5,3))
sns.heatmap(cm_val, annot=True, fmt='d', cmap='Blues', xticklabels=['A','B'],
            xlabel="Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix RF - Validation")
plt.show()

# aggiungiamo probabilità RF come feature (continua)
X_bin_tr = np.hstack([X_tr, rf_probs_train])
X_bin_val = np.hstack([X_val, rf_probs_val])
n_features_aug = X_bin_tr.shape[1]

print("Shape train:", X_bin_tr.shape)
print("Shape val:", X_bin_val.shape)

return X_bin_tr, X_bin_val, n_features_aug

def tune_hyperparameters(X_tr, y_tr, X_val, y_val, _num_features, _num_classes,
                        **kwargs):
    """
    Funzione che esegue l'hyperparameter tuning usando Keras Tuner.
    """
    pass

def build_model(hp):
    """
    Funzione che costruisce un modello di rete neurale con iperparametri
    definiti da Keras Tuner.
    """

    model = tf.keras.Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape=(_num_features,)))

    # calibra il numero di neuroni nel primo Layer denso
    hp_units_1 = hp.Int('units_1', min_value=32, max_value=128, step=16)
    # calibra la funzione di attivazione
    hp_activation = hp.Choice('activation', values=['relu', 'tanh'])
    # calibra il valore di regolarizzazione L2
    hp_l2 = hp.Float('l2_reg', min_value=0.0000001, max_value=0.01, sampling_type='log')

    model.add(tf.keras.layers.Dense(units=hp_units_1, activation=hp_activation,
                                    kernel_regularizer=tf.keras.regularizers.

    # calibra il tasso di Dropout
    hp_dropout = hp.Float('dropout', min_value=0.1, max_value=0.5, step=0.1)
    model.add(tf.keras.layers.Dropout(hp_dropout))

    # Aggiungiamo un secondo Layer denso, trovando il numero di neuroni migliore
    hp_units_2 = hp.Int('units_2', min_value=16, max_value=64, step=16)
    model.add(tf.keras.layers.Dense(units=hp_units_2, activation=hp_activation,
                                    kernel_regularizer=tf.keras.regularizers.

    model.add(tf.keras.layers.Dense(_num_classes, activation='softmax'))

    # --- calibra il Learning Rate dell'ottimizzatore ---
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])
```

```
model.compile(  
    optimizer=tf.keras.optimizers.Adam(learning_rate=hp_learning_rate),  
    loss='sparse_categorical_crossentropy',  
    metrics=['accuracy'])  
  
return model  
  
# Definisce il tuner  
tuner = kt.RandomSearch(  
    build_model,  
    objective='val_accuracy',    # obiettivo: massimizzare accuracy sul valid  
    max_trials=100,              # Numero totale di combinazioni da provare  
    executions_per_trial=2,      # eseguire due volte il modello serve per qu  
    directory='my_keras_tuner',  # Cartella dove salvare i risultati  
    project_name='dwm_tuning')  
  
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)  
  
tuner.search(  
    X_tr, y_tr,  
    epochs=150, #non serve un numero eccessivo, basta che tenda a convergere  
    validation_data=(X_val, y_val),  
    callbacks=[stop_early],  
    class_weight=weights)  
)  
  
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]  
  
print(f"""  
--- RICERCA COMPLETATA ---  
  
I migliori iperparametri trovati sono:  
- Unità primo layer: {best_hps.get('units_1')}  
- Unità secondo layer: {best_hps.get('units_2')}  
- Funzione di attivazione: {best_hps.get('activation')}  
- Tasso di Dropout: {best_hps.get('dropout'):.2f}  
- Regolarizzazione L2: {best_hps.get('l2_reg'):.4f}  
- Learning Rate: {best_hps.get('learning_rate')}  
""")  
  
#-----  
# 5. ADDESTRAMENTO DEL MODELLO FINALE CON I PARAMETRI MIGLIORI  
#-----  
  
# Costruisci il modello con i migliori iperparametri trovati  
final_model = tuner.hypermodel.build(best_hps)  
  
# Ora addestra questo modello finale su TUTTI i dati di training (training +  
# per dargli più dati possibili prima del test finale.  
X_tr_full = pd.concatenate([X_tr, X_val])  
y_tr_full = pd.concatenate([y_tr, y_val])  
  
print("\n👉 Addestramento del modello finale con i parametri ottimali...")  
  
history = final_model.fit(  
    X_tr_full, y_tr_full,
```

```

    epochs=500, # Addestra per più epoch, l'early stopping si occuperà di f
    validation_data=(X_val, y_val),
    callbacks=[
        tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=75),
        tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2,
    ],
    class_weight=weights,
    verbose=0
)

final_model.save('neural_network_tuned_model.h5')

return X_tr_full, y_tr_full, final_model

```

**def test\_ann\_pipeline(X\_train, y\_train, X\_test, y\_test, SEED, final\_model, class**

# imputazione + scaling

X\_train\_scaled, X\_test\_scaled = impute\_missing\_values(X\_train, X\_test, SEED)

# RF cascade

X\_train\_aug, X\_test\_aug, n\_features\_aug = rf\_cascade(X\_train\_scaled, y\_train)

# predizione sul test

y\_pred\_proba = final\_model.predict(X\_test\_aug)

y\_pred = np.argmax(y\_pred\_proba, axis=1)

print("== ANN Ensemble - Classification Report (Test) ==")

print(classification\_report(y\_test, y\_pred, digits=4))

cm = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
 xticklabels=classes, yticklabels=classes)

plt.xlabel("Predicted")

plt.ylabel("True")

plt.title("ANN Confusion Matrix (Test)")

plt.show()

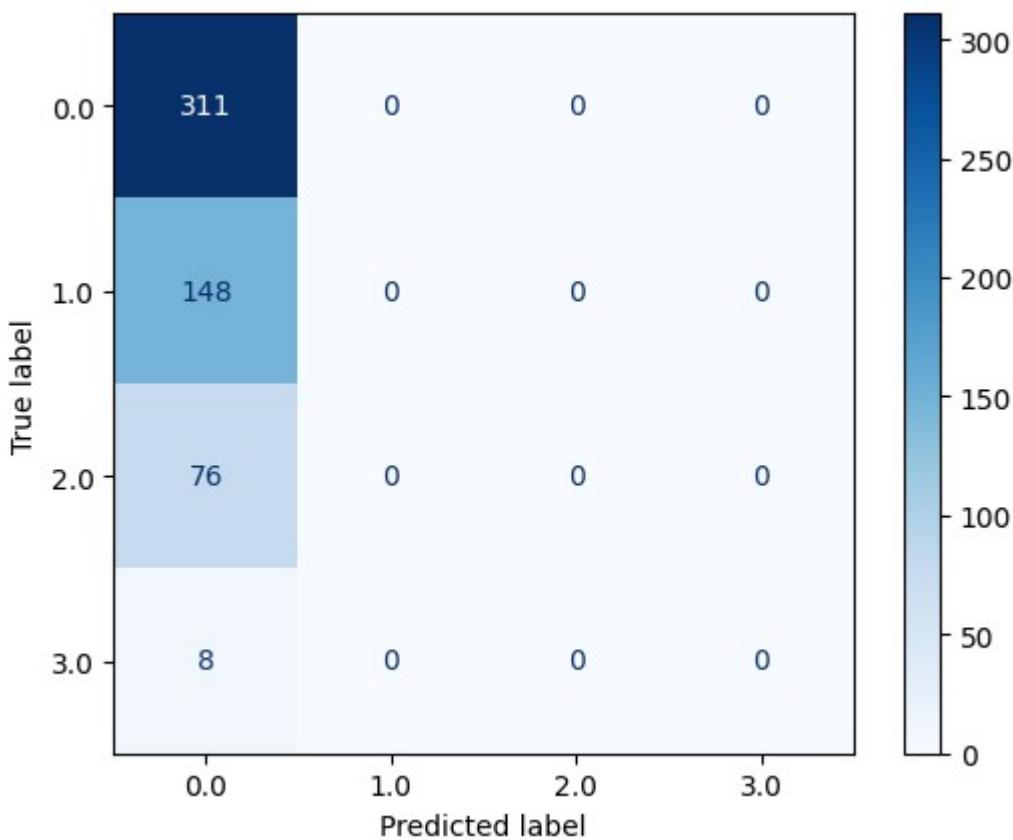
```
C:\Users\David\AppData\Local\Programs\Python\Python3.11_qbz5n2kf
ra8p0\LocalCache\local-packages\Python311\site-packages\tqdm\auto.py:21: TqdmWarning: IPProgress not found. Please update jupyter and ipywidgets. See https://ipyw
idgets.readthedocs.io/en/stable/user_install.html
from .autonotebook import tqdm as notebook_tqdm
```

## 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

## 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 [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_s
```

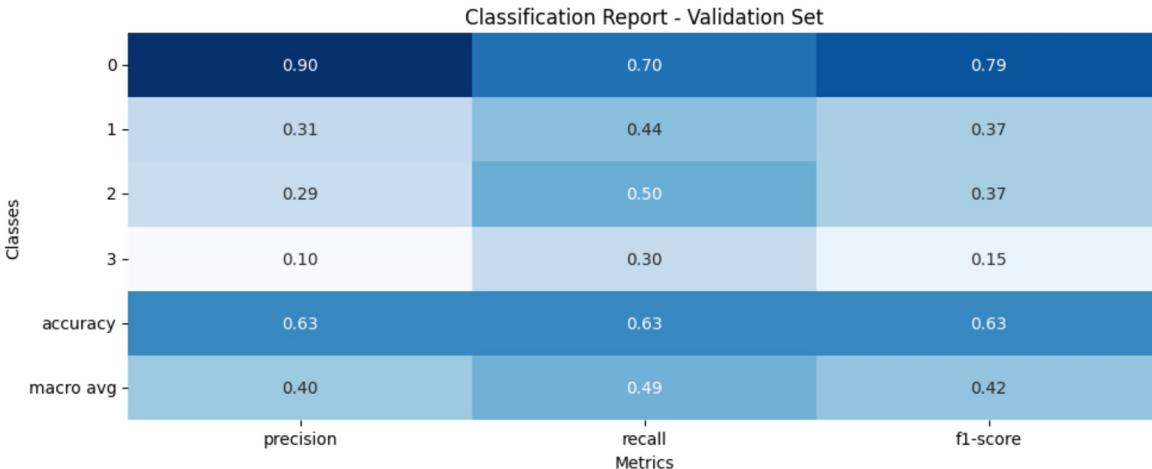
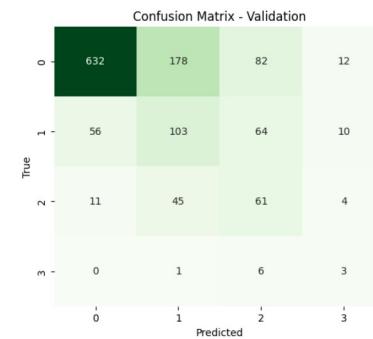
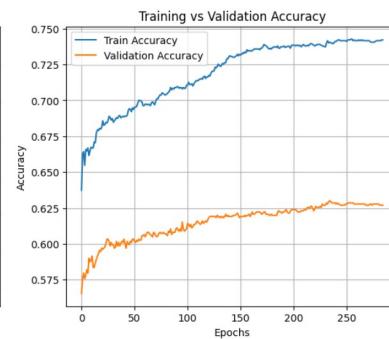
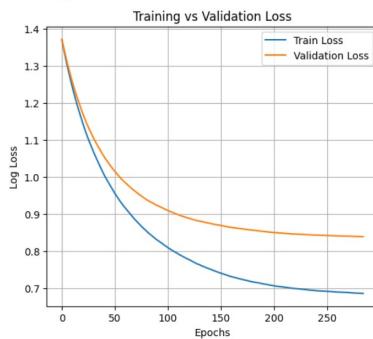
```
In [11]: print("== Model 1 ==")
model1, study = tune_and_train_xgboost(X_tr, y_tr, X_val, y_val, num_classes = 4
```

```
== Model 1 ==
Best hyperparameters: {'max_depth': 5, 'learning_rate': 0.0218016051837264, 'min_child_weight': 13, 'subsample': 0.9318721986717466, 'colsample_bytree': 0.9577335563027067, 'gamma': 1.5051811553899528, 'reg_alpha': 1.42561176235442, 'reg_lambda': 3.0600125615683194}
```

```
== Validation Set ==
```

```
Accuracy: 0.6301261829652997
```

```
Weighted F1: 0.6658282623300927
```



## 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 ==")
X_tr_scaled, X_val_scaled, class_weights = prepare_X(X_tr, X_val, y_tr, y_val, S
X_tr_scaled, X_val_scaled = impute_missing_values(X_tr, X_val, SEED)

== Model 2: Neural Network ==
C:\Users\David\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfr8p0\LocalCache\local-packages\Python311\site-packages\sklearn\impute\_iterativ
e.py:895: ConvergenceWarning: [IterativeImputer] Early stopping criterion not rea
ched.
    warnings.warn(
In [13]: # primo step random forest
X_tr_aug, X_val_aug, n_features_aug = rf_cascade(X_tr_scaled, y_tr, X_val_scaled

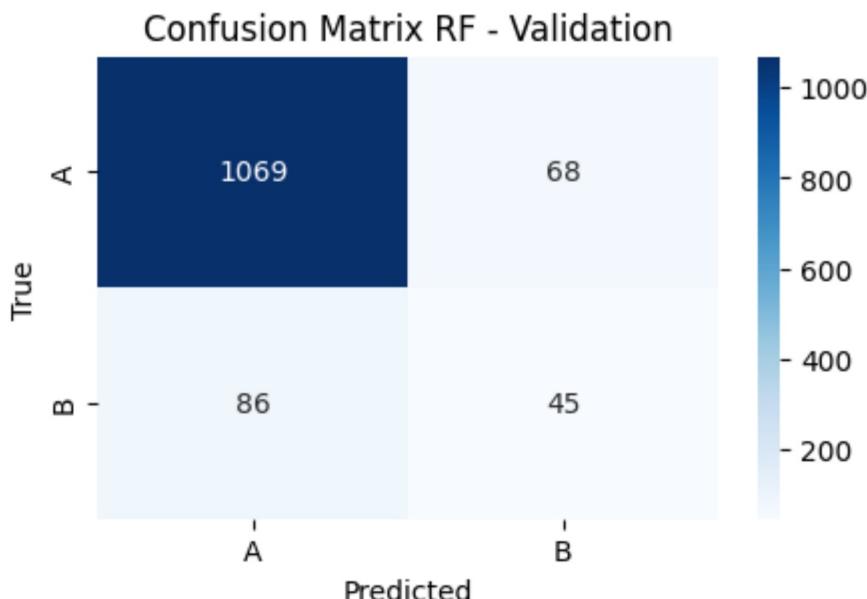
== Confusion Matrix - TRAIN ==
[[1670  33]
 [ 1 196]]

== 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 [14]: # secondo step tuning automatico
X_tr_full, y_tr_full, final_model = tune_hyperparameters(X_tr_aug, y_tr, X_val_a
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

C:\Users\David\AppData\Local\Programs\Python\Python3.11_qbz5n2kfr8p0\LocalCache\local-packages\Python311\site-packages\keras\src\layers\core\input_layer.py:27: UserWarning: Argument `input_shape` is deprecated. Use `shape` instead.
  warnings.warn(
```

```
-----  
AttributeError Traceback (most recent call last)  
Cell In[14], line 2  
      1 # secondo step tuning automatico  
----> 2 X_tr_full, y_tr_full, final_model = tune_hyperparameters(X_tr_aug, y_tr,  
X_val_aug, y_val, n_features_aug, 4, class_weights)  
  
Cell In[8], line 513, in tune_hyperparameters(X_tr, y_tr, X_val, y_val, _num_features, _num_classes, weights)  
    508 final_model = tuner.hypermodel.build(best_hps)  
    511 # Ora addestra questo modello finale su TUTTI i dati di training (trainin  
g + validation)  
    512 # per dargli più dati possibili prima del test finale.  
--> 513 X_tr_full = pd.concatenate([X_tr, X_val])  
    514 y_tr_full = pd.concatenate([y_tr, y_val])  
    516 print("\n👉 Addestramento del modello finale con i parametri ottimal  
i...")  
  
AttributeError: module 'pandas' has no attribute 'concatenate'
```

## Test dei modelli

```
In [ ]: y_test
```

```
Out[ ]: 149      0  
1025     0  
1846      0  
325       0  
408       1  
..  
3548      0  
1226     0  
736       0  
3292     1  
927       0  
Name: sii, Length: 543, dtype: int64
```

```
In [ ]: test_xgboost(model1, X_test, y_test)
```

```
c:\Users\David\Uni_Lavoro\data-web-mining-project\src\data_cleaning_module.py:27: S  
ettingWithCopyWarning:  
A value is trying to be set on a copy of a slice from a DataFrame.  
Try using .loc[row_indexer,col_indexer] = value instead
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab  
le/user_guide/indexing.html#returning-a-view-versus-a-copy  
    X[col] = X[col].map(season_map)  
C:\Users\David\AppData\Local\ Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kf  
ra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\impute\_base.py:6  
37: UserWarning: Skipping features without any observed values: [6]. At least one  
non-missing value is needed for imputation with strategy='mean'.  
    warnings.warn(  
C:\Users\David\AppData\Local\ Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kf  
ra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\impute\_iterativ  
e.py:895: ConvergenceWarning: [IterativeImputer] Early stopping criterion not rea  
ched.  
    warnings.warn(
```

```
-----  
ValueError                                     Traceback (most recent call last)  
Cell In[32], line 1  
----> 1 test_xgboost(model1, X_test, y_test)  
  
Cell In[8], line 152, in test_xgboost(model, X_test, y_test, classes, SEED)
    149 scaler = StandardScaler()
    150 X_test_scaled = scaler.fit_transform(X_test_imputed)
--> 152 y_pred = model.predict(X_test_scaled)
    153 print("== XGBoost - Classification Report (Test) ==")
    154 print(classification_report(y_test, y_pred, digits=4))  
  
File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\xgboost\core.py:729, in requir
e_keyword_args.<locals>.throw_if.<locals>.inner_f(*args, **kwargs)
    727 for k, arg in zip(sig.parameters, args):
    728     kwargs[k] = arg
--> 729 return func(**kwargs)  
  
File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\xgboost\sklearn.py:1719, in XG
BClassifier.predict(self, X, output_margin, validate_features, base_margin, itera
tion_range)
    1708 @_deprecate_positional_args
    1709 def predict(
    1710     self,
    (...)> 1716     iteration_range: Optional[IterationRange] = None,
    1717 ) -> ArrayLike:
    1718     with config_context(verbosity=self.verbosity):
--> 1719         class_probs = super().predict(
    1720             X=X,
    1721             output_margin=output_margin,
    1722             validate_features=validate_features,
    1723             base_margin=base_margin,
    1724             iteration_range=iteration_range,
    1725         )
    1726         if output_margin:
    1727             # If output_margin is active, simply return the scores
    1728             return class_probs  
  
File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\xgboost\core.py:729, in requir
e_keyword_args.<locals>.throw_if.<locals>.inner_f(*args, **kwargs)
    727 for k, arg in zip(sig.parameters, args):
    728     kwargs[k] = arg
--> 729 return func(**kwargs)  
  
File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\xgboost\sklearn.py:1327, in XG
BModel.predict(self, X, output_margin, validate_features, base_margin, iteration_
range)
    1325 if self._can_use_inplace_predict():
    1326     try:
--> 1327         predts = self.get_booster().inplace_predict(
    1328             data=X,
    1329             iteration_range=iteration_range,
    1330             predict_type="margin" if output_margin else "value",
    1331             missing=self.missing,
    1332             base_margin=base_margin,
    1333             validate_features=validate_features,
```

```
1334         )
1335     if _is_cupy_alike(predts):
1336         cp = import_cupy()

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\xgboost\core.py:729, in require_
e_keyword_args.<locals>.throw_if.<locals>.inner_f(*args, **kwargs)
    727 for k, arg in zip(sig.parameters, args):
    728     kwargs[k] = arg
--> 729 return func(**kwargs)

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\xgboost\core.py:2677, in Boost
er.inplace_predict(self, data, iteration_range, predict_type, missing, validate_f
eatures, base_margin, strict_shape)
    2673     raise TypeError(
    2674         "`shape` attribute is required when `validate_features` is Tr
ue."
    2675     )
    2676     if len(data.shape) != 1 and self.num_features() != data.shape[1]:
-> 2677         raise ValueError(
    2678             f"Feature shape mismatch, expected: {self.num_features()}, "
    2679             f"got {data.shape[1]}"
    2680     )
    2682 if _is_np_array_like(data):
    2683     from .data import _ensure_np_dtype

ValueError: Feature shape mismatch, expected: 11, got 10
```

```
In [ ]: test_ann_pipeline(X_tr_full, y_tr_full, X_test, y_test, SEED, final_model)
```

```
C:\Users\David\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\utils\validatio
n.py:2742: UserWarning: X has feature names, but IterativeImputer was fitted with
out feature names
  warnings.warn(
```

```
-----  
ValueError                                     Traceback (most recent call last)  
Cell In[17], line 1  
----> 1 test_ann_pipeline(X_tr_full, y_tr_full, X_test, y_test, SEED, final_mode  
1)  
  
Cell In[8], line 537, in test_ann_pipeline(X_train, y_train, X_test, y_test, SEED  
, final_model, classes)  
    535 def test_ann_pipeline(X_train, y_train, X_test, y_test, SEED, final_model  
, classes=["0","1","2","3"]):  
    536     # imputazione + scaling  
--> 537     X_train_scaled, X_test_scaled = impute_missing_values(X_train, X_te  
st, SEED)  
    539     # RF cascade  
    540     X_train_aug, X_test_aug, n_features_aug = rf_cascade(X_train_scaled,  
y_train, X_test_scaled, y_test, SEED)  
  
Cell In[8], line 356, in impute_missing_values(X_tr, X_val, SEED)  
    350 imputer = IterativeImputer(  
    351     estimator=RandomForestRegressor(n_estimators=50, random_state=SEED),  
    352     max_iter=10,  
    353     random_state=SEED  
    354 )  
    355 X_tr_imputed = imputer.fit_transform(X_tr)  
--> 356 X_val_imputed = imputer.transform(X_val)  
    358 # eseguo quindi la standardizzazione  
    359 scaler = StandardScaler()  
  
File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0  
\LocalCache\local-packages\Python311\site-packages\sklearn\utils\_set_output.py:3  
16, in _wrap_method_output.<locals>.wrapped(self, X, *args, **kwargs)  
314 @wraps(f)  
315 def wrapped(self, X, *args, **kwargs):  
--> 316     data_to_wrap = f(self, X, *args, **kwargs)  
317     if isinstance(data_to_wrap, tuple):  
318         # only wrap the first output for cross decomposition  
319         return_tuple = (  
320             _wrap_data_with_container(method, data_to_wrap[0], X, self),  
321             *data_to_wrap[1:],  
322         )  
  
File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0  
\LocalCache\local-packages\Python311\site-packages\sklearn\impute\_iterative.py:9  
21, in IterativeImputer.transform(self, X)  
904 """Impute all missing values in `X`.  
905  
906 Note that this is stochastic, and that if `random_state` is not fixed,  
(...) 917     The imputed input data.  
918 """  
919 check_is_fitted(self)  
--> 921 X, Xt, mask_missing_values, complete_mask = self._initial_imputation(  
922     X, in_fit=False  
923 )  
925 X_indicator = super()._transform_indicator(complete_mask)  
927 if self.n_iter_ == 0 or np.all(mask_missing_values):  
  
File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0  
\LocalCache\local-packages\Python311\site-packages\sklearn\impute\_iterative.py:6  
27, in IterativeImputer._initial_imputation(self, X, in_fit)  
624 else:
```

```
625     ensure_all_finite = True
--> 627 X = validate_data(
628     self,
629     X,
630     dtype=FLOAT_DTYPES,
631     order="F",
632     reset=in_fit,
633     ensure_all_finite=ensure_all_finite,
634 )
635 _check_inputs_dtype(X, self.missing_values)
637 X_missing_mask = _get_mask(X, self.missing_values)

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\sklearn\utils\validation.py:29
75, in validate_data(_estimator, X, y, reset, validate_separately, skip_check_array, **check_params)
2972     out = X, y
2974 if not no_val_X and check_params.get("ensure_2d", True):
-> 2975     _check_n_features(_estimator, X, reset=reset)
2977 return out

File ~\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0
\LocalCache\local-packages\Python311\site-packages\sklearn\utils\validation.py:28
39, in _check_n_features(estimator, X, reset)
2836     return
2838 if n_features != estimator.n_features_in_:
-> 2839     raise ValueError(
2840         f"X has {n_features} features, but {estimator.__class__.__name__}"
"
2841         f"is expecting {estimator.n_features_in_} features as input."
2842     )

ValueError: X has 11 features, but IterativeImputer is expecting 12 features as input.
```