```
import pandas as pd
from sklearn.preprocessing import OneHotEncoder
from sklearn.neighbors import KNeighborsClassifier, RadiusNeighborsClassif
from sklearn.metrics import accuracy_score, confusion_matrix, ConfusionMat
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.collections import LineCollection

import seaborn as sns
import glob
```

```
# 1️⃣ Caricamento dati (es. monks-1)
train_path = "/content/sample_data/monks-3.train"
test_path = "/content/sample_data/monks-3.test"
cols = ['target', 'a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'id']

train_df = pd.read_csv(train_path,
                       sep=r'\s+', names=cols, usecols=range(7))
test_df  = pd.read_csv(test_path,
                       sep=r'\s+', names=cols, usecols=range(7))

print(train_df.head())
print(test_df.head())
```

Separazione X e y, sia nel Test che nel Training set

```
# 2️⃣ Separiamo X e y
# La separazione training, viene divisa SOLO per fare ENCODER di X_train
#Poi viene rimessa insieme quando viene passato al KNN in "TRAINING"
X_train, y_train = train_df.drop('target', axis=1), train_df['target']
X_test,  y_test  = test_df.drop('target', axis=1), test_df['target']
```

One Hot Encoder

```
# One-Hot Encoding
encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
X_train_enc = encoder.fit_transform(X_train)
X_test_enc  = encoder.transform(X_test)
```

Distanza (Hamming qua...)

```
# === 4️⃣ Loop su Knn classico neighbors k = 1..6 e valutazione ===
k_values = range(1, 7)
accuracies = []

for k in k_values:
    print(f"\n--- KNN con k={k} ---")
```

```
    knn = KNeighborsClassifier(n_neighbors=k, metric='hamming')
    #Qua metto faccio il FITTING dei dati di TRAINING
    knn.fit(X_train_enc, y_train)
    #Qua uso i dati di TEST per fare la stima della classificazione
    y_pred = knn.predict(X_test_enc)
    #Qua calcola la accuratezza tra quelli effettivi e quelli del knn
    acc = accuracy_score(y_test, y_pred)
    accuracies.append(acc)

    print(f"Accuracy: {acc:.3f}")

    # Matrice di confusione
    disp = ConfusionMatrixDisplay.from_estimator(knn, X_test_enc, y_test, cm
    disp.ax_.set_title(f"Matrice di Confusione - k={k}")
    plt.show()
# === 4 Loop su Knnradius ===
radius_values = [0.1, 0.2, 0.3, 0.32,0.33,0.35,0.4,0.43,0.45, 0.5]  # puoi m
accuraciesRadius = []

for r in radius_values:
    print(f"\n--- RadiusNeighborsClassifier con radius={r} ---")

    rknn = RadiusNeighborsClassifier(radius=r, metric='hamming', weights='un
    rknn.fit(X_train_enc, y_train)

    # Attenzione: se nessun vicino entro il radius, può dare errore
    try:
        y_pred = rknn.predict(X_test_enc)
        acc = accuracy_score(y_test, y_pred)
        accuraciesRadius.append(acc)
        print(f"Accuracy: {acc:.3f}")

        # Matrice di confusione
        disp = ConfusionMatrixDisplay.from_estimator(rknn, X_test_enc, y_tes
        disp.ax_.set_title(f"Matrice di Confusione - radius={r}")
        plt.show()
    except ValueError as e:
        print(f"Errore con radius={r}: {e}")
        accuraciesRadius.append(None)
```

```
# Funzione helper per linee colorate con fade
def plot_colored_line_fade(x, y, title, xlabel, ylabel, steps_per_segment=
    # Assicuriamoci che y sia tra 0 e 1
    y = np.clip(y, 0, 1)
    x = np.array(x)
    y = np.array(y)

    # Creiamo punti intermedi per fade
    x_fade = []
    y_fade = []
    for i in range(len(x)-1):
        xs = np.linspace(x[i], x[i+1], steps_per_segment)
        ys = np.linspace(y[i], y[i+1], steps_per_segment)
        x_fade.extend(xs)
        y_fade.extend(ys)
```

```
            y_fade.extend(ys)

        x_fade = np.array(x_fade)
        y_fade = np.array(y_fade)

        # Segmenti per LineCollection
        points = np.array([x_fade, y_fade]).T.reshape(-1,1,2)
        segments = np.concatenate([points[:-1], points[1:]], axis=1)

        # Colormap rosso->giallo->verde
        cmap = mpl.cm.get_cmap('RdYlGn')
        norm = mpl.colors.Normalize(vmin=0, vmax=1)

        lc = LineCollection(segments, cmap=cmap, norm=norm)
        lc.set_array(y_fade)  # colore basato sui valori intermedi
        lc.set_linewidth(3)

        fig, ax = plt.subplots(figsize=(8,4))
        ax.add_collection(lc)
        ax.set_xlim(min(x), max(x))
        ax.set_ylim(0, 1)
        ax.set_title(title)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        ax.grid(True)
        plt.show()

    # === Grafico accuracies vs k con fade ===
    plot_colored_line_fade(list(k_values), accuracies,
                           "Accuracy vs K (MONK-1) con fade",
                           "Numero di vicini (k)", "Accuracy")

    # === Grafico accuracies vs radius con fade ===
    plot_colored_line_fade(radius_values, [a if a is not None else 0 for a in a
                           "Accuracy vs Radius (RadiusNeighborsClassifier) - M
                           "Radius", "Accuracy")
```

```
    from sklearn.metrics import pairwise_distances

    # === 6 MKNN ===
    print("\n=== Modified KNN (MKNN) ===")

    k_mknn = 3  # puoi provare altri valori di k per la validità
    accuracies_mknn = []

    # 1 Calcolo della validità dei punti di training
    # Distanze Hamming tra tutti i punti di train
    dist_train = pairwise_distances(X_train_enc, X_train_enc, metric='hamming')

    validity = []
    for i in range(len(X_train_enc)):
        # escludi se stesso
        idx = np.argsort(dist_train[i])[1:k_mknn+1]
```

```
    # calcola la frazione dei vicini con la stessa classe
    same_class = np.sum(y_train.iloc[idx].values == y_train.iloc[i])
    validity.append(same_class / k_mknn)
validity = np.array(validity)

# 2 Predizione su test set usando validità come peso
for k in range(1, 7):
    y_pred = []
    for xt in X_test_enc:
        # distanza Hamming dal test point a tutti i punti di train
        dists = np.sum(np.abs(X_train_enc - xt), axis=1) / X_train_enc.shape
        # prendi i k vicini più vicini
        nn_idx = np.argsort(dists)[:k]
        # pesi = validità dei vicini / distanza (evitiamo divisione per 0)
        eps = 1e-5
        weights = validity[nn_idx] / (dists[nn_idx] + eps)
        # somma dei pesi per classe
        classes = y_train.iloc[nn_idx].values
        vote = {}
        for c, w in zip(classes, weights):
            vote[c] = vote.get(c, 0) + w
        # scegli la classe con peso massimo
        y_pred.append(max(vote, key=vote.get))
    y_pred = np.array(y_pred)
    acc = accuracy_score(y_test, y_pred)
    accuracies_mknn.append(acc)
    print(f"MKNN Accuracy (k={k}): {acc:.3f}")

# Grafico accuracies MKNN con fade
plot_colored_line_fade(list(range(1,7)), accuracies_mknn,
                       "Accuracy vs K (MKNN) - MONK-1 con fade",
                       "Numero di vicini (k)", "Accuracy")
```

```
from sklearn.metrics import pairwise_distances

# === 7 Weighted KNN ===
print("\n=== Weighted KNN ===")

k_wknn = 3  # numero di vicini
accuracies_wknn = []

for k in range(1, 7):
    y_pred = []
    for xt in X_test_enc:
        # distanza Hamming dal test point a tutti i punti di train
        dists = np.sum(np.abs(X_train_enc - xt), axis=1) / X_train_enc.sha

        # prendi i k vicini più vicini
        nn_idx = np.argsort(dists)[:k]

        # pesi = 1 / distanza (aggiungiamo epsilon per evitare divisione pe
        eps = 1e-5
        weights = 1 / (dists[nn_idx] + eps)

        # somma dei pesi per classe
```

```
            # somma dei pesi per classe
            classes = y_train.iloc[nn_idx].values
            vote = {}
            for c, w in zip(classes, weights):
                vote[c] = vote.get(c, 0) + w

            # scegli la classe con peso massimo
            y_pred.append(max(vote, key=vote.get))

        y_pred = np.array(y_pred)
        acc = accuracy_score(y_test, y_pred)
        accuracies_wknn.append(acc)
        print(f"Weighted KNN Accuracy (k={k}): {acc:.3f}")

    # Grafico accuracies Weighted KNN con fade
    plot_colored_line_fade(list(range(1,7)), accuracies_wknn,
                           "Accuracy vs K (Weighted KNN) - MONK-1 con fade",
                           "Numero di vicini (k)", "Accuracy")
```