```python
import numpy as np
import pandas as pd
from LinearRegression import LinearRegression
from LogisticRegression import LogisticRegression
np.random.seed(123)

# IMPORT DATASET
houses = pd.read_csv('./houses.csv')
# se non ha header
wine = pd.read_csv('../data/wine.csv', header=None)


# SHUFFLING, AVOID GRUP BIAS, frac=1 == 100%, reset_index drop= sostituiamo indici con numerici
houses = houses.sample(frac=1).reset_index(drop=True)


# COMBINATION OF FEATURES ES
houses['average_rating'] = houses[['OverallQual', 'OverallCond']].mean(axis=1)


# REPLACE OF FEATURES , es true/false con 0 e 1
wine['deasese'] = wine['desease'].replace('False', 0)
wine['deasese'] = wine['desease'].replace('True', 1)

# SELECT FEATURES
# opzione 1: per nome di colonne
x = houses[['GrLivArea', 'LotArea', 'GarageArea', 'FullBath']].values
y = houses['SalePrice'].values

# opzione 2A: Trasformo in np.array e faccio slicing
y = houses.values[:, -1]
x = houses.values[:, 0:10]

# opzione 2B: iloc
x = houses.drop(houses.iloc[:, 0:10], axis=1).values
x = houses.drop(houses.iloc[:, [1, 2, 3]], axis=1).values


# HOLD OUT SPLITTING = 80% train , 20%test
train_index = round(len(x) * 0.8)
X_train = x[:train_index]
y_train = y[:train_index]
X_test = x[train_index:]
y_test = y[train_index:]

# HOLD OUT STRATIFICATO, supponiamo che feature da predire sia a colonna 0
data = houses.values
target1 = []
target2 = []

for i in range(len(data)):
    if data[i, 0] == 1:
        target1.append(data[i])
    else:
        target2.append(data[i])

target1 = np.array(target1)
target2 = np.array(target2)
```

```python
train_index_1 = round(len(target1) * 0.8)
train_index_2 = round(len(target2) * 0.8)

data_train = np.concatenate((target1[:train_index_1], target2[:train_index_2]), axis=0)
data_test = np.concatenate((target1[train_index_1:], target2[train_index_2:]), axis=0)

np.random.shuffle(data_test)
np.random.shuffle(data_train)

x_train = data_train[:, 1:]
y_train = data_train[:, 0]
x_test = data_test[:, 1:]
y_test = data_test[:, 0]


# ZSCORE NORMALIZATION, axis = 0 (verticale) | , axis=1 orrizzontale ->
mean = X_train.mean(axis=0)
std = X_train.std(axis=0)
X_train = (X_train - mean) / std
X_test = (X_test - mean) / std

# MIN/MAX NORMALIZATION es tra 0 e 100
a = 0
b = 100
x_train = ((x_train - min(x_train)) / (max(x_train) - min(x_train))) * (b - a) + a


# COLONNA BIAS PE RI THETA0 ( valore che non viene moltiplicato per gli xi)
# x.shape == (righe, colonne) ==> x.shape[0] = righe, x.shape[1] = colonne
# np.c_ .. aggiungiamo matrice di dimensione righeX x 1, composta da tutti 1 davanti a X_train
# np.ones([4,2]) = mat 1 4 righe 2 colonne
X_train = np.c_[np.ones(X_train.shape[0]), X_train]


# SPLIT TRAINING SET into training and validation ( 70% train , 30% val)
validation_index = round(train_index * 0.7)
X_validation = X_train[validation_index:]
y_validation = y_train[validation_index:]

X_train = X_train[:validation_index]
y_train = y_train[:validation_index]


# K FOLD PER SCEGLIERE BEST FEATURES
k = 4
fold = round(len(X_train) / k)

features_list = []
for feature in range(X_train.shape[1]):
    feature_GER = 0
    for j in range(0, k):
        if j == k - 1:
            x_validation = X_train[k*j:, feature]
            y_validation = y_train[k*j:]
            x_train = X_train[0:k*j, feature]
            y_train1 = y_train[0:k*j]

        else:
```

```
            x_validation = X_train[k*j:k*(j+1), feature]
            y_validation = y_train[k*j:k*(j+1)]
            x_train = np.concatenate((X_train[k * (j + 1):, feature], X_train[0:k * j, feature]), axis=0)
            y_train1 = np.concatenate((y_train[0:k * j], y_train[k * (j + 1):]), axis=0)

        # x_train_list = X_train[k*(j+1):, feature] + X_train[0:k*j, feature]
        # x_train = np.array(x_train_list)


        # bias column
        x_validation = np.c_[np.ones(x_validation.shape[0]), x_validation]
        x_train = np.c_[np.ones(x_train.shape[0]), x_train]

        # y_train_list = y_train[0:k*j] + y_train[k*(j+1):]
        # y_train1 = np.arrat(y_train)


        regressor = LinearRegression(nfeatures=2, steps=1000, a=0.05, lmd=2)
        _, cost_list, _ = regressor.fit_reg(x_train, y_train1, x_validation, y_validation)
        feature_GER += cost_list[-1]
    feature_GER = feature_GER / k
    features_list.append((feature_GER, feature))

# best for features
features_list.sort(key=lambda x: x[0])
# retrain on the best 4 features
column = []
n_features = 4
for j in [x[1] for x in features_list]:
    if len(column) < n_features:
        column.append(j)

X_train = X_train[:, column]
X_train = np.c_[np.ones(X_train.shape[0]), X_train]
X_validation = X_train[validation_index:, :]
X_train = X_train[0:validation_index, :]

y_validation = y_train[validation_index:]
y_train = y_train[0:validation_index]


# REGRESSOR E FIT DATI
linear = LinearRegression(nfeatures=X_train.shape[1], steps=1000, a=0.05, lmd=2)
cost_history, cost_history_val, theta_history = linear.fit(X_train, y_train, X_validation, y_validation)


# CLASSIFICAZIONE MULTICLASS
y = wine[['fruity', 'chocholate', 'caramel']].values
# solite robe, normalizzaz  ecc
lr, pred = [], [] # lista di regressori e valori predetti
for i in range(y.shape[1]):
    lr[i] = LogisticRegression(learning_rate=X_train.shape[1], n_steps=1000)
    predicted = lr[i].predict(x_train, 0.6)
    pred[i] = predicted
pred_fruity = pred[0]
pred_choco = pred[1]
pred_caramel = pred[2]

```

```python
import numpy as np

np.random.seed(123)

''' dal main la richimiamo come:
    nn = NeuralNet(layers=[X_train.shape[1], 25, 1], learning_rate=0.5, iterations=1000, lmd=0)
    nn.fit(X_train, y_label_train)
    y_hat = nn.predict(X_test)
    nn.rmse(y_label_test, y_hat)
    '''

class NeuralNetworkClass:
    def __init__(self, layers, learning_rate, iterations, lmd):
        self.layers = layers
        self.learning_rate = learning_rate
        self.n_iterations = iterations
        self.lmd = lmd
        self.w = {}
        self.b = {}
        self.loss = []
        self.X = None
        self.y = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def sigmoid_derivative(self, a):
        return a * (1 - a)

    def init_weights(self):
        L = len(self.layers)
        np.random.seed(42)

        for l in range(1, L):
            self.w[l] = np.random.randn(self.layers[l], self.layers[l - 1])
            self.b[l] = np.random.randn(self.layers[l], 1)

    def forward_propagation(self):
        L = len(self.layers)
        Z = {}
        A = {0: self.X.T}

        for l in range(1, L):
            Z[l] = np.dot(self.w[l], A[l - 1]) + self.b[l]
            A[l] = self.sigmoid(Z[l])

        return Z, A

    def back_propagation(self, Z, A):
        L = len(self.layers)
        m = len(self.y)

        dW = {}
        dB = {}

        for l in range(L - 1, 0, -1):
            if l == L - 1:
                # -y/a + (1-y)/(1-a) * a(1-a) si semplifica in a - y
```

```python
                # -y +ya a -ay / a(1-a)
                dZ = A[l] - self.y.T
            else:
                dA = np.dot(self.w[l + 1].T, dZ)
                dZ = np.multiply(dA, self.sigmoid_derivative(A[l]))

            dW[l] = 1 / m * np.dot(dZ, A[l - 1].T) + self.lmd * self.w[l]
            dB[l] = 1 / m * np.sum(dZ, axis=1, keepdims=True)

        return dW, dB

    def update_params(self, dW, dB):
        L = len(self.layers)

        for l in range(1, L):
            self.w[l] -= self.learning_rate * dW[l]
            self.b[l] -= self.learning_rate * dB[l]

    def compute_cost(self, A):
        m = len(self.y)
        L = len(self.layers)

        preds = A[len(A) - 1]

        cost = -np.average(self.y.T * np.log(preds) + (1 - self.y.T) * np.log(1 - preds))
        reg_sum = 0
        for l in range(1, len(self.layers)):
            reg_sum += (np.sum(np.square(self.w[l])))
        L2_regularization_cost = reg_sum * (self.lmd / (2 * m))
        return cost + L2_regularization_cost

    def fit(self, X, y):
        self.X = X
        self.y = y
        self.init_weights()

        for i in range(self.n_iterations):
            Z, A = self.forward_propagation()
            dW, dB = self.back_propagation(Z, A)
            self.update_params(dW, dB)
            cost = self.compute_cost(A)
            self.loss.append(cost)

    def predict(self, X, t=0.5):
        self.X = X
        _, A = self.forward_propagation()
        preds = A[len(A) - 1]
        return preds >= t


    def confusion_matrix(self, ytest,  predicted):
        tp, fp, tn, fn = 0, 0, 0, 0
        i = 0
        for pred in predicted:
            if pred == ytest[i]:
                tp += 1
            elif pred == 1 and ytest[i] == 0:
                fp += 1
```

```python
            elif pred == 0 and ytest[i] == 0:
                tn += 1
            elif pred == 0 and ytest[i] == 1:
                fn += 1
            i += 1
        # sensityvity = recall = tp / tot positivi in y
        recall = tp / (tp + fn)

        # accuracy true / tutto
        accuracy = (tp + tn) / (tp + tn + fp + fn)

        # precision = positivi azzeccati / tot positivi predetti
        precision = tp / (tp + fp)

        # specificity = tn rate = negativi azzeccati / negativi reali
        specificity = tn / (tn + fp)

        # error rate = falsi / tutto
        errorrate = (fp + fn) / (tp + fp + tn + fn)

        # fmeasure = 2 * (precision * recall ) / (precision + recall)
        fmeasure = 2 * (precision * recall) / (precision + recall)

        return recall, accuracy, precision, specificity, errorrate, fmeasure
```

```
import numpy as np

np.random.seed(123)

''' dal main la richimiamo come:
    nn = NeuralNet(layers=[X_train.shape[1], 25, 1], learning_rate=0.5, iterations=1000, lmd=0)
    nn.fit(X_train, y_label_train)
    y_hat = nn.predict(X_test)
    nn.rmse(y_label_test, y_hat)
    '''


class NeuralLinear:

    def __init__(self, layers=[13, 8, 2], learning_rate=0.05, steps= 1000, lmd=1):
        self.layers = layers
        self.learning_rate = learning_rate
        self.steps = steps
        self.lmd = lmd
        self.w = {}
        self.b = {}
        self.Y = None
        self.X = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def sigmoid_derivative(self, a):
        return a * (1 - a)

    def init_weights(self):
        L = len(self.layers)
        for l in range(1, L):
            self.w[l] = np.random.randn(self.layers[l], self.layers[l-1])
            self.b[l] = np.random.randn(self.layers[l], 1)

    def forward_propagation(self):
        L = len(self.layers)
        A = {0: self.X.T}
        Z = {}

        for l in range(1, L):
            Z[l] = np.dot(self.w[l], A[l-1]) + self.b[l]
            if l == L - 1:
                A[l] = Z[l]
            else:
                A[l] = self.sigmoid(Z[l])
        return Z, A

    def back_propagation(self, Z, A):
        L = len(self.layers)
        dW = {}
        dB = {}
        m = len(self.X)
        for l in range(L-1, 0, -1):
            if l == L - 1:
                dA = A[l] - self.Y.T
                dZ = dA
```

```
            else:
                dA = np.dot(self.w[l+1].T, dZ)
                dZ = np.multiply(dA, self.sigmoid_derivative(A[l]))
            dW[l] = 1/ m * np.dot(dZ, A[l-1].T) + (self.lmd * self.w[l])
            dB[l] = 1 / m * np.sum(dZ, axis=1, keepdims=True)
        return dW, dB

    def update_param(self, dW, dB):
        L = len(self.layers)
        for l in range(1, L):
            self.w[l] -= self.learning_rate * dW[l]
            self.b[l] -= self.learning_rate * dB[l]

    def cost(self, A):
        m = len(self.Y.T)
        L = len(self.layers)
        cost = (1 / (2 * m)) * np.sum(np.square(A[L-1] - self.Y.T))
        reg = 0
        for l in range(1, L):
            reg += (np.sum(np.square(self.w[l])))
        reg_cost = reg * ( self.lmd / (2 * m))
        return cost + reg_cost

    def predict(self, X):
        self.X = X
        _, A = self.forward_propagation()
        L = len(self.layers)
        prediction = A[L-1]
        return prediction

    def fit(self, x, y):
        self.X = x
        self.Y = y
        cost = []
        self.init_weights()
        for step in range(self.steps):
            Z, A = self.forward_propagation()
            dW, dB = self.back_propagation(Z, A)
            self.update_param(dW, dB)
            costo = self.cost(A)
            cost.append(costo)

    def rmse2(self, pred, y):
        n = len(y)
        square = np.sqrt(np.average((pred - y) ** 2))

        return np.sqrt(square)

    def mae(self, pred, y):
        return np.average(np.abs(pred - y))

    def mse(self, pred, y):
        square = (pred - y) ** 2
        return np.average(square)

    def rmse(self, pred, y):
        return np.sqrt(self.mse(pred, y))

```

```
    def mpe(self, pred, y):
        err = (pred - y)/ y
        return np.average(err)

    def mape(self, pred, y):
        err = np.abs((pred - y)/ y)
        return np.average(err)

    def r2(self, pred, y ):
        a = np.sum((pred - y) ** 2)
        b = np.sum((y - y.mean()) ** 2)
        return 1 - (a / b)
```

```
import numpy as np

np.random.seed(123)

class LinearRegression:
    def __init__(self, nfeatures, steps=1000, a=0.05, lmd=2):
        self.steps = steps
        self.a = a
        self.lmd = lmd
        self.lmd_ = np.full(nfeatures, lmd)
        self.lmd_[0] = 0
        self.thetas = np.random.randn(nfeatures)

    def fit_no_reg(self, X, y, X_val, y_val):

        m = len(X)
        cost_history = np.zeros(self.n_steps)
        cost_history_val = np.zeros(self.n_steps)
        theta_history = np.zeros((self.n_steps, self.theta.shape[0]))

        for step in range(0, self.n_steps):
            preds = np.dot(X, self.theta)
            preds_val = np.dot(X_val, self.theta)

            error = preds - y
            error_val = preds_val - y_val

            self.theta = self.theta - (self.learning_rate * (1/m) * np.dot(X.T, error))
            theta_history[step, :] = self.theta.T
            cost_history[step] = 1/(2*m) * np.dot(error.T, error)
            cost_history_val[step] = 1 / (2 * m) * np.dot(error_val.T, error_val)

        return cost_history, cost_history_val, theta_history

    def fit(self, x, xva, y, yva):
        m = len(x)
        cost_history = np.zeros(self.steps)
        cost_history_va = np.zeros(self.steps)
        theta_history = np.zeros((self.steps, self.thetas.shape[0]))

        for step in range(self.steps):
            pred = np.dot(x, self.thetas)
            pred_va = np.dot(xva, self.thetas)
            error = pred - y
            err_va = pred_va - yva

            self.thetas = self.thetas - ((self.a / m) * (np.dot(x.T, error) + (self.thetas * self.lmd_)))
            theta_history[step] = self.thetas
            cost_history_va = (1 / (2 * m)) * (np.dot(err_va.T, err_va) +
                                               (self.lmd * np.dot(self.thetas.T, self.thetas)))
            cost_history[step] = (1 / (2 * m)) * (np.dot(error.T, error) +
                                                  (self.lmd * np.dot(self.thetas.T[1:], self.thetas[1:])))

        return cost_history, cost_history_va, theta_history

    def fit_stochastic(self, x, y, xva, yva):
        cost_history = np.zeros(self.steps)
        cost_history_va = np.zeros(self.steps)
```

```
        thetas_history = np.zeros(self.steps, (self.thetas.shape[0]))
        m = len(x)

        for step in range(self.steps):
            pred_va = np.dot(xva, self.thetas)
            error_va = pred_va - yva
            cost = 0
            for i in range(m):
                x_i = x[i, :]
                y_i = y[i]
                pred = np.dot(x_i, self.thetas)
                error = pred - y_i
                self.thetas = self.thetas - self.a * ( np.dot(x_i.T, error) + np.dot(self.lmd_.T, self.thetas))
                cost += 0.5 * (np.dot(error.T, error) + (self.lmd * np.dot(self.thetas.T, self.thetas)))

            cost_history_va[step] = (1 / (2 * m)) * (np.dot(error_va.T, error_va) + self.lmd * np.dot(self.thetas.T[1:], self.thetas[1:]))
            cost_history[step] = (1 / m) * cost
            thetas_history[step] = self.thetas
        return cost_history, cost_history_va, thetas_history

    def fit_minitbatch(self, x, y, xval, yval, batch_size = 100):
        cost_history = np.zeros(self.steps)
        cost_history_va = np.zeros(self.steps)
        thetas_history = np.zeros(self.steps, (self.thetas.shape[0]))
        m = len(x)

        for step in range(self.steps):
            pred_va = np.dot(xval, self.thetas)
            error_va = pred_va - yval
            cost = 0
            for i in range(0, m, batch_size):
                x_i = x[i: i+batch_size]
                y_i = y[i: i+batch_size]
                pred = np.dot(x_i, self.thetas)
                error = pred - y_i
                self.thetas = self.thetas - self.a * (np.dot(x_i.T, error) + np.dot(self.lmd_.T, self.thetas))
                cost += 0.5 * (np.dot(error.T, error) + (self.lmd * np.dot(self.thetas.T, self.thetas)))
            cost_history_va[step] = (1 / (2 * m)) * (
                    np.dot(error_va.T, error_va) + self.lmd * np.dot(self.thetas.T[1:], self.thetas[1:]))
            cost_history[step] = (1 / m) * cost
            thetas_history[step] = self.thetas
        return cost_history, cost_history_va, thetas_history

    def predict(self, x):
        x = np.c_[np.ones(x.shape[0]), x]
        return np.dot(x, self.thetas)

    def curve(self, x, y, xva, yva):
        m = len(x)
        cost_history = np.zeros(m)
        cost_history_va = np.zeros(m)

        for i in range(m):
            x_i = x[:i+1]
            y_i = y[:i+1]
            c_h, cv, _ = self.fit(x, y, xva, yva)
            cost_history[i] = c_h[-1]
```

```
            cost_history_va[i] = cv[-1]

    def metrics(self, x, y):
        prev = self.predict(x)
        mae = self.mae(prev, y)
        mse = self.mse(prev, y)
        mpe = self.mpe(prev, y)
        mape = self.mape(prev, y)
        r2 = self.r2(prev, y)
        print("MAE: {}  MSE: {}  MPE: {}  mape: {}  R2: {}".format(mae, mse, mpe, mape, r2))


    def mae(self, pred, y):
        return np.average(np.abs(pred - y))

    def mse(self, pred, y):
        square = (pred - y) ** 2
        return np.average(square)

    def rmse(self, pred, y):
        return np.sqrt(self.mse(pred, y))

    def mpe(self, pred, y):
        err = (pred - y)/ y
        return np.average(err)

    def mape(self, pred, y):
        err = np.abs((pred - y)/ y)
        return np.average(err)

    def r2(self, pred, y ):
        a = np.sum((pred - y) ** 2)
        b = np.sum((y - y.mean()) ** 2)
        return 1 - (a / b)
```

```
import numpy as np


np.random.seed(123)


class LogisticRegression:

    def __init__(self, learning_rate=1e-2, n_steps=2000, n_features=1, lmd=1):

        self.learning_rate = learning_rate
        self.n_steps = n_steps
        self.theta = np.random.rand(n_features)
        self.lmd = lmd
        self.lmd_ = np.full((n_features,), lmd)
        self.lmd_[0] = 0

    def _sigmoid(self, z):

        return 1 / (1 + np.exp(-z))

    def fit(self, X, y, X_val, y_val):

        m = len(X)
        cost_history_train = np.zeros(self.n_steps)
        cost_history_val = np.zeros(self.n_steps)
        theta_history = np.zeros((self.n_steps, self.theta.shape[0]))

        for step in range(0, self.n_steps):
            preds = self._sigmoid(np.dot(X, self.theta))
            preds_val = self._sigmoid(np.dot(X_val, self.theta))

            error = preds - y

            self.theta = self.theta - (self.learning_rate * (1 / m) * (np.dot(X.T, error) + (self.theta.T*self.lmd_
)))
            theta_history[step, :] = self.theta.T

            cost_history_train[step] = -(1/m) * (np.dot(y.T, np.log(preds)) + np.dot((1-y.T), np.log(1-preds)))
            cost_history_val[step] = -(1/m) * (np.dot(y_val.T, np.log(preds_val)) + np.dot((1-y_val.T),
                                                                          np.log(1-preds_val)))

        return cost_history_train, cost_history_val, theta_history

    def fit_reg(self, X, y, X_val, y_val):

        m = len(X)
        cost_history_train = np.zeros(self.n_steps)
        cost_history_val = np.zeros(self.n_steps)
        theta_history = np.zeros((self.n_steps, self.theta.shape[0]))

        for step in range(0, self.n_steps):
            preds = self._sigmoid(np.dot(X, self.theta))
            preds_val = self._sigmoid(np.dot(X_val, self.theta))

            error = preds - y

            self.theta = self.theta - (self.learning_rate * (1 / m) * (np.dot(X.T, error) + (self.theta.T*self.lmd_
```

```
)))
            theta_history[step, :] = self.theta.T

            loss = -(1/m) * (np.dot(y.T, np.log(preds)) + np.dot((1-y.T), np.log(1-preds)))
            loss_validation = -(1 / m) * (
                        np.dot(y_val.T, np.log(preds_val)) + np.dot((1 - y_val.T), np.log(1 - preds_val)))
            reg = (self.lmd / (2*m)) * np.dot(self.theta.T[1:], self.theta[1:])



            cost_history_train[step] = loss + reg
            cost_history_val[step] = loss_validation + reg

        return cost_history_train, cost_history_val, theta_history

    def _predict_prob(self, X):

        return self._sigmoid(np.dot(X, self.theta))

    def predict(self, X, threshold):
        """
        perform a complete prediction about X samples
        :param X: test sample with shape (m, n_features)
        :param threshold: threshold value to disambiguate positive or negative sample
        :return: prediction wrt X sample. The shape of return array is (m,)
        """
        Xpred = np.c_[np.ones(X.shape[0]), X]
        return self._predict_prob(Xpred) >= threshold

    def confusion_matrix(self, xtest, treshold, ytest):
        tp, fp, tn, fn = 0, 0, 0, 0
        i = 0
        predicted = self.predict(xtest, treshold)
        for pred in predicted:
            if pred == ytest[i]:
                tp += 1
            elif pred == 1 and ytest[i] == 0:
                fp += 1
            elif pred == 0 and ytest[i] == 0:
                tn += 1
            elif pred == 0 and ytest[i] == 1:
                fn += 1
            i += 1
        # sensityvity = recall = tp / tot positivi in y
        recall = tp / (tp + fn)

        # accuracy true / tutto
        accuracy = (tp + tn) / (tp + tn + fp + fn)

        # precision = positivi azzeccati / tot positivi predetti
        precision = tp / (tp + fp)

        # specificity = tn rate = negativi azzeccati / negativi reali
        specificity = tn / (tn + fp)

        # error rate = falsi / tutto
        errorrate = (fp + fn) / (tp + fp + tn + fn)

```

```
    # fmeasure = 2 * (precision * recall ) / (precision + recall)
    fmeasure = 2 * (precision * recall) / (precision + recall)
    return recall, accuracy, precision, specificity, errorrate, fmeasure

```