

```

1 import numpy as np
2 import pandas as pd
3 from LinearRegression import LinearRegression
4 from LogisticRegression import LogisticRegression
5 np.random.seed(123)
6
7 # IMPORT DATASET
8 houses = pd.read_csv('./houses.csv')
9 # se non ha header
10 wine = pd.read_csv('../data/wine.csv', header=None)
11
12
13 # SHUFFLING, AVOID GRUP BIAS, frac=1 == 100%, reset_index drop= sostituiamo indici con numerici
14 houses = houses.sample(frac=1).reset_index(drop=True)
15
16
17 # COMBINATION OF FEATURES ES
18 houses['average_rating'] = houses[['OverallQual', 'OverallCond']].mean(axis=1)
19
20
21 # REPLACE OF FEATURES , es true/false con 0 e 1
22 wine['deasese'] = wine['desease'].replace('False', 0)
23 wine['deasese'] = wine['desease'].replace('True', 1)
24
25 # SELECT FEATURES
26 # opzione 1: per nome di colonne
27 x = houses[['GrLivArea', 'LotArea', 'GarageArea', 'FullBath']].values
28 y = houses['SalePrice'].values
29
30 # opzione 2A: Trasformo in np.array e faccio slicing
31 y = houses.values[:, -1]
32 x = houses.values[:, 0:10]
33
34 # opzione 2B: iloc
35 x = houses.drop(houses.iloc[:, 0:10], axis=1).values
36 x = houses.drop(houses.iloc[:, [1, 2, 3]], axis=1).values
37
38
39 # HOLD OUT SPLITTING = 80% train , 20%test
40 train_index = round(len(x) * 0.8)
41 X_train = x[:train_index]
42 y_train = y[:train_index]
43 X_test = x[train_index:]
44 y_test = y[train_index:]
45
46 # HOLD OUT STRATIFICATO, supponiamo che feature da predire sia a colonna 0
47 data = houses.values
48 target1 = []
49 target2 = []
50
51 for i in range(len(data)):
52     if data[i, 0] == 1:
53         target1.append(data[i])
54     else:
55         target2.append(data[i])
56
57 target1 = np.array(target1)
58 target2 = np.array(target2)

```

```

59 train_index_1 = round(len(target1) * 0.8)
60 train_index_2 = round(len(target2) * 0.8)
61
62 data_train = np.concatenate((target1[:train_index_1], target2[:train_index_2]), axis=0)
63 data_test = np.concatenate((target1[train_index_1:], target2[train_index_2:]), axis=0)
64
65 np.random.shuffle(data_test)
66 np.random.shuffle(data_train)
67
68 x_train = data_train[:, 1:]
69 y_train = data_train[:, 0]
70 x_test = data_test[:, 1:]
71 y_test = data_test[:, 0]
72
73
74 # ZSCORE NORMALIZATION, axis = 0 (verticale) | , axis=1 orizzontale ->
75 mean = X_train.mean(axis=0)
76 std = X_train.std(axis=0)
77 X_train = (X_train - mean) / std
78 X_test = (X_test - mean) / std
79
80 # MIN/MAX NORMALIZATION es tra 0 e 100
81 a = 0
82 b = 100
83 x_train = ((x_train - min(x_train)) / (max(x_train) - min(x_train))) * (b - a) + a
84
85
86 # COLONNA BIAS PE RI THETA0 ( valore che non viene moltiplicato per gli xi)
87 # x.shape == (righe, colonne) ==> x.shape[0] = righe, x.shape[1] = colonne
88 # np.c_ .. aggiungiamo matrice di dimensione righeX 1, composta da tutti 1 davanti a X_train
89 # np.ones([4,2]) = mat 1 4 righe 2 colonne
90 X_train = np.c_[np.ones(X_train.shape[0]), X_train]
91
92
93 # SPLIT TRAINING SET into training and validation ( 70% train , 30% val)
94 validation_index = round(train_index * 0.7)
95 X_validation = X_train[validation_index:]
96 y_validation = y_train[validation_index:]
97
98 X_train = X_train[:validation_index]
99 y_train = y_train[:validation_index]
100
101
102 # K FOLD PER SCEGLIERE BEST FEATURES
103 k = 4
104 fold = round(len(X_train) / k)
105
106 features_list = []
107 for feature in range(X_train.shape[1]):
108     feature_GER = 0
109     for j in range(0, k):
110         if j == k - 1:
111             x_validation = X_train[k*j:, feature]
112             y_validation = y_train[k*j:]
113             x_train = X_train[0:k*j, feature]
114             y_train1 = y_train[0:k*j]
115
116         else:

```

```

117     x_validation = X_train[k*j:k*(j+1), feature]
118     y_validation = y_train[k*j:k*(j+1)]
119     x_train = np.concatenate((X_train[k * (j + 1):, feature], X_train[0:k * j, feature]), axis=0)
120     y_train1 = np.concatenate((y_train[0:k * j], y_train[k * (j + 1):]), axis=0)
121
122     # x_train_list = X_train[k*(j+1):, feature] + X_train[0:k*j, feature]
123     # x_train = np.array(x_train_list)
124
125
126     # bias column
127     x_validation = np.c_[np.ones(x_validation.shape[0]), x_validation]
128     x_train = np.c_[np.ones(x_train.shape[0]), x_train]
129
130     # y_train_list = y_train[0:k*j] + y_train[k*(j+1):]
131     # y_train1 = np.array(y_train)
132
133
134     regressor = LinearRegression(nfeatures=2, steps=1000, a=0.05, lmd=2)
135     _, cost_list, _ = regressor.fit_reg(x_train, y_train1, x_validation, y_validation)
136     feature_GER += cost_list[-1]
137     feature_GER = feature_GER / k
138     features_list.append((feature_GER, feature))
139
140 # best for features
141 features_list.sort(key=lambda x: x[0])
142 # retrain on the best 4 features
143 column = []
144 n_features = 4
145 for j in [x[1] for x in features_list]:
146     if len(column) < n_features:
147         column.append(j)
148
149 X_train = X_train[:, column]
150 X_train = np.c_[np.ones(X_train.shape[0]), X_train]
151 X_validation = X_train[validation_index:, :]
152 X_train = X_train[0:validation_index, :]
153
154 y_validation = y_train[validation_index:]
155 y_train = y_train[0:validation_index]
156
157
158 # REGRESSOR E FIT DATI
159 linear = LinearRegression(nfeatures=X_train.shape[1], steps=1000, a=0.05, lmd=2)
160 cost_history, cost_history_val, theta_history = linear.fit(X_train, y_train, X_validation, y_validation)
161
162
163 # CLASSIFICAZIONE MULTICLASS
164 y = wine[['fruity', 'chocolate', 'caramel']].values
165 # solite robe, normalizzaz ecc
166 lr, pred = [], [] # lista di regressori e valori predetti
167 for i in range(y.shape[1]):
168     lr[i] = LogisticRegression(learning_rate=X_train.shape[1], n_steps=1000)
169     predicted = lr[i].predict(x_train, 0.6)
170     pred[i] = predicted
171     pred_fruity = pred[0]
172     pred_choco = pred[1]
173     pred_caramel = pred[2]
174

```

175

176

177

```

1 import numpy as np
2
3 np.random.seed(123)
4
5 """ dal main la richiamiamo come:
6     nn = NeuralNet(layers=[X_train.shape[1], 25, 1], learning_rate=0.5, iterations=1000, lmd=0)
7     nn.fit(X_train, y_label_train)
8     y_hat = nn.predict(X_test)
9     nn.rmse(y_label_test, y_hat)
10    """
11
12 class NeuralNetworkClass:
13     def __init__(self, layers, learning_rate, iterations, lmd):
14         self.layers = layers
15         self.learning_rate = learning_rate
16         self.n_iterations = iterations
17         self.lmd = lmd
18         self.w = {}
19         self.b = {}
20         self.loss = []
21         self.X = None
22         self.y = None
23
24     def sigmoid(self, z):
25         return 1 / (1 + np.exp(-z))
26
27     def sigmoid_derivative(self, a):
28         return a * (1 - a)
29
30     def init_weights(self):
31         L = len(self.layers)
32         np.random.seed(42)
33
34         for l in range(1, L):
35             self.w[l] = np.random.randn(self.layers[l], self.layers[l - 1])
36             self.b[l] = np.random.randn(self.layers[l], 1)
37
38     def forward_propagation(self):
39         L = len(self.layers)
40         Z = {}
41         A = {0: self.X.T}
42
43         for l in range(1, L):
44             Z[l] = np.dot(self.w[l], A[l - 1]) + self.b[l]
45             A[l] = self.sigmoid(Z[l])
46
47         return Z, A
48
49     def back_propagation(self, Z, A):
50         L = len(self.layers)
51         m = len(self.y)
52
53         dW = {}
54         dB = {}
55
56         for l in range(L - 1, 0, -1):
57             if l == L - 1:
58                 # -y/a + (1-y)/(1-a) * a(1-a) si semplifica in a - y

```

```

59      # -y +ya a -ay / a(1-a)
60      dZ = A[l] - self.y.T
61      else:
62          dA = np.dot(self.w[l + 1].T, dZ)
63          dZ = np.multiply(dA, self.sigmoid_derivative(A[l]))
64
65      dW[l] = 1 / m * np.dot(dZ, A[l - 1].T) + self.lmd * self.w[l]
66      dB[l] = 1 / m * np.sum(dZ, axis=1, keepdims=True)
67
68  return dW, dB
69
70 def update_params(self, dW, dB):
71     L = len(self.layers)
72
73     for l in range(1, L):
74         self.w[l] -= self.learning_rate * dW[l]
75         self.b[l] -= self.learning_rate * dB[l]
76
77     def compute_cost(self, A):
78         m = len(self.y)
79         L = len(self.layers)
80
81         preds = A[len(A) - 1]
82
83         cost = -np.average(self.y.T * np.log(preds) + (1 - self.y.T) * np.log(1 - preds))
84         reg_sum = 0
85         for l in range(1, len(self.layers)):
86             reg_sum += (np.sum(np.square(self.w[l]))))
87         L2_regularization_cost = reg_sum * (self.lmd / (2 * m))
88         return cost + L2_regularization_cost
89
90     def fit(self, X, y):
91         self.X = X
92         self.y = y
93         self.init_weights()
94
95         for i in range(self.n_iterations):
96             Z, A = self.forward_propagation()
97             dW, dB = self.back_propagation(Z, A)
98             self.update_params(dW, dB)
99             cost = self.compute_cost(A)
100            self.loss.append(cost)
101
102    def predict(self, X, t=0.5):
103        self.X = X
104        _, A = self.forward_propagation()
105        preds = A[len(A) - 1]
106        return preds >= t
107
108
109    def confusion_matrix(self, ytest, predicted):
110        tp, fp, tn, fn = 0, 0, 0, 0
111        i = 0
112        for pred in predicted:
113            if pred == ytest[i]:
114                tp += 1
115            elif pred == 1 and ytest[i] == 0:
116                fp += 1

```

```
117     elif pred == 0 and ytest[i] == 0:  
118         tn += 1  
119     elif pred == 0 and ytest[i] == 1:  
120         fn += 1  
121         i += 1  
122     # sensitivity = recall = tp / tot positivi in y  
123     recall = tp / (tp + fn)  
124  
125     # accuracy true / tutto  
126     accuracy = (tp + tn) / (tp + tn + fp + fn)  
127  
128     # precision = positivi azzeccati / tot positivi predetti  
129     precision = tp / (tp + fp)  
130  
131     # specificity = tn rate = negativi azzeccati / negativi reali  
132     specificity = tn / (tn + fp)  
133  
134     # error rate = falsi / tutto  
135     errorrate = (fp + fn) / (tp + fp + tn + fn)  
136  
137     # fmeasure = 2 * (precision * recall) / (precision + recall)  
138     fmeasure = 2 * (precision * recall) / (precision + recall)  
139  
140     return recall, accuracy, precision, specificity, errorrate, fmeasure
```

```

1 import numpy as np
2
3 np.random.seed(123)
4
5 """ dal main la richiamiamo come:
6     nn = NeuralNet(layers=[X_train.shape[1], 25, 1], learning_rate=0.5, iterations=1000, lmd=0)
7     nn.fit(X_train, y_label_train)
8     y_hat = nn.predict(X_test)
9     nn.rmse(y_label_test, y_hat)
10    """
11
12
13 class NeuralLinear:
14
15     def __init__(self, layers=[13, 8, 2], learning_rate=0.05, steps= 1000, lmd=1):
16         self.layers = layers
17         self.learning_rate = learning_rate
18         self.steps = steps
19         self.lmd = lmd
20         self.w = {}
21         self.b = {}
22         self.Y = None
23         self.X = None
24
25     def sigmoid(self, z):
26         return 1 / (1 + np.exp(-z))
27
28     def sigmoid_derivative(self, a):
29         return a * (1 - a)
30
31     def init_weights(self):
32         L = len(self.layers)
33         for l in range(1, L):
34             self.w[l] = np.random.randn(self.layers[l], self.layers[l-1])
35             self.b[l] = np.random.randn(self.layers[l], 1)
36
37     def forward_propagation(self):
38         L = len(self.layers)
39         A = {0: self.X.T}
40         Z = {}
41
42         for l in range(1, L):
43             Z[l] = np.dot(self.w[l], A[l-1]) + self.b[l]
44             if l == L - 1:
45                 A[l] = Z[l]
46             else:
47                 A[l] = self.sigmoid(Z[l])
48
49         return Z, A
50
51     def back_propagation(self, Z, A):
52         L = len(self.layers)
53         dW = {}
54         dB = {}
55         m = len(self.X)
56         for l in range(L-1, 0, -1):
57             if l == L - 1:
58                 dA = A[l] - self.Y.T
59                 dZ = dA
60
61             else:
62                 dA = np.dot(self.w[l+1].T, dZ) * self.sigmoid_derivative(A[l])
63                 dZ = dA
64
65             dW[l] = np.dot(dA, A[l].T)
66             dB[l] = np.sum(dA, axis=1, keepdims=True) / m
67
68         return dW, dB

```

```

59     else:
60         dA = np.dot(self.w[l+1].T, dZ)
61         dZ = np.multiply(dA, self.sigmoid_derivative(A[l]))
62         dW[l] = 1 / m * np.dot(dZ, A[l-1].T) + (self.lmd * self.w[l])
63         dB[l] = 1 / m * np.sum(dZ, axis=1, keepdims=True)
64     return dW, dB
65
66 def update_param(self, dW, dB):
67     L = len(self.layers)
68     for l in range(1, L):
69         self.w[l] -= self.learning_rate * dW[l]
70         self.b[l] -= self.learning_rate * dB[l]
71
72 def cost(self, A):
73     m = len(self.Y.T)
74     L = len(self.layers)
75     cost = (1 / (2 * m)) * np.sum(np.square(A[L-1] - self.Y.T))
76     reg = 0
77     for l in range(1, L):
78         reg += (np.sum(np.square(self.w[l])))
79     reg_cost = reg * (self.lmd / (2 * m))
80     return cost + reg_cost
81
82 def predict(self, X):
83     self.X = X
84     _, A = self.forward_propagation()
85     L = len(self.layers)
86     prediction = A[L-1]
87     return prediction
88
89 def fit(self, x, y):
90     self.X = x
91     self.Y = y
92     cost = []
93     self.init_weights()
94     for step in range(self.steps):
95         Z, A = self.forward_propagation()
96         dW, dB = self.back_propagation(Z, A)
97         self.update_param(dW, dB)
98         costo = self.cost(A)
99         cost.append(costo)
100
101 def rmse2(self, pred, y):
102     n = len(y)
103     square = np.sqrt(np.average((pred - y) ** 2))
104
105     return np.sqrt(square)
106
107 def mae(self, pred, y):
108     return np.average(np.abs(pred - y))
109
110 def mse(self, pred, y):
111     square = (pred - y) ** 2
112     return np.average(square)
113
114 def rmse(self, pred, y):
115     return np.sqrt(self.mse(pred, y))
116

```

```
117  def mpe(self, pred, y):  
118      err = (pred - y)/ y  
119      return np.average(err)  
120  
121  def mape(self, pred, y):  
122      err = np.abs((pred - y)/ y)  
123      return np.average(err)  
124  
125  def r2(self, pred, y ):  
126      a = np.sum((pred - y) ** 2)  
127      b = np.sum((y - y.mean()) ** 2)  
128      return 1 - (a / b)
```

```

1 import numpy as np
2
3 np.random.seed(123)
4
5 class LinearRegression:
6     def __init__(self, nfeatures, steps=1000, a=0.05, lmd=2):
7         self.steps = steps
8         self.a = a
9         self.lmd = lmd
10        self.lmd_ = np.full(nfeatures, lmd)
11        self.lmd_[0] = 0
12        self.thetas = np.random.randn(nfeatures)
13
14    def fit_no_reg(self, X, y, X_val, y_val):
15
16        m = len(X)
17        cost_history = np.zeros(self.n_steps)
18        cost_history_val = np.zeros(self.n_steps)
19        theta_history = np.zeros((self.n_steps, self.theta.shape[0]))
20
21        for step in range(0, self.n_steps):
22            preds = np.dot(X, self.theta)
23            preds_val = np.dot(X_val, self.theta)
24
25            error = preds - y
26            error_val = preds_val - y_val
27
28            self.theta = self.theta - (self.learning_rate * (1/m) * np.dot(X.T, error))
29            theta_history[step, :] = self.theta.T
30            cost_history[step] = 1/(2*m) * np.dot(error.T, error)
31            cost_history_val[step] = 1 / (2 * m) * np.dot(error_val.T, error_val)
32
33        return cost_history, cost_history_val, theta_history
34
35    def fit(self, x, xva, y, yva):
36        m = len(x)
37        cost_history = np.zeros(self.steps)
38        cost_history_va = np.zeros(self.steps)
39        theta_history = np.zeros((self.steps, self.thetas.shape[0]))
40
41        for step in range(self.steps):
42            pred = np.dot(x, self.thetas)
43            pred_va = np.dot(xva, self.thetas)
44            error = pred - y
45            err_va = pred_va - yva
46
47            self.thetas = self.thetas - ((self.a / m) * (np.dot(x.T, error) + (self.thetas * self.lmd_)))
48            theta_history[step] = self.thetas
49            cost_history_va = (1 / (2 * m)) * (np.dot(err_va.T, err_va) +
50                                              (self.lmd * np.dot(self.thetas.T, self.thetas)))
51            cost_history[step] = (1 / (2 * m)) * (np.dot(error.T, error) +
52                                              (self.lmd * np.dot(self.thetas.T[1:], self.thetas[1:])))
53
54        return cost_history, cost_history_va, theta_history
55
56    def fit_stochastic(self, x, y, xva, yva):
57        cost_history = np.zeros(self.steps)
58        cost_history_va = np.zeros(self.steps)

```

```

59     thetas_history = np.zeros(self.steps, (self.thetas.shape[0]))
60     m = len(x)
61
62     for step in range(self.steps):
63         pred_va = np.dot(xva, self.thetas)
64         error_va = pred_va - yva
65         cost = 0
66         for i in range(m):
67             x_i = x[i, :]
68             y_i = y[i]
69             pred = np.dot(x_i, self.thetas)
70             error = pred - y_i
71             self.thetas = self.thetas - self.a * (np.dot(x_i.T, error) + np.dot(self.lmd_.T, self.thetas))
72             cost += 0.5 * (np.dot(error.T, error) + (self.lmd * np.dot(self.thetas.T, self.thetas)))
73
74         cost_history_va[step] = (1 / (2 * m)) * (np.dot(error_va.T, error_va) + self.lmd * np.dot(self.
75             thetas.T[1:], self.thetas[1:]))
76         cost_history[step] = (1 / m) * cost
77         thetas_history[step] = self.thetas
78     return cost_history, cost_history_va, thetas_history
79
80     def fit_minitbatch(self, x, y, xval, yval, batch_size = 100):
81         cost_history = np.zeros(self.steps)
82         cost_history_va = np.zeros(self.steps)
83         thetas_history = np.zeros(self.steps, (self.thetas.shape[0]))
84         m = len(x)
85
86         for step in range(self.steps):
87             pred_va = np.dot(xval, self.thetas)
88             error_va = pred_va - yval
89             cost = 0
90             for i in range(0, m, batch_size):
91                 x_i = x[i:i+batch_size]
92                 y_i = y[i:i+batch_size]
93                 pred = np.dot(x_i, self.thetas)
94                 error = pred - y_i
95                 self.thetas = self.thetas - self.a * (np.dot(x_i.T, error) + np.dot(self.lmd_.T, self.thetas))
96                 cost += 0.5 * (np.dot(error.T, error) + (self.lmd * np.dot(self.thetas.T, self.thetas)))
97                 cost_history_va[step] = (1 / (2 * m)) * (
98                     np.dot(error_va.T, error_va) + self.lmd * np.dot(self.thetas.T[1:], self.thetas[1:]))
99                 cost_history[step] = (1 / m) * cost
100                thetas_history[step] = self.thetas
101            return cost_history, cost_history_va, thetas_history
102
103    def predict(self, x):
104        x = np.c_[np.ones(x.shape[0]), x]
105        return np.dot(x, self.thetas)
106
107    def curve(self, x, y, xva, yva):
108        m = len(x)
109        cost_history = np.zeros(m)
110        cost_history_va = np.zeros(m)
111
112        for i in range(m):
113            x_i = x[:i+1]
114            y_i = y[:i+1]
115            c_h, cv, _ = self.fit(x, y, xva, yva)
116            cost_history[i] = c_h[-1]

```

```
116     cost_history_va[i] = cv[-1]
117
118     def metrics(self, x, y):
119         prev = self.predict(x)
120         mae = self.mae(prev, y)
121         mse = self.mse(prev, y)
122         mpe = self.mpe(prev, y)
123         mape = self.mape(prev, y)
124         r2 = self.r2(prev, y)
125         print("MAE: {} MSE: {} MPE: {} mape: {} R2: {}".format(mae, mse, mpe, mape, r2))
126
127
128     def mae(self, pred, y):
129         return np.average(np.abs(pred - y))
130
131     def mse(self, pred, y):
132         square = (pred - y) ** 2
133         return np.average(square)
134
135     def rmse(self, pred, y):
136         return np.sqrt(self.mse(pred, y))
137
138     def mpe(self, pred, y):
139         err = (pred - y) / y
140         return np.average(err)
141
142     def mape(self, pred, y):
143         err = np.abs((pred - y) / y)
144         return np.average(err)
145
146     def r2(self, pred, y):
147         a = np.sum((pred - y) ** 2)
148         b = np.sum((y - y.mean()) ** 2)
149         return 1 - (a / b)
150
151
```

```

1 import numpy as np
2
3
4 np.random.seed(123)
5
6
7 class LogisticRegression:
8
9     def __init__(self, learning_rate=1e-2, n_steps=2000, n_features=1, lmd=1):
10
11         self.learning_rate = learning_rate
12         self.n_steps = n_steps
13         self.theta = np.random.rand(n_features)
14         self.lmd = lmd
15         self.lmd_ = np.full((n_features,), lmd)
16         self.lmd_[0] = 0
17
18     def _sigmoid(self, z):
19
20         return 1 / (1 + np.exp(-z))
21
22     def fit(self, X, y, X_val, y_val):
23
24         m = len(X)
25         cost_history_train = np.zeros(self.n_steps)
26         cost_history_val = np.zeros(self.n_steps)
27         theta_history = np.zeros((self.n_steps, self.theta.shape[0]))
28
29         for step in range(0, self.n_steps):
30             preds = self._sigmoid(np.dot(X, self.theta))
31             preds_val = self._sigmoid(np.dot(X_val, self.theta))
32
33             error = preds - y
34
35             self.theta = self.theta - (self.learning_rate * (1 / m) * (np.dot(X.T, error)) + (self.theta.T * self.lmd_))
36             theta_history[step, :] = self.theta.T
37
38             cost_history_train[step] = -(1/m) * (np.dot(y.T, np.log(preds)) + np.dot((1-y.T), np.log(1-preds)))
39             cost_history_val[step] = -(1/m) * (np.dot(y_val.T, np.log(preds_val)) + np.dot((1-y_val.T),
40                                         np.log(1-preds_val)))
41
42         return cost_history_train, cost_history_val, theta_history
43
44     def fit_reg(self, X, y, X_val, y_val):
45
46         m = len(X)
47         cost_history_train = np.zeros(self.n_steps)
48         cost_history_val = np.zeros(self.n_steps)
49         theta_history = np.zeros((self.n_steps, self.theta.shape[0]))
50
51         for step in range(0, self.n_steps):
52             preds = self._sigmoid(np.dot(X, self.theta))
53             preds_val = self._sigmoid(np.dot(X_val, self.theta))
54
55             error = preds - y
56
57             self.theta = self.theta - (self.learning_rate * (1 / m) * (np.dot(X.T, error)) + (self.theta.T * self.lmd_))

```

```

57 )))  

58     theta_history[step, :] = self.theta.T  

59  

60     loss = -(1/m) * (np.dot(y.T, np.log(preds)) + np.dot((1-y.T), np.log(1-preds)))  

61     loss_validation = -(1 / m) * (  

62         np.dot(y_val.T, np.log(preds_val)) + np.dot((1 - y_val.T), np.log(1 - preds_val)))  

63     reg = (self.lmd / (2*m)) * np.dot(self.theta.T[1:], self.theta[1:])  

64  

65  

66     cost_history_train[step] = loss + reg  

67     cost_history_val[step] = loss_validation + reg  

68  

69     return cost_history_train, cost_history_val, theta_history  

70  

71 def _predict_prob(self, X):  

72  

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

74  

75 def predict(self, X, threshold):  

76     """  

77     perform a complete prediction about X samples  

78     :param X: test sample with shape (m, n_features)  

79     :param threshold: threshold value to disambiguate positive or negative sample  

80     :return: prediction wrt X sample. The shape of return array is (m,)  

81     """  

82  

83     Xpred = np.c_[np.ones(X.shape[0]), X]  

84     return self._predict_prob(Xpred) >= threshold  

85  

86 def confusion_matrix(self, xtest, treshold, ytest):  

87     tp, fp, tn, fn = 0, 0, 0, 0  

88     i = 0  

89     predicted = self.predict(xtest, treshold)  

90     for pred in predicted:  

91         if pred == ytest[i]:  

92             tp += 1  

93         elif pred == 1 and ytest[i] == 0:  

94             fp += 1  

95         elif pred == 0 and ytest[i] == 0:  

96             tn += 1  

97         elif pred == 0 and ytest[i] == 1:  

98             fn += 1  

99         i += 1  

100    # sensitivity = recall = tp / tot positivi in y  

101    recall = tp / (tp + fn)  

102  

103    # accuracy true / tutto  

104    accuracy = (tp + tn) / (tp + tn + fp + fn)  

105  

106    # precision = positivi azzeccati / tot positivi predetti  

107    precision = tp / (tp + fp)  

108  

109    # specificity = tn rate = negativi azzeccati / negativi reali  

110    specificity = tn / (tn + fp)  

111  

112    # error rate = falsi / tutto  

113    errorrate = (fp + fn) / (tp + fp + tn + fn)
114

```

```
115     # fmeasure = 2 * (precision * recall) / (precision + recall)
116     fmeasure = 2 * (precision * recall) / (precision + recall)
117     return recall, accuracy, precision, specificity, errorrate, fmeasure
118
```