***PRE-PROCESSING***

1. ***Caricamento del dataset***

from google.colab import files

uploaded = files.upload()

import pandas as pd

# Caricamento del dataset dal file CSV

df = pd.read\_csv("heart\_disease\_health\_indicators\_BRFSS2015.csv")

# Visualizzazione della forma del dataset

print("Dimensioni del dataset:", df.shape)

# Visualizzazione dei tipi di dati

print("Tipi di variabili:\n", df.dtypes)

# Visualizzazione dei primi 5 record

print("Prime 5 righe del dataset:\n", df.head())

# Verifica della presenza di valori mancanti

print("Valori nulli per colonna:\n", df.isnull().sum())

# Mostra le prime righe

df.head()

***2) Separazione delle variabili indipendenti e target***

# Separazione tra features (X) e target (y)

X = df.drop(columns=['HeartDiseaseorAttack'])

y = df['HeartDiseaseorAttack']

# Controlli veloci

print("Shape X (features):", X.shape)

print("Shape y (target):", y.shape)

print("\nPrime colonne di X:", list(X.columns[:10]))

print("\nDistribuzione target (conteggi):")

print(y.value\_counts())

print("\nDistribuzione target (%):")

print((y.value\_counts(normalize=True)\*100).round(2))

***3) Train-Test Split***

from sklearn.model\_selection import train\_test\_split

# Suddivisione in training e test set (80%-20%)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, random\_state=42, stratify=y

)

print("Shape X\_train:", X\_train.shape)

print("Shape X\_test:", X\_test.shape)

print("Distribuzione target nel training set:")

print(y\_train.value\_counts(normalize=True) \* 100)

***4) Standardizzazione del BMI***

from sklearn.preprocessing import StandardScaler

# Creazione copia per non modificare direttamente il dataset

X\_train\_scaled = X\_train.copy()

X\_test\_scaled = X\_test.copy()

scaler = StandardScaler()

# Adattare lo scaler sul train e trasformiamo train e test

X\_train\_scaled['BMI'] = scaler.fit\_transform(X\_train[['BMI']])

X\_test\_scaled['BMI'] = scaler.transform(X\_test[['BMI']])

print("BMI normalizzato - media (train):", round(X\_train\_scaled['BMI'].mean(), 4))

print("BMI normalizzato - deviazione standard (train):", round(X\_train\_scaled['BMI'].std(), 4))

***5) Encoding delle variabili categorici***

ordinal\_cols = ['GenHlth', 'Education', 'Income', 'Age']

missing\_ordinal = [c for c in ordinal\_cols if c not in X\_train\_scaled.columns]

if missing\_ordinal:

print("\n[WARN] Colonne ordinali non trovate (verifica header):", missing\_ordinal)

***6) verifica varianza quasi nulla (feature poco informative)***

from sklearn.feature\_selection import VarianceThreshold

# Rimozione di feature costanti

selector = VarianceThreshold(threshold=0.0) # elimina solo variabili con varianza nulla

selector.fit(X\_train\_scaled)

# Maschera booleana delle feature mantenute

constant\_mask = selector.get\_support()

# Identificazione delle feature eliminate

dropped\_features = list(X\_train\_scaled.columns[~constant\_mask])

print("Feature eliminate (costanti):", dropped\_features)

***7) Oversampling opzionale (SMOTE) sul training***

# Installazione imbalanced-learn se non già presente

!pip install -U imbalanced-learn

from imblearn.over\_sampling import SMOTE

# Applicazione SMOTE SOLO sul training set

smote = SMOTE(random\_state=42)

X\_train\_smote, y\_train\_smote = smote.fit\_resample(X\_train\_scaled, y\_train)

# Verifica distribuzione dopo SMOTE

print("\n[INFO] Distribuzione target - TRAIN dopo SMOTE:")

print((y\_train\_smote.value\_counts(normalize=True) \* 100).round(2))

***8) Salvataggio artefatti su Storage (Colab/ content)***

import joblib

import pandas as pd

# Salvataggio dataset di training e test

train\_df = pd.DataFrame(X\_train\_smote, columns=X\_train\_scaled.columns)

train\_df['target'] = y\_train\_smote

train\_df.to\_csv("/content/train\_preprocessed.csv", index=False)

test\_df = pd.DataFrame(X\_test\_scaled, columns=X\_train\_scaled.columns)

test\_df['target'] = y\_test

test\_df.to\_csv("/content/test\_preprocessed.csv", index=False)

# Salvataggio dello scaler

joblib.dump(scaler, "/content/scaler.pkl")

# Salvataggio della lista delle feature

joblib.dump(list(X\_train\_scaled.columns), "/content/features\_list.pkl")

print("[INFO] Artefatti salvati con successo:")

print("- train\_preprocessed.csv")

print("- test\_preprocessed.csv")

print("- scaler.pkl")

print("- features\_list.pkl")

***9) Scarica file***

from google.colab import files

files.download("/content/train\_preprocessed.csv")

files.download("/content/test\_preprocessed.csv")

files.download("/content/scaler.pkl")

files.download("/content/features\_list.pkl")

***MODULO PREDITTIVO - ALGORITMO DI CLASSIFICAZIONE “RANDOM FOREST”***

1. ***SETUP AMBIENTE (LIBRERIE+CARTELLE)***

!pip install xgboost joblib pyarrow fastparquet --quiet

import os, json, joblib, pathlib, datetime, numpy as np, pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import StratifiedKFold, RandomizedSearchCV, cross\_validate

from sklearn.metrics import (roc\_auc\_score, roc\_curve, precision\_recall\_curve,

classification\_report, confusion\_matrix, auc,

precision\_score, recall\_score, f1\_score)

from sklearn.ensemble import RandomForestClassifier

from xgboost import XGBClassifier

np.random.seed(42)

# Dove salva output e figure (in Colab locale)

RUN\_ID = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

BASE\_OUT = pathlib.Path(f"./artifacts/{RUN\_ID}")

(BASE\_OUT / "random\_forest").mkdir(parents=True, exist\_ok=True)

(BASE\_OUT / "xgboost").mkdir(parents=True, exist\_ok=True)

print("Run folder:", BASE\_OUT)

***2) CARICAMENTO ARTEFATTI (UPLOAD DAL PC) IN MEMORIA E VERIFICA COERENZA***

from google.colab import files

uploaded = files.upload() # seleziona qui i 4 file dal tuo PC

# Riconoscere i file caricati in modo robusto (anche se hanno suffissi diversi)

def pick(name\_contains, exts=(".csv",".pkl",".joblib")):

for k in uploaded.keys():

if name\_contains in k and k.endswith(exts):

return k

raise ValueError(f"File con '{name\_contains}' non trovato tra: {list(uploaded.keys())}")

train\_path = pick("train\_preprocessed", exts=(".csv",))

test\_path = pick("test\_preprocessed", exts=(".csv",))

scaler\_path = pick("scaler", exts=(".pkl",".joblib"))

feat\_path = pick("features", exts=(".pkl",".joblib"))

print("Train CSV: ", train\_path)

print("Test CSV: ", test\_path)

print("Scaler: ", scaler\_path)

print("Features: ", feat\_path)

import pandas as pd, joblib, pickle, numpy as np

# 1) Leggere CSV

train\_df = pd.read\_csv(train\_path)

test\_df = pd.read\_csv(test\_path)

# 2) Caricare scaler

try:

scaler = joblib.load(scaler\_path)

except:

with open(scaler\_path, "rb") as f:

scaler = pickle.load(f)

# 3) Caricare lista feature

try:

feature\_names = joblib.load(feat\_path)

except:

with open(feat\_path, "rb") as f:

feature\_names = pickle.load(f)

print("Train shape:", train\_df.shape, "Test shape:", test\_df.shape)

print("Prime colonne:", train\_df.columns.tolist()[:8])

print("N. feature dichiarate:", len(feature\_names))

***3) SEPARAZIONE X E Y CON CONTROLLI ROBUSTI***

# individuare target

possible\_y = ["target", "HeartDiseaseorAttack", "HeartDisease"]

y\_col = next((c for c in possible\_y if c in train\_df.columns), None)

assert y\_col is not None, f"Colonna target non trovata. Disponibili: {train\_df.columns.tolist()}"

print("Target column:", y\_col)

# normalizzare {2:0} se necessario

def normalize\_y(s):

s = pd.Series(s).astype(int)

return s.replace({2:0}).values

y\_train = normalize\_y(train\_df[y\_col])

y\_test = normalize\_y(test\_df[y\_col])

# usare solo le feature presenti sia nella lista che nel dataframe

feature\_set = [c for c in feature\_names if c in train\_df.columns]

X\_train = train\_df[feature\_set].values

X\_test = test\_df[feature\_set].values

print("X\_train:", X\_train.shape, "X\_test:", X\_test.shape)

print("Positivi train (%):", round(100\*np.mean(y\_train), 2), " — Positivi test (%):", round(100\*np.mean(y\_test), 2))

***4) VALIDAZIONE INCROCIATA E FUNZIONI DI VALUTAZIONE***

cv = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

def evaluate\_on\_test(model\_name, y\_true, y\_proba, out\_dir):

out\_dir.mkdir(parents=True, exist\_ok=True)

# Pred a soglia 0.5

y\_pred = (y\_proba >= 0.5).astype(int)

# Metriche principali

metrics = {

"threshold": 0.5,

"accuracy": float((y\_pred == y\_true).mean()),

"roc\_auc": float(roc\_auc\_score(y\_true, y\_proba)),

"precision": float(precision\_score(y\_true, y\_pred, zero\_division=0)),

"recall": float(recall\_score(y\_true, y\_pred, zero\_division=0)),

"f1": float(f1\_score(y\_true, y\_pred, zero\_division=0))

}

# Report + CM

report\_dict = classification\_report(y\_true, y\_pred, output\_dict=True, zero\_division=0)

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

# ROC figure

fpr, tpr, \_ = roc\_curve(y\_true, y\_proba)

roc\_auc\_val = auc(fpr, tpr)

plt.figure()

plt.plot(fpr, tpr, label=f"AUC = {roc\_auc\_val:.3f}")

plt.plot([0,1],[0,1],'--',linewidth=1)

plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")

plt.title(f"ROC - {model\_name}")

plt.legend(loc="lower right")

plt.tight\_layout(); plt.savefig(out\_dir / "roc\_curve.png", dpi=150); plt.close()

# PR figure

prec, rec, \_ = precision\_recall\_curve(y\_true, y\_proba)

pr\_auc\_val = auc(rec, prec)

plt.figure()

plt.plot(rec, prec, label=f"PR AUC = {pr\_auc\_val:.3f}")

plt.xlabel("Recall"); plt.ylabel("Precision")

plt.title(f"Precision-Recall - {model\_name}")

plt.legend(loc="lower left")

plt.tight\_layout(); plt.savefig(out\_dir / "pr\_curve.png", dpi=150); plt.close()

# Aggiunta PR AUC alle metriche salvate

metrics["pr\_auc"] = float(pr\_auc\_val)

# Salvataggi

with open(out\_dir / "metrics.json", "w") as f:

json.dump({\*\*metrics,

"classification\_report": report\_dict,

"confusion\_matrix": cm.tolist()}, f, indent=2)

# CSV utili

pd.DataFrame(cm, index=["Actual\_0","Actual\_1"], columns=["Pred\_0","Pred\_1"])\

.to\_csv(out\_dir / "confusion\_matrix.csv")

pd.DataFrame(report\_dict).to\_csv(out\_dir / "classification\_report.csv")

return metrics

***5) RANDOM FOREST - RICERCA LIGHT + TRAINING+ VALUTAZIONE***

rf = RandomForestClassifier(random\_state=42, n\_jobs=-1)

rf\_param\_dist = {

"n\_estimators": np.arange(200, 601, 50),

"max\_depth": [None] + list(range(4, 21, 2)),

"min\_samples\_split": [2, 5, 10, 20],

"min\_samples\_leaf": [1, 2, 4, 8],

"max\_features": ["sqrt", None, 0.5]

}

rf\_search = RandomizedSearchCV(

estimator=rf,

param\_distributions=rf\_param\_dist,

n\_iter=25,

scoring="f1",

cv=cv,

n\_jobs=-1,

random\_state=42,

verbose=1

)

rf\_search.fit(X\_train, y\_train)

rf\_best = rf\_search.best\_estimator\_

print("RF best params:", rf\_search.best\_params\_)

# CV informativa (multi-metrica)

scoring = {"roc\_auc":"roc\_auc", "precision":"precision", "recall":"recall", "f1":"f1"}

rf\_cv = cross\_validate(rf\_best, X\_train, y\_train, cv=cv, scoring=scoring, n\_jobs=-1, return\_train\_score=False)

print({m: (rf\_cv["test\_"+m].mean(), rf\_cv["test\_"+m].std()) for m in scoring})

# Fit finale + test

rf\_best.fit(X\_train, y\_train)

rf\_proba\_test = rf\_best.predict\_proba(X\_test)[:,1]

# Salvataggi

joblib.dump(rf\_best, BASE\_OUT / "random\_forest" / "random\_forest.joblib")

rf\_metrics = evaluate\_on\_test("random\_forest", y\_test, rf\_proba\_test, BASE\_OUT / "random\_forest")

with open(BASE\_OUT / "random\_forest" / "cv\_summary.json","w") as f:

json.dump({m: {"mean": float(rf\_cv["test\_"+m].mean()), "std": float(rf\_cv["test\_"+m].std())} for m in scoring}, f, indent=2)

print("RF test metrics:", rf\_metrics)

***5.1) PATCH per errore rf\_search.best\_params\_***

# PATCH salvataggi "robusti" e consistenti

import json, joblib, numpy as np, pandas as pd, matplotlib.pyplot as plt

from pathlib import Path

from sklearn.metrics import classification\_report, confusion\_matrix, roc\_auc\_score, roc\_curve, precision\_recall\_curve, auc

out\_dir = Path("rf\_results")

out\_dir.mkdir(exist\_ok=True)

# best\_params\_ in tipi Python "puri"

def to\_py(obj):

if isinstance(obj, (np.integer,)): return int(obj)

if isinstance(obj, (np.floating,)): return float(obj)

if isinstance(obj, (np.bool\_,)): return bool(obj)

return obj

best\_params\_py = {k: to\_py(v) for k, v in rf\_search.best\_params\_.items()}

# Salvataggio modello + iperparametri (usa rf\_best, non best\_rf)

joblib.dump(rf\_best, out\_dir / "random\_forest\_model.pkl")

with open(out\_dir / "best\_params.json", "w") as f:

json.dump(best\_params\_py, f, indent=2)

# Ricalcolare pred, cm, report e curve sul test (riuso y\_test / rf\_proba\_test)

y\_prob = rf\_proba\_test

y\_pred = (y\_prob >= 0.5).astype(int)

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

roc\_auc = float(roc\_auc\_score(y\_test, y\_prob))

report\_dict = classification\_report(y\_test, y\_pred, output\_dict=True, zero\_division=0)

# ROC

fpr, tpr, \_ = roc\_curve(y\_test, y\_prob)

plt.figure(figsize=(6,5))

plt.plot(fpr, tpr, label=f"AUC = {auc(fpr,tpr):.3f}")

plt.plot([0,1],[0,1],'--',linewidth=1)

plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate"); plt.title("ROC - Random Forest"); plt.legend()

plt.tight\_layout(); plt.savefig(out\_dir / "roc\_curve.png", dpi=150); plt.close()

# PR

prec, rec, \_ = precision\_recall\_curve(y\_test, y\_prob)

plt.figure(figsize=(6,5))

plt.plot(rec, prec, label=f"PR AUC = {auc(rec,prec):.3f}")

plt.xlabel("Recall"); plt.ylabel("Precision"); plt.title("Precision-Recall - Random Forest"); plt.legend()

plt.tight\_layout(); plt.savefig(out\_dir / "pr\_curve.png", dpi=150); plt.close()

# Salvataggi JSON/CSV

with open(out\_dir / "metrics.json", "w") as f:

json.dump({

"threshold": 0.5,

"accuracy": float((y\_pred == y\_test).mean()),

"roc\_auc": roc\_auc,

"precision": float(precision\_score(y\_test, y\_pred, zero\_division=0)),

"recall": float(recall\_score(y\_test, y\_pred, zero\_division=0)),

"f1": float(f1\_score(y\_test, y\_pred, zero\_division=0)),

"classification\_report": report\_dict,

"confusion\_matrix": cm.tolist()

}, f, indent=2)

pd.DataFrame(cm, index=["Actual\_0","Actual\_1"], columns=["Pred\_0","Pred\_1"]).to\_csv(out\_dir / "confusion\_matrix.csv")

pd.DataFrame(report\_dict).to\_csv(out\_dir / "classification\_report.csv")

print(" Salvataggi completati in:", out\_dir)

**5.2) MIGLIORARE RECALL (SENZA LEAKAGE)**

# Dato che X\_test ha la distribuzione reale (non SMOTE), si crea da esso Validation e Test finale (50-50 stratificato). Scegliamo la soglia sulla validation e poi valutiamo sul test finale.

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import precision\_score, recall\_score, f1\_score, confusion\_matrix, classification\_report, roc\_curve, precision\_recall\_curve, auc

import numpy as np, json

from pathlib import Path

import matplotlib.pyplot as plt

# Cartella risultati RF già usata

OUT = Path("rf\_results")

# 1A) Split del test in validation e test finale (stessa distribuzione, niente leakage)

X\_val, X\_test\_final, y\_val, y\_test\_final = train\_test\_split(

X\_test, y\_test, test\_size=0.5, stratify=y\_test, random\_state=42

)

# 1B) Probabilità sul validation e scelta soglia

y\_val\_prob = rf\_best.predict\_proba(X\_val)[:,1]

def choose\_threshold(y\_true, y\_score, metric="recall", min\_precision=None, max\_fpr=None):

thresholds = np.linspace(0.05, 0.95, 181)

best\_t, best\_val = 0.5, -1

for t in thresholds:

y\_pred = (y\_score >= t).astype(int)

tp = ((y\_true==1)&(y\_pred==1)).sum()

fp = ((y\_true==0)&(y\_pred==1)).sum()

fn = ((y\_true==1)&(y\_pred==0)).sum()

tn = ((y\_true==0)&(y\_pred==0)).sum()

prec = tp / (tp+fp) if (tp+fp)>0 else 0.0

rec = tp / (tp+fn) if (tp+fn)>0 else 0.0

fpr = fp / (fp+tn) if (fp+tn)>0 else 0.0

if min\_precision is not None and prec < min\_precision:

continue

if max\_fpr is not None and fpr > max\_fpr:

continue

# metrica di ottimizzazione (qui massimizziamo il Recall)

val = rec if metric=="recall" else (2\*prec\*rec/(prec+rec) if (prec+rec)>0 else 0.0)

if val > best\_val:

best\_val, best\_t = val, float(t)

return best\_t, best\_val

# Esempio: massimizza Recall mantenendo Precision almeno 0.25

t\_opt, best\_recall = choose\_threshold(y\_val, y\_val\_prob, metric="recall", min\_precision=0.25)

print(f"Soglia scelta su validation: {t\_opt:.3f} | Recall(validation)={best\_recall:.3f}")

# Valutazione sul test finale (quello "vero")

y\_test\_final\_prob = rf\_best.predict\_proba(X\_test\_final)[:,1]

y\_test\_final\_pred\_default = (y\_test\_final\_prob >= 0.5).astype(int)

y\_test\_final\_pred\_opt = (y\_test\_final\_prob >= t\_opt).astype(int)

def summarize(y\_true, y\_pred, y\_score):

pr, rc, f1 = precision\_score(y\_true, y\_pred, zero\_division=0), recall\_score(y\_true, y\_pred, zero\_division=0), f1\_score(y\_true, y\_pred, zero\_division=0)

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

fpr, tpr, \_ = roc\_curve(y\_true, y\_score)

return {

"precision": float(pr),

"recall": float(rc),

"f1": float(f1),

"cm": cm.tolist(),

"roc\_auc": float(auc(fpr, tpr))

}

metrics\_default = summarize(y\_test\_final, y\_test\_final\_pred\_default, y\_test\_final\_prob)

metrics\_opt = summarize(y\_test\_final, y\_test\_final\_pred\_opt, y\_test\_final\_prob)

print("== Test finale (soglia 0.5) ==>", metrics\_default)

print("== Test finale (soglia opt.) ==>", metrics\_opt)

# Salva tutto

with open(OUT / "threshold\_selection.json", "w") as f:

json.dump({

"validation\_threshold": t\_opt,

"validation\_constraint": "maximize recall with precision >= 0.25",

"test\_final\_metrics\_default": metrics\_default,

"test\_final\_metrics\_opt": metrics\_opt

}, f, indent=2)

# Grafico Precision-Recall con punto soglia

prec, rec, thr = precision\_recall\_curve(y\_test\_final, y\_test\_final\_prob)

pr\_auc = auc(rec, prec)

plt.figure(figsize=(6,5))

plt.plot(rec, prec, label=f"PR curve (AUC={pr\_auc:.3f})")

# punto corrispondente a t\_opt

# (troviamo il punto più vicino a t\_opt tra le soglie restituite)

thr\_full = np.r\_[thr, 1.0] # allinea dimensioni a rec/prec

idx = (np.abs(thr\_full - t\_opt)).argmin()

plt.scatter(rec[idx], prec[idx], s=60, marker="o", label=f"Soglia ottimale ~{t\_opt:.2f}")

plt.xlabel("Recall"); plt.ylabel("Precision"); plt.title("Precision-Recall (test finale)")

plt.legend()

plt.tight\_layout(); plt.savefig(OUT / "precision\_recall\_threshold.png", dpi=150); plt.close()

print(" File salvati:", OUT / "threshold\_selection.json", "e PR curve aggiornata")

**5.3) Interpretabilità: importanza delle feature (grafico + csv)**

# Interpretabilità Random Forest: feature importances (robusto)

import os, json, joblib, numpy as np, pandas as pd, matplotlib.pyplot as plt

from pathlib import Path

OUT = Path("rf\_results")

OUT.mkdir(exist\_ok=True)

# Recupero importanze dal modello addestrato

importances = rf\_best.feature\_importances\_

# Recupero dei nomi delle feature in modo robusto (in ordine di priorità):

feat\_names = None

# Se già in memoria la lista caricata dagli artefatti (features\_list.pkl)

if 'feature\_names' in globals() and isinstance(feature\_names, (list, tuple)) and len(feature\_names) == len(importances):

feat\_names = list(feature\_names)

# Se X\_train è un DataFrame, usa le colonne

elif 'X\_train' in globals():

try:

import pandas as pd

if isinstance(X\_train, pd.DataFrame) and len(X\_train.columns) == len(importances):

feat\_names = list(X\_train.columns)

except Exception:

pass

# Se hai ancora train\_df/test\_df (o un df con la target), prova a dedurle

if feat\_names is None:

for df\_name in ['train\_df', 'test\_df']:

if df\_name in globals():

\_df = globals()[df\_name]

if isinstance(\_df, pd.DataFrame):

candidates\_target = [c for c in ["target","HeartDiseaseorAttack","HeartDisease"] if c in \_df.columns]

if candidates\_target:

candidate\_feats = [c for c in \_df.columns if c not in candidates\_target]

else:

candidate\_feats = list(\_df.columns)

if len(candidate\_feats) == len(importances):

feat\_names = candidate\_feats

break

# Se proprio non troviamo nulla, crea nomi fittizi (ma segnala)

if feat\_names is None:

print(" Non sono riuscito a ricavare i nomi reali delle feature; uso nomi generici.")

feat\_names = [f"feat\_{i}" for i in range(len(importances))]

# Costruzione tabella importanze e salvataggio

fi = pd.DataFrame({"feature": feat\_names, "importance": importances}) \

.sort\_values("importance", ascending=False)

fi\_path = OUT / "feature\_importances.csv"

fi.to\_csv(fi\_path, index=False)

# Grafico TOP-K (barre orizzontali, solo matplotlib per massima compatibilità)

TOPK = 15 if len(fi) >= 15 else len(fi)

plt.figure(figsize=(8, 0.4\*TOPK + 2))

top = fi.head(TOPK)[::-1] # reverse per barh (dal basso verso alto)

plt.barh(top["feature"], top["importance"])

plt.xlabel("Importance")

plt.title(f"Random Forest - Top {TOPK} Feature Importances")

plt.tight\_layout()

plot\_path = OUT / "feature\_importances\_top.png"

plt.savefig(plot\_path, dpi=150)

plt.close()

print(" Salvati:")

print(" -", fi\_path)

print(" -", plot\_path)

***5.4) SALVARE OUTPUT***

import shutil

import pathlib

from google.colab import files

# Percorso alla cartella della Random Forest

rf\_out\_dir = BASE\_OUT / "random\_forest"

# Percorso zip temporaneo

zip\_path = pathlib.Path("/content/random\_forest\_results.zip")

# Crea il file zip con tutti i risultati

shutil.make\_archive(str(zip\_path).replace(".zip", ""), 'zip', rf\_out\_dir)

# Scarica il file sul PC

files.download(zip\_path)

print(f"File zip creato e pronto per il download: {zip\_path}”)

***MODULO PREDITTIVO - ALGORITMO DI CLASSIFICAZIONE “XGBOOST”***

# ==========================================================

# XGBOOST - PIPELINE COMPLETA (RICERCA, FIT, VALUTAZIONE, SALVATAGGI)

# ==========================================================

# !pip install xgboost joblib --quiet

import os, json, joblib, pickle, datetime, pathlib, gc

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from pathlib import Path

from google.colab import files

from sklearn.metrics import (

accuracy\_score, precision\_score, recall\_score, f1\_score,

roc\_auc\_score, average\_precision\_score,

roc\_curve, precision\_recall\_curve, confusion\_matrix, ConfusionMatrixDisplay,

classification\_report

)

from sklearn.model\_selection import RandomizedSearchCV, StratifiedKFold

from xgboost import XGBClassifier

# ----------------------------------------------------------

# Setup esecuzione e cartelle

# ----------------------------------------------------------

np.random.seed(42)

RUN\_ID = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

BASE\_OUT = Path(f"./artifacts/{RUN\_ID}")

XGB\_OUT = BASE\_OUT / "xgboost"

XGB\_OUT.mkdir(parents=True, exist\_ok=True)

print(" Run folder:", BASE\_OUT)

# ----------------------------------------------------------

# 1) Caricamento artefatti (upload dal PC) + pick robusto

# Attesi: train\_preprocessed.csv, test\_preprocessed.csv,

# features\_list.pkl (scaler.pkl opzionale)

# ----------------------------------------------------------

print(" Seleziona i file artefatto (train/test CSV + features\_list.pkl; scaler.pkl opzionale)...")

uploaded = files.upload()

def pick(name\_contains, exts=(".csv",".pkl",".joblib")):

for k in uploaded.keys():

if name\_contains in k and k.endswith(exts):

return k

raise ValueError(f"File con '{name\_contains}' non trovato. Presenti: {list(uploaded.keys())}")

train\_path = pick("train\_preprocessed", exts=(".csv",))

test\_path = pick("test\_preprocessed", exts=(".csv",))

feat\_path = pick("features", exts=(".pkl",".joblib"))

# scaler è opzionale per XGBoost (tree-based), lo carichiamo solo se c'è

scaler\_path = None

for k in uploaded.keys():

if ("scaler" in k) and k.endswith((".pkl",".joblib")):

scaler\_path = k

break

print(" Train CSV:", train\_path)

print(" Test CSV:", test\_path)

print(" Features :", feat\_path)

print(" Scaler :", scaler\_path if scaler\_path else "non usato per XGBoost")

# ----------------------------------------------------------

# Lettura CSV + feature list; costruzione X/y con controlli

# ----------------------------------------------------------

train\_df = pd.read\_csv(train\_path)

test\_df = pd.read\_csv(test\_path)

# Carica lista feature

try:

feature\_names = joblib.load(feat\_path)

except:

with open(feat\_path, "rb") as f:

feature\_names = pickle.load(f)

print("Train shape:", train\_df.shape, " | Test shape:", test\_df.shape)

print("Prime colonne train:", train\_df.columns.tolist()[:8])

print("N. feature dichiarate:", len(feature\_names))

# Individua colonna target in modo robusto

possible\_y = ["target", "HeartDiseaseorAttack", "HeartDisease"]

y\_col = next((c for c in possible\_y if c in train\_df.columns), None)

assert y\_col is not None, f"Colonna target non trovata. Disponibili: {train\_df.columns.tolist()}"

print(" Target column:", y\_col)

# Normalizza target {2->0} se presente questa codifica

def normalize\_y(s):

s = pd.Series(s).astype(int)

return s.replace({2:0}).values

y\_train = normalize\_y(train\_df[y\_col])

y\_test = normalize\_y(test\_df[y\_col])

# Seleziona solo le feature attese e presenti nei DF

feature\_set = [c for c in feature\_names if c in train\_df.columns]

assert len(feature\_set) > 0, "Nessuna feature valida trovata nel train\_df!"

X\_train = train\_df[feature\_set].astype(np.float32).values

X\_test = test\_df[feature\_set].astype(np.float32).values

print(" X\_train:", X\_train.shape, " | X\_test:", X\_test.shape)

print(" Positivi train (%):", round(100\*np.mean(y\_train), 2),

" — Positivi test (%):", round(100\*np.mean(y\_test), 2))

gc.collect()

# ----------------------------------------------------------

# Funzioni di valutazione + salvataggi (metriche, curve, CSV)

# ----------------------------------------------------------

def evaluate\_and\_save(model\_name, y\_true, y\_prob, out\_dir: Path, threshold=0.5):

"""

Calcola metriche, salva JSON/CSV, plotta e salva ROC/PR/Confusion Matrix.

Ritorna dict con metriche top-level.

"""

out\_dir.mkdir(parents=True, exist\_ok=True)

y\_pred = (y\_prob >= threshold).astype(int)

# Metriche top-level

metrics = {

"threshold": float(threshold),

"accuracy": float(accuracy\_score(y\_true, y\_pred)),

"roc\_auc": float(roc\_auc\_score(y\_true, y\_prob)),

"precision": float(precision\_score(y\_true, y\_pred, zero\_division=0)),

"recall": float(recall\_score(y\_true, y\_pred, zero\_division=0)),

"f1": float(f1\_score(y\_true, y\_pred, zero\_division=0)),

"pr\_auc": float(average\_precision\_score(y\_true, y\_prob))

}

# Report esteso e CM

report\_dict = classification\_report(y\_true, y\_pred, output\_dict=True, zero\_division=0)

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

# ROC

fpr, tpr, \_ = roc\_curve(y\_true, y\_prob)

plt.figure()

plt.plot(fpr, tpr, label=f"AUC = {metrics['roc\_auc']:.3f}")

plt.plot([0,1],[0,1],'--', linewidth=1)

plt.xlabel("False Positive Rate")

plt.ylabel("True Positive Rate")

plt.title(f"ROC Curve - {model\_name}")

plt.legend(loc="lower right")

plt.tight\_layout(); plt.savefig(out\_dir / "roc\_curve.png", dpi=150); plt.close()

# PR

prec, rec, \_ = precision\_recall\_curve(y\_true, y\_prob)

plt.figure()

plt.plot(rec, prec, label=f"PR AUC = {metrics['pr\_auc']:.3f}")

plt.xlabel("Recall")

plt.ylabel("Precision")

plt.title(f"Precision-Recall Curve - {model\_name}")

plt.legend(loc="lower left")

plt.tight\_layout(); plt.savefig(out\_dir / "pr\_curve.png", dpi=150); plt.close()

# Confusion Matrix

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=["No Disease", "Disease"])

disp.plot(cmap="Blues")

plt.title(f"Confusion Matrix - {model\_name} (thr={threshold})")

plt.tight\_layout(); plt.savefig(out\_dir / "confusion\_matrix.png", dpi=150); plt.close()

# Salvataggi

with open(out\_dir / "metrics.json", "w") as f:

json.dump({

\*\*metrics,

"classification\_report": report\_dict,

"confusion\_matrix": cm.tolist()

}, f, indent=2)

pd.DataFrame(cm, index=["Actual\_0","Actual\_1"], columns=["Pred\_0","Pred\_1"])\

.to\_csv(out\_dir / "confusion\_matrix.csv")

pd.DataFrame(report\_dict).to\_csv(out\_dir / "classification\_report.csv")

return metrics

# ----------------------------------------------------------

# 4) Spazio iperparametrico + RandomizedSearchCV

# (scoring=roc\_auc per confronto con RF / robustezza globale)

# ----------------------------------------------------------

param\_dist = {

"n\_estimators": [200, 300, 400],

"max\_depth": [4, 6, 8],

"learning\_rate": [0.05, 0.1],

"subsample": [0.8, 1.0],

"colsample\_bytree": [0.8, 1.0],

"gamma": [0, 0.1],

"min\_child\_weight": [1, 2],

"reg\_alpha": [0, 0.1],

"reg\_lambda": [1, 2],

"scale\_pos\_weight": [1.0] # il train è già bilanciato da SMOTE; 1.0 è coerente

}

xgb\_base = XGBClassifier(

objective="binary:logistic",

eval\_metric="logloss", # mettere qui (fit(...) su XGB 3.x non accetta eval\_metric=...)

n\_jobs=-1,

random\_state=42,

tree\_method="hist" # più veloce/memoria efficiente su Colab

)

cv = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

print(" Avvio RandomizedSearchCV per XGBoost...")

xgb\_search = RandomizedSearchCV(

estimator=xgb\_base,

param\_distributions=param\_dist,

n\_iter=25,

scoring="roc\_auc",

cv=cv,

n\_jobs=-1,

random\_state=42,

verbose=1

)

xgb\_search.fit(X\_train, y\_train)

best\_params\_raw = xgb\_search.best\_params\_

print(" Migliori iperparametri (CV):", best\_params\_raw)

# converti eventuali tipi numpy in tipi Python puri (per JSON)

def to\_py(o):

if isinstance(o, (np.integer,)): return int(o)

if isinstance(o, (np.floating,)): return float(o)

if isinstance(o, (np.bool\_,)): return bool(o)

return o

best\_params = {k: to\_py(v) for k, v in best\_params\_raw.items()}

with open(XGB\_OUT / "best\_params.json", "w") as f:

json.dump(best\_params, f, indent=2)

# ----------------------------------------------------------

# Fit finale con i migliori iperparametri

# ----------------------------------------------------------

xgb\_final = XGBClassifier(

\*\*best\_params,

objective="binary:logistic",

eval\_metric="logloss",

n\_jobs=-1,

random\_state=42,

tree\_method="hist"

)

print(" Fit finale su tutto il training set...")

xgb\_final.fit(X\_train, y\_train)

# ----------------------------------------------------------

# Valutazione su test (soglia standard 0.5) + salvataggi

# ----------------------------------------------------------

print(" Valutazione su test set (thr=0.5)...")

y\_prob = xgb\_final.predict\_proba(X\_test)[:,1]

xgb\_metrics = evaluate\_and\_save("XGBoost", y\_test, y\_prob, XGB\_OUT, threshold=0.5)

# Importanza feature (Gini gain di XGBoost)

try:

importances = xgb\_final.feature\_importances\_

fi = pd.DataFrame({"feature": feature\_set, "importance": importances})\

.sort\_values("importance", ascending=False)

fi.to\_csv(XGB\_OUT / "feature\_importances.csv", index=False)

TOPK = min(15, len(fi))

plt.figure(figsize=(8, 0.45\*TOPK + 2))

top = fi.head(TOPK)[::-1]

plt.barh(top["feature"], top["importance"])

plt.xlabel("Importance")

plt.title(f"XGBoost - Top {TOPK} Feature Importances")

plt.tight\_layout(); plt.savefig(XGB\_OUT / "feature\_importances\_top.png", dpi=150); plt.close()

except Exception as e:

print(" Impossibile salvare importanze feature:", e)

# Salva modello

joblib.dump(xgb\_final, XGB\_OUT / "xgb\_model.pkl")

print("\n Metriche TEST XGBoost:")

for k,v in xgb\_metrics.items():

if isinstance(v, float):

print(f" - {k}: {v:.6f}")

else:

print(f" - {k}: {v}")

print(f"\n File generati in: {XGB\_OUT.resolve()}")

# ----------------------------------------------------------

# ZIP dei risultati per download locale

# ----------------------------------------------------------

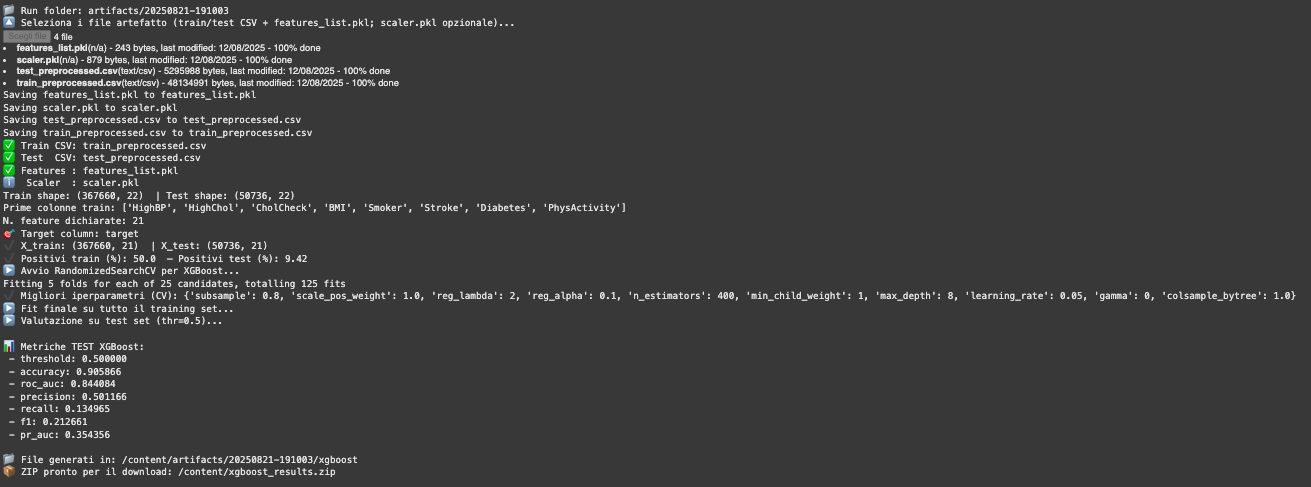
import shutil

zip\_path = Path("/content/xgboost\_results.zip")

shutil.make\_archive(str(zip\_path).replace(".zip",""), "zip", XGB\_OUT)

files.download(zip\_path)

print(f" ZIP pronto per il download: {zip\_path}")

******

***Blocco “robusto” per scelta soglia XGBoost***

# XGBoost: scelta soglia robusta + salvataggi ===

import json, pickle, pathlib

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import (

precision\_score, recall\_score, f1\_score, roc\_curve, auc,

precision\_recall\_curve, confusion\_matrix

)

# Percorsi (aggiorna la RUN se necessario)

ARTIFACTS\_DIR = pathlib.Path("/content/artifacts/20250821-191003/xgboost")

MODEL\_PATH = ARTIFACTS\_DIR / "xgb\_model.pkl"

OUT\_DIR = ARTIFACTS\_DIR / "thresholding"

OUT\_DIR.mkdir(parents=True, exist\_ok=True)

# Carica modello + dati pre-elaborati

with open(MODEL\_PATH, "rb") as f:

xgb\_model = pickle.load(f)

test\_df = pd.read\_csv("/content/test\_preprocessed.csv")

train\_df = pd.read\_csv("/content/train\_preprocessed.csv") # solo per coerenza di feature

target\_col = "target"

features = [c for c in test\_df.columns if c != target\_col]

X\_test\_full = test\_df[features].values

y\_test\_full = test\_df[target\_col].astype(int).values

# Split test -> validation + test\_final (50/50 stratificato)

X\_val, X\_test\_final, y\_val, y\_test\_final = train\_test\_split(

X\_test\_full, y\_test\_full, test\_size=0.5, stratify=y\_test\_full, random\_state=42

)

# Predizioni probabilistiche

y\_val\_prob = xgb\_model.predict\_proba(X\_val)[:, 1]

y\_test\_prob = xgb\_model.predict\_proba(X\_test\_final)[:, 1]

# Scansione soglie robusta

def scan\_thresholds(y\_true, y\_score, min\_precision=0.25, metric="recall"):

"""

Ritorna: df con soglia/prec/rec/f1 e (best\_t, best\_row)

metric: "recall" (consigliato in ambito clinico) oppure "f1"

"""

thresholds = np.linspace(0.01, 0.99, 99) # evita 0 e 1 estremi

rows = []

best\_val, best\_t, best\_row = -1.0, 0.5, None

for t in thresholds:

y\_pred = (y\_score >= t).astype(int)

# metriche dirette, zero\_division=0 evita warn/crash quando non ci sono predetti positivi

prec = precision\_score(y\_true, y\_pred, zero\_division=0)

rec = recall\_score(y\_true, y\_pred, zero\_division=0)

f1 = f1\_score(y\_true, y\_pred, zero\_division=0)

# vincolo (facoltativo): precisione minima

if prec < min\_precision:

rows.append({"threshold": t, "precision": prec, "recall": rec, "f1": f1, "valid": False})

continue

rows.append({"threshold": t, "precision": prec, "recall": rec, "f1": f1, "valid": True})

# funzione obiettivo

val = rec if metric == "recall" else f1

if val > best\_val:

best\_val, best\_t = val, float(t)

best\_row = {"threshold": best\_t, "precision": prec, "recall": rec, "f1": f1}

df = pd.DataFrame(rows)

return df, best\_t, best\_row

# Selezione soglia su validation (massimizza recall, precision >= 0.25)

scan\_df, t\_opt, best\_row = scan\_thresholds(y\_val, y\_val\_prob, min\_precision=0.25, metric="recall")

# Valutazione sul test finale alle due soglie

def eval\_at\_threshold(y\_true, y\_score, t):

y\_pred = (y\_score >= t).astype(int)

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

pr = precision\_score(y\_true, y\_pred, zero\_division=0)

rc = recall\_score(y\_true, y\_pred, zero\_division=0)

f1 = f1\_score(y\_true, y\_pred, zero\_division=0)

fpr, tpr, \_ = roc\_curve(y\_true, y\_score)

roc\_auc = auc(fpr, tpr)

return {

"threshold": float(t),

"precision": float(pr),

"recall": float(rc),

"f1": float(f1),

"roc\_auc": float(roc\_auc),

"confusion\_matrix": cm.tolist()

}

metrics\_default = eval\_at\_threshold(y\_test\_final, y\_test\_prob, 0.5)

metrics\_opt = eval\_at\_threshold(y\_test\_final, y\_test\_prob, t\_opt)

print(f"Soglia scelta su validation: {t\_opt:.3f} | Recall(val)={best\_row['recall']:.3f} | Precision(val)={best\_row['precision']:.3f}")

# Curve PR con marker soglia ottimale

prec, rec, thr = precision\_recall\_curve(y\_test\_final, y\_test\_prob)

pr\_auc = auc(rec, prec)

thr\_full = np.r\_[thr, 1.0]

idx = (np.abs(thr\_full - t\_opt)).argmin()

plt.figure(figsize=(9,6))

plt.plot(rec, prec, label=f"PR AUC = {pr\_auc:.3f}")

plt.scatter(rec[idx], prec[idx], s=70, label=f"Soglia ottimale ≈ {t\_opt:.2f}")

plt.xlabel("Recall"); plt.ylabel("Precision"); plt.title("Precision-Recall (XGBoost) - Test finale")

plt.legend()

plt.tight\_layout(); plt.savefig(OUT\_DIR / "precision\_recall\_threshold\_xgb.png", dpi=150); plt.close()

# Salvataggi

scan\_df.to\_csv(OUT\_DIR / "threshold\_scan\_xgb.csv", index=False)

with open(OUT\_DIR / "threshold\_selection\_xgb.json", "w") as f:

json.dump({

"picked\_threshold": t\_opt,

"validation\_best\_row": best\_row,

"test\_metrics\_default": metrics\_default,

"test\_metrics\_opt": metrics\_opt,

"pr\_auc\_test": float(pr\_auc)

}, f, indent=2)

print("== Test finale (thr=0.50) =>", metrics\_default)

print("== Test finale (thr≈opt) =>", metrics\_opt)

print(" Salvato:", OUT\_DIR)

immagine-incollata.png

***MODULO DI SIMILARITà***

1. ***Setup ambiente e librerie: installazione e importazione di librerie e si crea una cartella dedicata agli output di questo modulo. Si userà NearestNeighbors per cercare i K vicini e matplotlib per i grafici***

# Install e import

!pip install joblib --quiet

import os, json, glob, pickle, datetime, pathlib, shutil

import numpy as np

import pandas as pd

from sklearn.neighbors import NearestNeighbors

from google.colab import files

import matplotlib.pyplot as plt

np.random.seed(42)

# Cartelle output “run-specific”

RUN\_ID = datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

BASE\_OUT = pathlib.Path(f"./artifacts/{RUN\_ID}")

SIM\_OUT = BASE\_OUT / "similarity"

SIM\_OUT.mkdir(parents=True, exist\_ok=True)

print("Run folder:", BASE\_OUT)

***2) Caricamento artefatti e scelta del set di riferimento.***

***Per la similarità si vuole confrontare il paziente con casi reali. Dato che il train\_preprocessed.csv è bilanciato con SMOTE, si userà come corpus di riferimento il test set reale (non sintetico), così le vicinanze riflettono una popolazione “vera”. Carichiamo:***

* ***train\_preprocessed.csv (solo per completezza, non lo useremo come corpus);***
* ***test\_preprocessed.csv (corpus di riferimento);***
* ***features\_list.pkl (ordine/insieme delle feature);***
* ***scaler.pkl (StandardScaler del BMI).***

# Seleziona dal PC: train\_preprocessed.csv, test\_preprocessed.csv, scaler.pkl, features\_list.pkl

uploaded = files.upload()

def pick(name\_contains, exts=(".csv",".pkl",".joblib")):

for k in uploaded.keys():

if name\_contains in k and k.endswith(exts):

return k

raise ValueError(f"File con '{name\_contains}' non trovato. Caricati: {list(uploaded.keys())}")

train\_path = pick("train\_preprocessed", exts=(".csv",))

test\_path = pick("test\_preprocessed", exts=(".csv",))

scaler\_path = pick("scaler", exts=(".pkl",".joblib"))

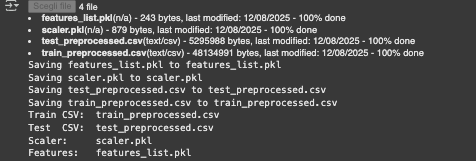
feat\_path = pick("features", exts=(".pkl",".joblib"))

print("Train CSV: ", train\_path)

print("Test CSV: ", test\_path)

print("Scaler: ", scaler\_path)

print("Features: ", feat\_path)



***3) Caricamento artefatti in memoria + individuazione target + X\_ref/y\_ref***

# Carica CSV

train\_df = pd.read\_csv(train\_path)

test\_df = pd.read\_csv(test\_path)

# Carica scaler

try:

import joblib

scaler = joblib.load(scaler\_path)

except Exception:

with open(scaler\_path, "rb") as f:

scaler = pickle.load(f)

# Carica lista feature

try:

feature\_names = joblib.load(feat\_path)

except Exception:

with open(feat\_path, "rb") as f:

feature\_names = pickle.load(f)

# Target

possible\_y = ["target", "HeartDiseaseorAttack", "HeartDisease"]

y\_col = next((c for c in possible\_y if c in test\_df.columns), None)

if y\_col is None:

raise ValueError(f"Colonna target non trovata. Colonne disponibili: {test\_df.columns.tolist()}")

def normalize\_y(s):

s = pd.Series(s).astype(int)

return s.replace({2:0}).values # prudenziale

y\_ref = normalize\_y(test\_df[y\_col])

# X\_ref = 21 feature

feature\_set = [c for c in feature\_names if c in test\_df.columns]

X\_ref = test\_df[feature\_set].copy()

print(f" Target individuata: {y\_col}")

print(" X\_ref shape:", X\_ref.shape)

print(" Distribuzione target (test):", dict(pd.Series(y\_ref).value\_counts().sort\_index()))

immagine-incollata.png

***4) Ricodifica pseudo-binarie in veri 1/0***

***Evitare il problema che alcune variabili hanno valori diversi da 0 e 1 in linea con la fonte del dataset***

# Dizionario binarie

BINARY\_CANDIDATES = [

"HighBP","HighChol","CholCheck","Smoker","Stroke","Diabetes",

"PhysActivity","Fruits","Veggies","HvyAlcoholConsump",

"AnyHealthcare","NoDocbcCost","DiffWalk","Sex"

]

def recode\_binary(df, cols):

cols = [c for c in cols if c in df.columns]

if not cols:

return df

# 1 se ==1, altrimenti 0

df.loc[:, cols] = (df[cols] == 1).astype(int)

return df

X\_ref = recode\_binary(X\_ref, BINARY\_CANDIDATES)

print("Ricodifica binarie → 0/1 completata su:", [c for c in BINARY\_CANDIDATES if c in X\_ref.columns][:10], "...")

immagine-incollata.png

***5) Normalizzazione per la distanza (BMI z-score + ordinali in [0,1]))***

***Quando vogliamo calcolare quanto due pazienti sono “simili”, usiamo una distanza matematica (nel nostro caso la distanza euclidea).  
Problema: se le variabili non sono tutte sulla stessa scala, una sola variabile può “dominare” la distanza e rendere i risultati distorti.***

***Esempio molto intuitivo:***

* ***la variabile BMI (Indice di Massa Corporea) può avere valori da 15 a 50,***
* ***la variabile Fumatore è solo 0 oppure 1.***

***Se non facciamo nulla, il BMI contribuisce molto di più al calcolo della distanza rispetto al fatto che uno sia fumatore o meno. Risultato: i vicini verrebbero scelti quasi solo sulla base del BMI, trascurando gli altri fattori di rischio.***

***Per evitare questo, normalizziamo tutte le variabili così che abbiano un peso comparabile:***

1. ***Variabili binarie (0/1)  
   Già perfette: non serve toccarle, perché “0” e “1” hanno lo stesso impatto di qualsiasi altra variabile normalizzata.***
2. ***BMI  
   Qui applichiamo lo StandardScaler salvato nel pre-processing: trasformiamo i valori in “z-score”, cioè quanto ciascun BMI si discosta dalla media in termini di deviazioni standard. Lo scaler.transform(df[["BMI"]]) produce un array float32 annidato ma la colonna BMI era float64 quindi è stato eseguito il cast a float64 al momento dell’assegnazione.***➝ ***Così un BMI di 30 non viene considerato “grande” solo perché il numero è alto, ma perché è più alto della media della popolazione.***
3. ***Variabili ordinali (scale finite)  
   Alcune variabili hanno scale numeriche diverse:***
   * ***GenHlth da 1 a 5,***
   * ***Age da 1 a 13,***
   * ***Income da 1 a 8,***
   * ***MentHlth da 0 a 30, ecc.***
4. ***Le dividiamo per il loro massimo teorico, così diventano tutte comprese tra 0 e 1.***➝ ***In questo modo, “età=13” diventa 1.0, “età=6” diventa circa 0.46.  
   Così l’età pesa come qualsiasi altro fattore in scala ridotta.***

# Scale teoriche per ordinali

ORDINAL\_SCALES = {

"GenHlth": 5, # 1..5

"Age": 13, # 1..13

"Education": 6, # 1..6

"Income": 8, # 1..8

"MentHlth": 30, # 0..30

"PhysHlth": 30 # 0..30

}

def normalize\_dataframe\_for\_distance(df, feature\_names, scaler, ordinal\_scales):

Xn = df[feature\_names].copy()

# BMI → z-score con scaler del pre-processing

if "BMI" in Xn.columns:

bmi\_df = Xn[["BMI"]].astype("float64")

Xn.loc[:, "BMI"] = pd.DataFrame(

scaler.transform(bmi\_df),

index=Xn.index,

columns=["BMI"]

).astype("float64")

# Ordinali → [0,1]

for col, mx in ordinal\_scales.items():

if col in Xn.columns:

Xn.loc[:, col] = Xn[col].astype("float64") / float(mx)

# Tutto float64 (sklearn compatibile)

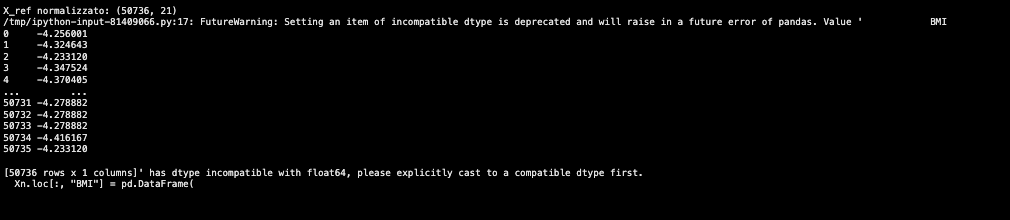
return Xn.astype("float64")

X\_ref\_norm = normalize\_dataframe\_for\_distance(

df=X\_ref.copy(), feature\_names=feature\_set, scaler=scaler, ordinal\_scales=ORDINAL\_SCALES

)

print(" X\_ref normalizzato:", X\_ref\_norm.shape)



***6) Fitting degli indici KNN (globale + solo positivi)***

# KNN globale

knn = NearestNeighbors(n\_neighbors=min(200, len(X\_ref\_norm)), metric="euclidean")

knn.fit(X\_ref\_norm.values)

# KNN solo positivi

mask\_pos = (y\_ref == 1)

X\_pos\_norm = X\_ref\_norm[mask\_pos]

knn\_pos = None

if X\_pos\_norm.shape[0] >= 1:

knn\_pos = NearestNeighbors(n\_neighbors=min(200, len(X\_pos\_norm)), metric="euclidean")

knn\_pos.fit(X\_pos\_norm.values)

print(f" KNN globale pronto con n={knn.n\_neighbors}")

print(f" KNN positivi: {'OK' if knn\_pos is not None else 'NON disponibile (nessun positivo?)'}")

immagine-incollata.png

***7) Utility: normalizzazione singolo paziente + analisi similarità***

# binarie effettive presenti (per fattori di rischio)

BINARY\_COLS = [c for c in BINARY\_CANDIDATES if c in feature\_set]

def normalize\_single\_patient\_for\_distance(p\_series, scaler, feature\_names, ordinal\_scales):

p = pd.DataFrame([p\_series.values], columns=feature\_names).copy()

if "BMI" in p.columns:

bmi\_df = p[["BMI"]].astype("float64")

p.loc[:, "BMI"] = pd.DataFrame(

scaler.transform(bmi\_df),

index=p.index,

columns=["BMI"]

).astype("float64")

for col, mx in ordinal\_scales.items():

if col in p.columns:

p.loc[:, col] = p[col].astype("float64") / float(mx)

return p.astype("float64")

def analyze\_similarity(paziente\_series, k\_total=50, k\_pos\_min=5, save\_json=True, out\_dir=SIM\_OUT):

"""

1) Trova k\_total vicini globali su X\_ref\_norm.

2) Se i positivi tra quei vicini < k\_pos\_min, aggiunge i più vicini dal KNN dei positivi.

3) Calcola % malattia (pesata per distanza) e fattori binari ricorrenti tra i positivi.

"""

# normalizza paziente

p\_norm = normalize\_single\_patient\_for\_distance(

p\_series=paziente\_series, scaler=scaler, feature\_names=feature\_set, ordinal\_scales=ORDINAL\_SCALES

).values

# vicini globali

d\_all, i\_all = knn.kneighbors(p\_norm, n\_neighbors=min(k\_total, len(X\_ref\_norm)), return\_distance=True)

d\_all = d\_all[0]; i\_all = i\_all[0]

y\_all = y\_ref[i\_all]

# garantisci almeno k\_pos\_min positivi (se possibile)

if (knn\_pos is not None) and (y\_all.sum() < k\_pos\_min):

need = int(k\_pos\_min - y\_all.sum())

need = max(0, min(need, X\_pos\_norm.shape[0]))

if need > 0:

d\_pos, i\_pos = knn\_pos.kneighbors(p\_norm, n\_neighbors=need, return\_distance=True)

i\_pos\_global = np.where(y\_ref == 1)[0][i\_pos[0]]

i\_all = np.r\_[i\_all, i\_pos\_global]

d\_all = np.r\_[d\_all, d\_pos[0]]

y\_all = y\_ref[i\_all]

# % malati pesata per distanza (pesi=1/(d+1e-6))

w = 1.0 / (d\_all + 1e-6)

perc\_malati = float((w \* y\_all).sum() / w.sum() \* 100.0)

# fattori binari ricorrenti tra i POSITIVI

fattori = {}

pos\_mask\_local = (y\_all == 1)

if pos\_mask\_local.any():

w\_pos = w[pos\_mask\_local]

idxpos = i\_all[pos\_mask\_local]

for col in BINARY\_COLS:

vals = X\_ref.loc[idxpos, col].values.astype(float) # ora veri 0/1

fattori[col] = float((w\_pos \* vals).sum() / (w\_pos.sum() + 1e-9))

fattori\_sorted = dict(sorted(fattori.items(), key=lambda kv: kv[1], reverse=True)[:10])

result = {

"k\_total": int(min(k\_total, len(X\_ref\_norm))),

"k\_pos\_min": int(k\_pos\_min),

"neighbors\_index": i\_all.tolist(),

"perc\_malati\_vicini": perc\_malati, # %

"fattori\_rischio\_top": fattori\_sorted

}

if save\_json:

out\_dir.mkdir(parents=True, exist\_ok=True)

pid = paziente\_series.name if paziente\_series.name is not None else "sample"

out\_path = out\_dir / f"similarity\_patient\_{pid}.json"

with open(out\_path, "w") as f:

json.dump(result, f, indent=2)

return result

***8) Esempio 1: un paziente del test per ID (con K alto)***

patient\_idx = int(np.random.choice(X\_ref.index.values, 1)[0])

paziente = X\_ref.loc[patient\_idx]

print("Paziente test index:", patient\_idx)

res = analyze\_similarity(paziente\_series=paziente, k\_total=50, k\_pos\_min=5, save\_json=True, out\_dir=SIM\_OUT)

print(f" % malattia tra i vicini (pesata): {res['perc\_malati\_vicini']:.2f}%")

print(" Fattori di rischio (top, 0..1):", res["fattori\_rischio\_top"])

immagine-incollata.png

***9) Analisi batch di N=10 pazienti (CSV riassunto + 10 JSON)***

import pandas as pd

import numpy as np

from pathlib import Path

def analyze\_similarity\_batch\_random(df, n=10, k\_total=200, k\_pos\_min=100, out\_dir=SIM\_OUT):

"""

Esegue l'analisi per n pazienti scelti CASUALMENTE dal DataFrame X\_ref.

Salva un CSV riassuntivo con:

- indice reale del paziente (quello del dataset),

- percentuale malati tra i vicini,

- top-3 fattori di rischio più ricorrenti.

"""

rows = []

# Campionamento casuale degli indici del dataset

sample\_idx = np.random.choice(df.index.values, size=n, replace=False)

for i, idx in enumerate(sample\_idx, 1):

print(f"[{i}/{len(sample\_idx)}] Analizzo paziente index={idx}")

paz = df.loc[idx]

res = analyze\_similarity(paz, k\_total=k\_total, k\_pos\_min=k\_pos\_min, save\_json=True, out\_dir=out\_dir)

# Top-3 fattori di rischio

top\_items = list(res["fattori\_rischio\_top"].items())

top1 = f"{top\_items[0][0]}={top\_items[0][1]:.2f}" if len(top\_items)>0 else ""

top2 = f"{top\_items[1][0]}={top\_items[1][1]:.2f}" if len(top\_items)>1 else ""

top3 = f"{top\_items[2][0]}={top\_items[2][1]:.2f}" if len(top\_items)>2 else ""

rows.append({

"patient\_index": idx, # indice reale dal dataset

"perc\_malati\_vicini": res["perc\_malati\_vicini"],

"top1": top1, "top2": top2, "top3": top3

})

# Costruisci DataFrame e salva CSV

df\_out = pd.DataFrame(rows)

out\_csv = out\_dir / "similarity\_batch\_summary.csv"

df\_out.to\_csv(out\_csv, index=False)

print(f"\n CSV riassuntivo salvato: {out\_csv}")

print(f"Report creato con {len(df\_out)} pazienti (campionati casualmente):")

print(df\_out.to\_string(index=False)) # stampa TUTTE le righe

return df\_out

# Esecuzione con n=10 pazienti casuali

df\_batch = analyze\_similarity\_batch\_random(

df=X\_ref, # il DataFrame di riferimento (test reale)

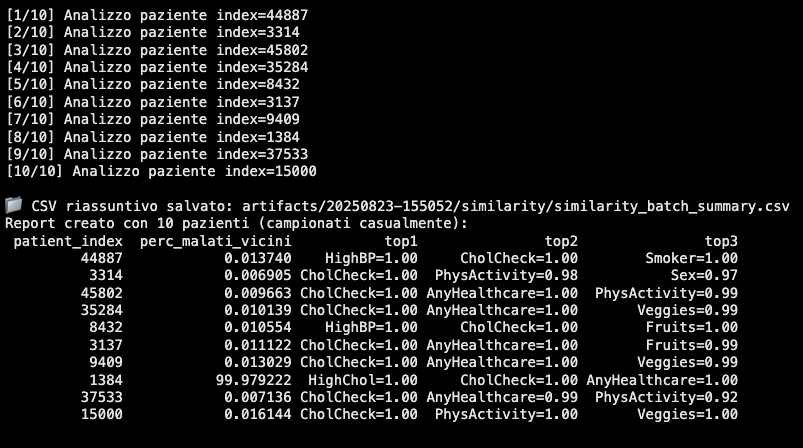
n=10, # numero di pazienti da analizzare

k\_total=200, # numero di vicini totali

k\_pos\_min=100, # min. vicini positivi garantiti

out\_dir=SIM\_OUT

)

******

***10.1) Scarica zip***

import glob

from pathlib import Path

import pandas as pd

# trova l'ultima cartella artifacts/.../similarity/

sim\_dirs = sorted(glob.glob("artifacts/\*/similarity/"))

assert sim\_dirs, "Nessuna cartella similarity trovata sotto artifacts/."

SIM\_DIR = Path(sim\_dirs[-1])

print("Cartella similarity:", SIM\_DIR)

# path del CSV

csv\_path = SIM\_DIR / "similarity\_batch\_summary.csv"

assert csv\_path.exists(), f"CSV non trovato: {csv\_path}"

# mostra anteprima

df = pd.read\_csv(csv\_path)

print("\nPrime 10 righe:")

print(df.head(10).to\_string(index=False))

# avvia download in locale (Colab)

from google.colab import files

files.download(str(csv\_path))

print("\n Download del CSV avviato:", csv\_path)

***10.2) Scarica ZIP***

import os, shutil, glob

from pathlib import Path

# 1) ultima cartella similarity

sim\_dirs = sorted(glob.glob("artifacts/\*/similarity/"))

assert sim\_dirs, "Nessuna cartella similarity trovata."

SIM\_DIR = Path(sim\_dirs[-1])

# 2) CSV riassuntivo

csv\_path = SIM\_DIR / "similarity\_batch\_summary.csv"

assert csv\_path.exists(), f"CSV non trovato: {csv\_path}"

# 3) leggi gli indici reali dal CSV

import pandas as pd

ids = pd.read\_csv(csv\_path)["patient\_index"].astype(int).tolist()

# 4) copia i JSON elencati nel CSV + il CSV in una sottocartella

dst\_dir = SIM\_DIR / "batch\_json\_10"

dst\_dir.mkdir(exist\_ok=True)

copiati = []

for pid in ids:

src = SIM\_DIR / f"similarity\_patient\_{pid}.json"

if src.exists():

shutil.copy2(src, dst\_dir / src.name)

copiati.append(src.name)

else:

print(" Mancante:", src.name)

# copia anche il CSV

shutil.copy2(csv\_path, dst\_dir / csv\_path.name)

print(f"File copiati ({len(copiati)} JSON + CSV):", len(copiati), "+ 1")

# 5) zippa e scarica

zip\_path = SIM\_DIR / "batch\_json\_10.zip"

if zip\_path.exists():

zip\_path.unlink()

shutil.make\_archive(str(zip\_path).replace(".zip",""), "zip", dst\_dir)

from google.colab import files

files.download(str(zip\_path))

print(" ZIP scaricato:", zip\_path)